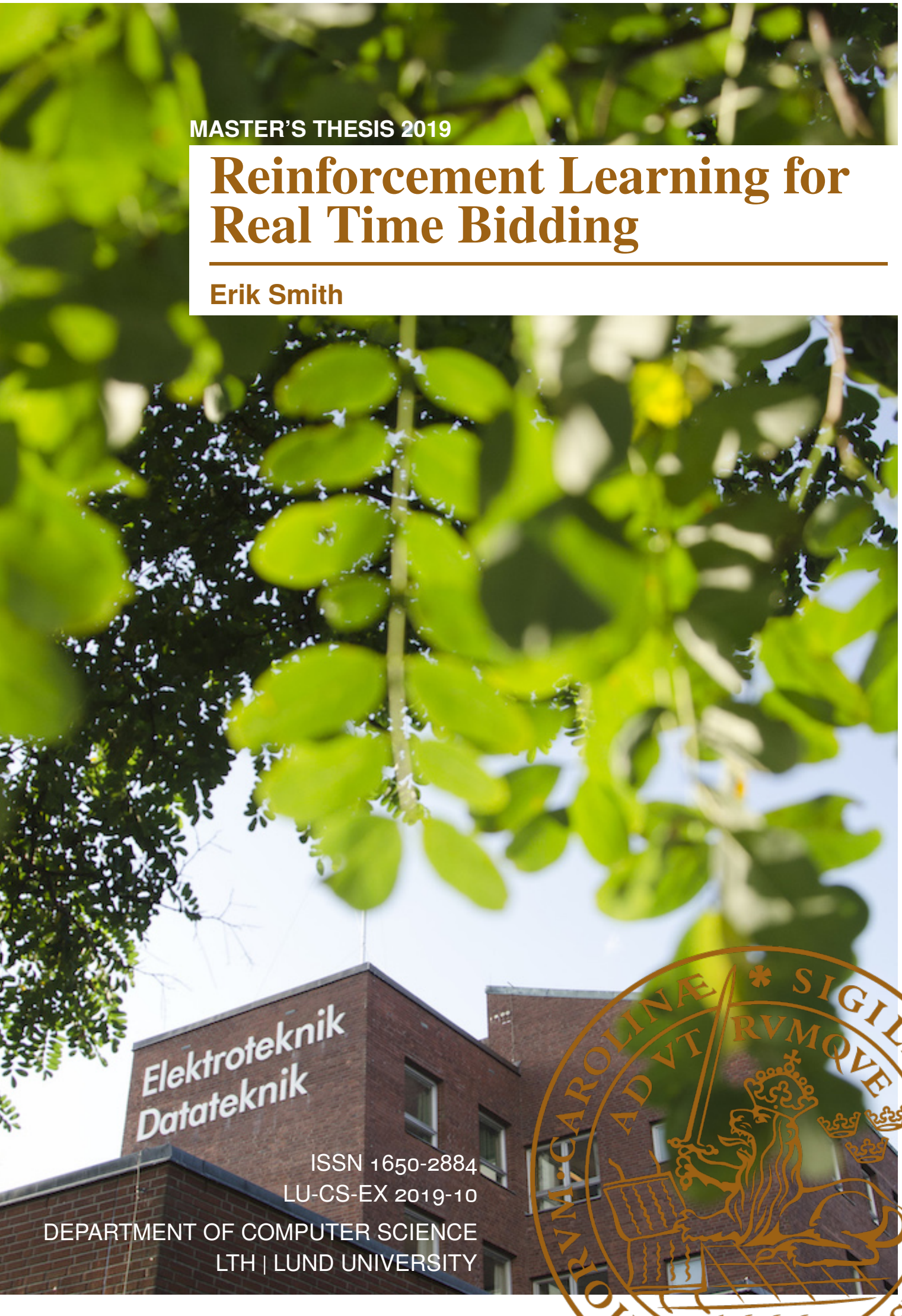


MASTER'S THESIS 2019

Reinforcement Learning for Real Time Bidding

Erik Smith

ISSN 1650-2884
LU-CS-EX 2019-10

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-10

Reinforcement Learning for Real Time Bidding

Erik Smith

Reinforcement Learning for Real Time Bidding

Erik Smith
me@eriksmith.se

June 24, 2019

Master's thesis work carried out at Emerse Sverige AB.

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Rasmus Larsson, rasmus.larsson@emerse.com
Carl-Johan Grund, carl-johan.grund@emerse.com

Examiner: Elin Anna Topp, elin_anna.topp@cs.lth.se

Abstract

When an internet user opens a web page containing an advertising slot, how is it determined which ad is shown? Today, the most common software-based approach to trading advertising slots is real time bidding: as soon as the user begins to load the web page, an auction for the slot is held in real time, and the highest bidder gets to display their advertisement of choice. But each bidder has a limited budget, and strives to spend it in a manner that maximizes the value of the advertisement slots bought. In this thesis, we formalize this problem by modelling the bidding process as a Markov decision process. To find the optimal auction bid, two different solution methods are proposed: value iteration and actor–critic policy gradients. The effectiveness of the value iteration Markov decision process approach (versus other common baselines methods) is demonstrated on real-world auction data.

Keywords: Reinforcement learning, Markov decision process, value iteration, policy gradient, real time bidding

Acknowledgements

I would like to thank Pierre Nugues and Elin Anna Topp at the Department of Computer Science at Lund University for all their helpful guidance and feedback. I would also like to express my heartfelt gratitude to Rasmus Larsson and Carl-Johan Grund at Emerse for providing me with this thesis opportunity, and for sharing their experience and expertise. Last but not least, a special thanks to Pontus Ericsson and Carl Dahl.

Contents

Chapter 1

Introduction

1.1 Real time bidding

When an internet user opens a web page containing advertising slots, how is it determined which ad is shown? Today, the most common way is through programmatic advertising, i.e. using software to trade advertising slots. In turn, the most common type of programmatic advertising is *real time bidding (RTB)*: a process where available advertising slots are auctioned off as soon as a user loads the web page containing them.

The essential steps of a single real time bidding auction are visualized in Figure 1.1. For clarity, we divided the process into two main parts. Figure 1.1a shows the bidding part, where the main steps are:

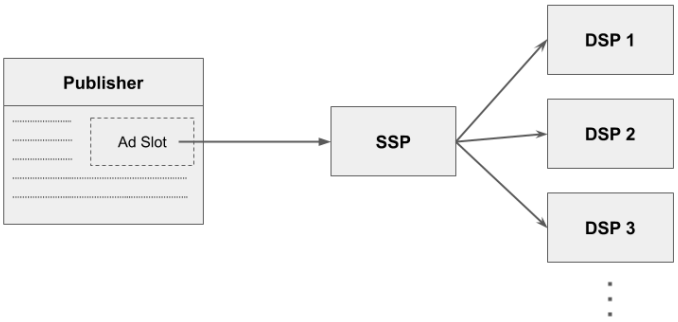
i. A user visits a publisher's website or app. When the user begins to load a page containing an eligible advertisement slot, the supply side platform (SSP) is almost instantly notified.

ii. A supply side platform holds an auction for the ad slot by sending out requests for bids to *several different bidders*, called demand side platforms (DSPs). The bid requests contain information about the ad slot and target user, and reach the DPSs approximately 10 milliseconds after the user loads the web page.

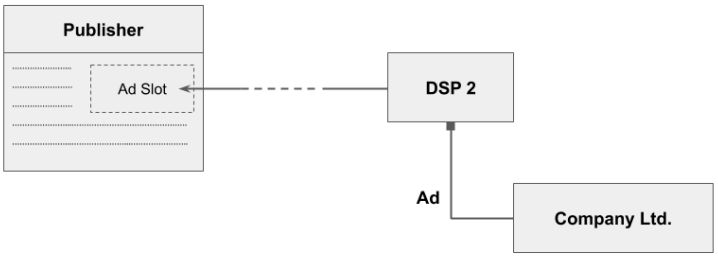
iii. Each DSP responds with a single bid (in USD) for the ad slot. The responses have to arrive, at the latest, roughly 100 ms after the bid request is relayed.

Figure 1.1b illustrates the steps which occur after the bids have been submitted:

iv. The supply side platform compares the bids and awards the ad slot to to the highest bidder. They can now, barring some technical details, place their advertisement of



(a) The bidding step.



(b) The serving step.

Figure 1.1: A flowchart visualizing the real time bidding auction process for a single advertisement slot. The bidding step starts as soon as a user begins to load a web page. A winner is determined and an ad is served in less than 100 ms.

choice in the slot. However, the actual price paid by the highest bidder is not his own bid. Instead, most SSPs employ *second price auctions*: the highest bidder wins, but only has to pay an amount equal to the second highest bid. Second price auctions are used to incentivize bidders to bid their true value.[1] Note that the final price is only revealed to the winner, and that competitors' bids are never revealed to any of the bidders.

v. The DSPs are not the advertisers (e.g. retailers, organisations) themselves. The role of a DSP is to handle the bidding process/infrastructure on behalf of their clients, which are the actual advertisers. Thus the chosen advertisement is not for the DSP itself, but from one of its clients.

Emerse AB owns and operates a demand side platform, and is continuously bidding for advertisement slots on behalf of its clients. The main problem is that each advertising campaign has a limited budget, and Emerse wants to spend it in a manner that maximizes the value of the purchased ad slots.

1.2 Approach

In this thesis, we utilize reinforcement learning to build two different bidding algorithms. The goal of these is offer bids in a manner that, compared to Emerse's current algorithms, increases the value gained from the limited campaign budgets. We formalize the bidding process by modelling it as a Markov decision process (MDP). We then propose two different solution methods for finding the most suitable bid for each auction: (i) value iteration and (ii) actor–critic policy gradients. The methods are evaluated using real-world historical auction data from Emerse AB. Assigning an all-encompassing numerical value to an ad slot is a futile task, wherefore we will primarily focus on two performance measures for our bidding methods: total number of clicks on bought ad slots and the cost per click. To summarize the results: the value iteration method successfully outperforms the baseline methods, while the actor–critic policy gradient approach suffers from convergence problems.

1.3 Contributions and problem statement

The contributions of this thesis are mainly practical; the theory and methods used are well-known. The literature study that we carried out made it apparent how the MDP for the bidding process should be formulated (see for example Wu et al. (2018)) and that it was possible to solve the RTB-specific formulation of the MDP with value iteration (Cai et al., 2017). Both value iteration and the baseline methods require us to estimate certain parameters, and ways to carry out these estimations were suggested in Zhang et al. (2014). We considered and combined the information gained from the literature study and implemented the value iteration solution and baseline methods on/for Emerse's data.

[1] Details: Vickrey auction on Wikipedia.

On the other hand, we were not able to find any papers applying a policy gradient based approach to RTB. Thus, our actor–critic policy gradient approach is the (apparent) first attempt at doing so. Even though our implementation failed to converge to a satisfactory solution, we hope that our efforts will facilitate the success of any further attempts at a policy gradient based approach to RTB.

The problem statement has a different focus for each of our two chosen solution methods:

i. *The value iteration solution*
Since the value iteration solution had previously been successfully applied to RTB, is it possible to implement it and reproduce its good results on Emerse's (not identical & more recent) data?

ii. *The actor–critic policy gradient solution*
The actor–critic policy gradient approach to RTB has not been previously attempted, thus we aim to answer the following questions. Is it a generally viable approach for RTB? Will a simple implementation of this method be fast enough to handle RTB? Will its ability to naturally handle continuous action spaces facilitate convergence to a satisfactory solution? Will the problem of sparse rewards be as impactful as in previously attempted Deep Q-Learning based methods?

1.4 Related work and other applications

There have been recent efforts to apply various machine-learning methods (reinforcement learning included) to real time bidding. Wu et al. (2018) proposed a model-free approach with Deep Q-Learning (Mnih et al., 2015), and also implemented a neural network solution for handling sparse rewards. Cai et al. (2017) took the value iteration approach, leveraging neural network value function approximation to achieve large-scale viability. Jin et al. (2018) utilized clustering methods to assign the most suitable bidding agent the each cluster of advertisers. Wang et al. (2017) used an asynchronous stochastic Deep Q-Learning method to successfully learn to bid from raw high-level semantic information. Lastly, Zhang et al. (2014) provided a first complete public RTB dataset and appropriate methods for benchmarking RTB performance.

Since real time bidding is a process unique to internet advertising, direct alternative applications of our solution implementations are very limited. However, reinforcement learning is a very general framework (see Section 2.1) and can, along with appropriate solution algorithms, therefore be applied to a wide variety of problems. Some of these arise in environments similar to RTB, while others arise in areas of a very different nature. The financial markets could in some sense be considered similar to the RTB environment. There, an actor–critic method called Deep Deterministic Policy Gradient (Lillicrap et al., 2015) has been successfully applied to portfolio management (stock trading); see for example Xiong et al. (2018) or Yu et al. (2019). For an extensive survey of reinforcement learning in the financial markets, see Fischer (2018).

As mentioned, reinforcement learning algorithms also have applications in areas not at all similar to RTB; as an example, take the area of robotics. In recent papers on how

to teach robots appropriate control schemes, we find reinforcement learning solutions. For example, Hwangbo et al. (2019) utilized a custom reinforcement learning setup to train a four-legged robot in a simulated environment, and then successfully transferred the knowledge to a real-world scenario. As another example, Huang et al. (2019) proposed a new reinforcement learning approach for teaching a robot to complete its tasks in a gentle manner.

Another area where reinforcement learning algorithms excel is in playing games. Perhaps the most famous example arose in March 2016, when DeepMind[2] made headlines after beating the 18-time world champion of the board-game Go in a five-game match.[3] To do so, they utilized a reinforcement learning algorithm (Silver et al., 2016). From there, the algorithm underwent further development to allow it to teach itself from scratch (Silver et al., 2017, e.g.). The most recent state-of-the-art version of the algorithm is, starting from scratch, able to to teach itself to beat world champions in both chess, Go, and shōgi (Silver et al., 2018).

1.5 Outline

Chapter 2 introduces the general idea behind reinforcement learning, and the theoretical foundation of the value iteration and actor–critic policy gradient methods. Chapter 3 begins by explaining our reasoning behind choosing those two methods in particular. We then describe how the theory presented in Chapter 2 is utilized in our RTB setting. We also give implementation-specific details of our approach. Chapter 4 presents the test results and associated discussion. Lastly, final thoughts and a summary of our findings are presented in Chapter 5.

[2] www.deepmind.com

[3] See e.g. "AlphaGo versus Lee Sedol" on Wikipedia.

1.6 Terminology quick-reference

As a closing note in this section, we list common jargon and acronyms used in the digital advertising industry (Table 1.1).

Term	Description
Impression	A successful purchase and delivery of *one* advertisement slot, i.e. one advertisement view.
Creative	An advertisement.
Advertiser	Someone who wishes to distribute advertisements.
Publisher	A provider of advertisement slots, e.g. a newspaper.
Campaign	An advertising campaign initiated by an advertiser. Usually active for pre-defined duration and with a pre-defined budget.
DSP	Demand side platform. Bids on RTB auctions on behalf of their clients (or themselves). Emerse AB provides such a platform.
SSP	Supply side platform, sells ad slots on behalf of publishers. Plays the role of the auctioneer in RTB auctions.
CTR	Click-through rate is the probability of an impression generating a click. Can also be used in the context of whole campaigns or groups of users.
CPM	Cost per mille is the average cost of one thousand impressions, often used in the context of a campaign.
CPC	Cost per click is the total advertisement cost divided by the number of generated clicks.

Table 1.1: Explanations of common terms used in the advertising industry.

Chapter 2

Theory

In this section, we present the theory behind the methods used in our report. We aim to give the reader both a qualitative and quantitative understanding of the methods used. First we will describe the general idea behind reinforcement learning (RL). Then we will introduce the standard mathematical framework, i.e. the Markov decision process (MDP). Next we present the theoretical foundation of the value iteration and actor–critic policy gradient approaches. As a closing note, we compare the two methods and discuss their advantages and disadvantages. Unless otherwise stated, we refer the reader to Sutton and Barto (2018) for details on the theory.

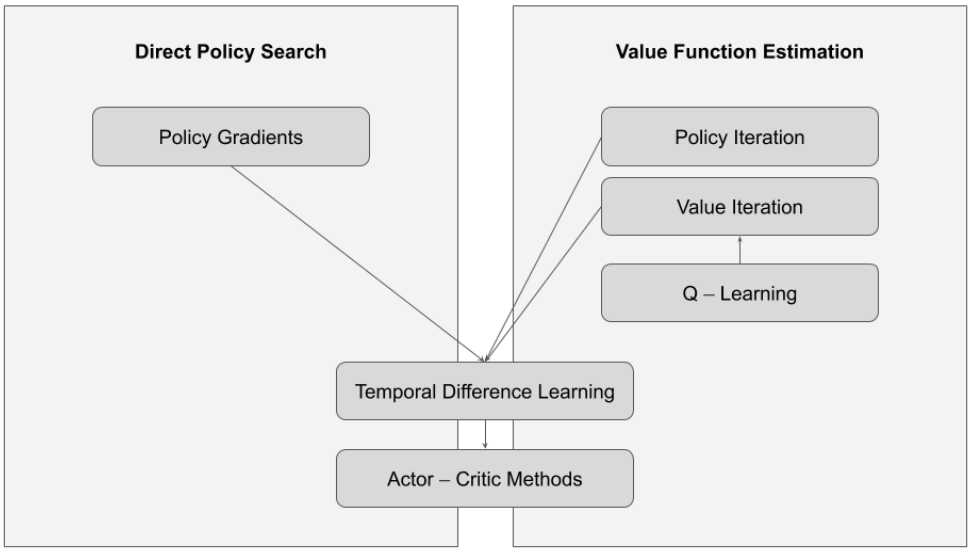


Figure 2.1: An overview of a few selected reinforcement learning methods.

2.1 Reinforcement Learning

In general, one can divide the field of machine learning into three main paradigms: supervised learning, unsupervised learning, and reinforcement learning. While there is overlap between the three categories, the definition of reinforcement learning is still fairly clean-cut: it is the area of machine learning studying how an agent should act in its environment in order to maximize some performance measure.

Figure 2.2 shows a standard reinforcement learning process. Starting from the depicted dotted line and proceeding clockwise, the steps of the cycle are as follows:

1. We have a given state and reward from previous interactions with the environment. The agent uses this information to decide on its next action. Index this action by i.

2. The agent then interacts with the environment by executing the chosen action. This yields a new state and reward.

3. The new state and reward is in turn used to decide action $i + 1$, and the cycle continues...

The general goal of reinforcement learning is to build an agent that through interactions with the environment *learns to perform actions that will maximize the cumulative reward over time*. To successfully reach its goal the agent needs to both exploit its current knowledge/experience, and gain new knowledge by freely exploring the environment.

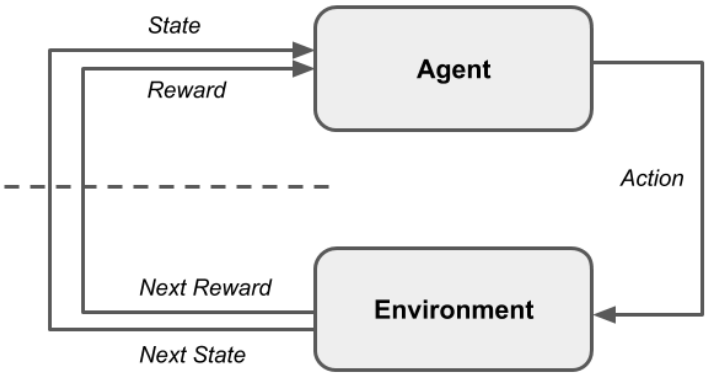


Figure 2.2: The reinforcement learning process. An agent interacts with the environment through an action, usually chosen with the current state and previous reward as input. The interaction with the environment yields a new state and reward, which in turn is used to choose a new action, and the cycle continues. The general idea is that through exploring the environment and exploiting past experience, the agent will eventually learn which actions result in the largest reward over time.

2.2 The Markov Decision Process

A Markov decision process is an extension of the Markov chain concept, so let us start from there. A Markov chain is simply a probability based model describing a series of events or states. The probability of ending up in state X after the next step *only* depends on where you are now. Conversely, the so called *transition probabilities* do *not* depend on the path that brought you to your current state. This property is often referred to as memorylessness or the Markov property. Figure 2.3 shows a simple Markov chain with only two states. Note that the transition probabilities from each state all sum to one.

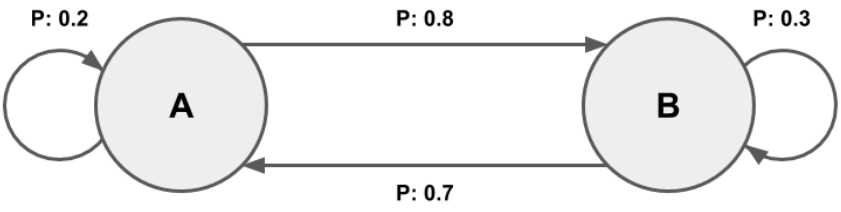


Figure 2.3: A visualization of a simple Markov Chain. Arrows and labels indicate possible transitions and their probabilities. Note that the transition probabilities from each state all sum to one.

The *Markov decision process* (MDP) extends the Markov chain by adding actions and rewards. Choosing actions is now allowed, and one goes from modelling a series of events to *modelling a decision making process*. A MDP is, in its entirety, defined by four objects:

Object 1: *State space*, S
This is a finite set of all possible states s. A state can be defined in various ways; details for our implementation are given in Sections 3.1.1 and 3.2.

Object 2: *Action space*, A or A_s
This is a finite set of available actions. The available actions can depend on which state s we are in, or alternatively they could stay the same for all states. Both discrete and continuous action spaces are allowed.

Object 3: *State transition function*, $P_a(s, s')$
Yields the probability of ending up in state s' after taking action a in state s.

Object 4: *Expected immediate reward function*, $R_a(s, s')$
This is the reward received when transitioning to state s' after taking action a in state s.

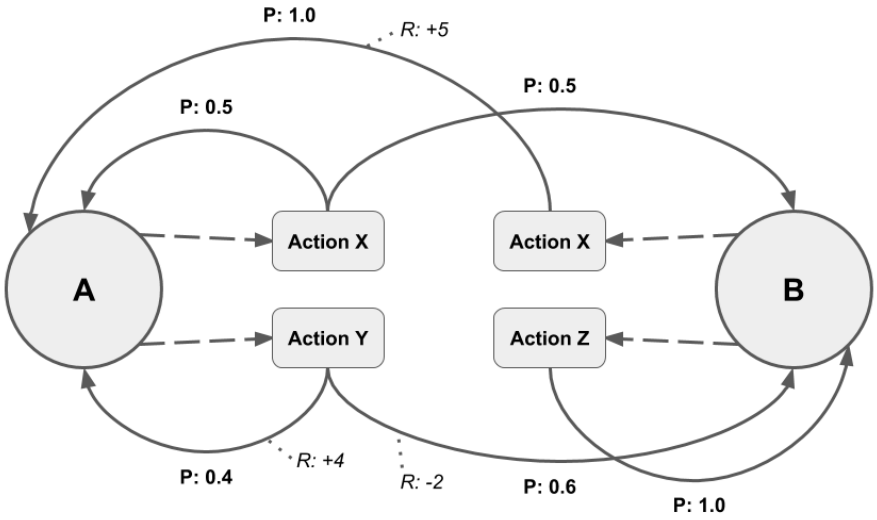


Figure 2.4: A visualization of a simple Markov decision process. Compare with Figure 2.3 and note the addition of choices (via actions) and rewards. To reduce clutter we did not mark every transition with a reward. One could assume that it is zero if not explicitly drawn. Note that the transition probabilities from each action all sum to one.

In addition to these objects, one also needs to set a *discount factor* $0 < \gamma \leq 1$ and a *horizon* $H \leq \infty$. Figure 2.4 illustrates a simple Markov decision process.

The last term we need to introduce is *policy*, denoted by π or $\pi(s)$. In simple terms, it is the set of rules the agent follows when deciding its actions. There are two types of policies:

- *Deterministic policy*
  Outputs a single action for each input state.
- *Stochastic/probabilistic policy*
  Outputs the probability of choosing each available action in the input state. The output is usually a probability distribution if A is continuous, and probabilities for each action if A is discrete.

Once the Markov decision process framework is established, the main goal remains: find the best decision in each state. As stated in the previous section, by "best decision in each state" one usually means the policy that results in the largest expected total (discounted) future reward. Such a policy is called an *optimal policy*, denoted by $\pi^*(s)$. To summarize in a more formal manner, a solution algorithm's main goal is to find the policy $\pi(s)$ that maximizes

$$\sum_{i=0}^{H} \gamma^i \cdot R_{a_i}(s_i, s'_{i+1}) \ , \quad \text{where } a_i = \pi(s_i) \ . \tag{2.1}$$

Note that combining a Markov decision process with a deterministic policy, fixes the action for each state and reduces it to a Markov chain (with rewards).

2.3 Methods for finding the optimal policy

Numerous methods for finding the optimal policy exist, Figure 2.1 only lists a few. In this thesis, we decided to use two different methods: value iteration and actor–critic policy gradients. As for why we chose these two methods in particular, see Section 3.1. Before we introduce the two chosen methods, a short paragraph on the most naive approach, i.e. *brute force*.

The brute force approach to finding the optimal policy would go as follows: for each possible policy, follow it for some time and record rewards; then pick the one with the best reward. Unsurprisingly, this method is rarely viable since the number of possible policies can be extremely large. Consequently, more sophisticated solution/search algorithms are required.

2.3.1 Value Functions and Value Iteration

As shown in Figure 2.1, value iteration belongs to the class of solution algorithms which perform some kind of value function estimation. But what is a value function? In short, the *value function* of some policy π *outputs the total expected (discounted) reward* when starting in state s and following π. One can intuitively think of it as a measure of how good it is to be in state s if we follow the current policy. Using the established MDP framework, we mathematically define the value function (of some policy π) as

$$V^{\pi}(s) = \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V(s')\right) \ , \text{where } a = \pi(s) \ . \tag{2.2}$$

Note the recursive nature of Equation 2.2, and that we choose actions by querying the policy with the current state as input. While Equation 2.2 is valid for any policy, the most interesting one is the value function acquired while following the optimal policy π^*. This optimal value function is denoted by $V^*(s)$, and can mathematically be defined as

$$V^*(s) = V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s) \ . \tag{2.3}$$

When working with reinforcement learning, the progression is often divided into segments called *episodes*. For example, if one is teaching an agent to play Tetris it is natural to take each playthrough attempt (from first block to failure) as an episode. Since the Tetris agent can survive a longer or shorter time, the episodes will be of varying length. For some environments it is more suitable to choose episodes of a fixed length. Such is the case in this thesis; to be able to balance the budget usage, our algorithms need to know how many auctions they have left "to work with". Since the constant flow of auctions (bid requests) has no natural episode length, we introduced an artificial one by segmenting the bidding process into fixed episodes of a 1000 auctions each. The total advertising budget budget was distributed over the episodes. We chose an episode length of 1000 auctions based on the fact that (judging from historical data) roughly one out of every 1000 ads are clicked. Note that after implementing our solution, we tried varying the episode length to 500, 2000, and 10000. The final results were ever so slightly worse for an episode length of 500, while they did not markedly improve for neither lengths of 2000 nor 10000. Thus, we surmised that keeping the episode length at 1000 auctions would strike a satisfactory balance between performance and computational complexity.

For a (general) process with fixed-length episodes, denote the number of step left in the episode with k. One then usually adds an extra index k to Equation 2.3: from $V^*(s)$ to

$$V_k^*(s) = \max_{a} \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V_{k-1}(s')\right) \ . \tag{2.4}$$

This equation still measures the total expected reward when following π^*, but now also considers that we have k steps left. Due to the recursive nature of Equation 2.3, to calculate $V_k^*(s)$ one also needs to calculate $V_{k-1}^*, V_{k-2}^*, \ \ldots \ , V_1^*, V_0^*$.

Equation 2.4 is used in the *value iteration* solution method proposed by Bellman (1957), shown in pseudo-code here:

This is a *bottom up dynamic programming* algorithm: we start with the smallest subproblem (i.e. V_0^*), memorize the solution, and use it to solve $V_1^*(s)$ without re-doing the calculation for $V_0^*(s)$. By repeating this scheme, we work our way up the recursion (from "the bottom") until V_0^* does not change anymore (convergence). Although here we are content as long as the difference is less than θ, which is a laxer (more reasonable) condition than a strict equality.

Once the calculation converges, we have found the optimal value function $V^*(s)$ which satisfies Equation 2.3 (for the infinite horizon problem with discounted rewards). No matter what initial value for V_0^* we pick, the algorithm will eventually converge (Puterman, 1994, pp. 161-163).

Algorithm: Value Iteration
Input: H, γ, $P_a(s, s')$, and $R_a(s, s')$ from MDP framework.
Output: Optimal policy $\pi^*(s)$ for horizon H.
For all states s in S, initialize $V_0^*(s) = 0$.
$k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  for each state s **do**
    $V_k^*(s) \leftarrow \max_a \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V_{k-1}(s')\right)$
  end
until $|V_k^*(s) - V_{k-1}^*(s)| < \theta \; \forall \; s$ **or** $k = H$;
for each state s **do**
  $\pi_k^*(s) \leftarrow \arg\max_a \sum_{s'} P_a(s, s') \left(R_a(s, s') + \gamma V_{k-1}^*(s')\right)$
end

As the reader may have noted, the optimal policy is only derived *once convergence in the value function is reached*. This is a main characteristic of value iteration versus other exact solution method like policy iteration. Once we have $V^*(s)$, we can derive $\pi^*(s)$ for any state s by calculating

$$\pi^*(s) = \arg\max_{a \in A_s} \sum_{s' \in S} P_a(s, s') \left(R_a(s, s') + \gamma V^*(s')\right) \; . \tag{2.5}$$

2.3.2 Policy Gradients – Vanilla and Actor-Critic

Direct policy search methods, in contrast to the previous section, do not concern themselves with estimating a value function. As hinted in Figure 2.1, they instead work directly with the policy. Standard policy gradients belong to this group of methods. This algorithm, without any extra frills, is often referred to as "vanilla" policy gradients. It is important to understand this method, since the mathematical step from the vanilla version to the actor–critic version is very small. We will for this reason describe and derive the vanilla version, and then take the (small) leap to actor–critic.

In policy gradient methods, actions u are generated by sampling from a probability distribution. The distribution we sample from is still called a policy, and can still depend on the current state. Formally noted, $u_t \sim \pi_\theta(u_t|s_t)$. Note the change of notation for an action: from a to u. The θ in π_θ arises from the assumption that we can describe our policy by a set of parameters. These are collected into a vector denoted by θ. An advantage of sampling actions from a probability distribution is that it naturally incorporates exploration of the environment which, as mentioned in previous sections, is a necessity for learning.

In vanilla policy gradients, *the policy is represented by a neural network*.[1] It takes the current state s_t as input and outputs the mean and standard derivation of a normal distribution, which we subsequently sample (probabilities of) actions from. Thus, θ contains the

[1] Often referred to as the "policy network".

weights and biases of the neural network. The main goal of reinforcement learning, i.e. finding a policy maximizing the total discounted reward, is now a problem of finding the right network parameters. As the method's name suggests, this is done though gradient descent.

The upcoming derivations follow Sutton and Barto (2018, pp. 324-336) and Sutton et al. (2000). To approach the gradient descent problem mathematically, let us begin by defining utility $U(\theta)$ as

$$U(\theta) = \mathbb{E}\left[\sum_{t=0}^{H} R(s_t, u_t; \pi_\theta)\right] = \sum_{\tau} P(\tau; \theta) R(\tau) \,. \tag{2.6}$$

Equation 2.6 introduces new notation which will be used this section:

- τ is a sample path (trajectory) of an episode, i.e. a series of state-action pairs.
- $R(\tau)$ is the total reward of a trajectory: $R(\tau) = \sum_{t=0}^{H} R(s_t, u_t)$. Note that we overload the notation to improve readability.
- $P(\tau; \theta)$ indicates the probability of a sample path under the policy determined by θ. Upcoming derivations will show that this does not need to be explicitly stated.

Now take the gradient of Equation 2.6:

$$\begin{aligned}
\nabla_\theta U(\theta) &= \nabla_\theta \sum_{\tau} P(\tau; \theta) R(\tau) = \sum_{\tau} \nabla_\theta P(\tau; \theta) R(\tau) \\
&= \sum_{\tau} \frac{P(\tau; \theta)}{P(\tau; \theta)} \nabla_\theta P(\tau; \theta) R(\tau) = \sum_{\tau} P(\tau; \theta) \frac{\nabla_\theta P(\tau; \theta)}{P(\tau; \theta)} R(\tau) \\
&= \left[\text{Recognize } \frac{\partial}{\partial x} \log f(x) = \frac{1}{f(x)} \frac{\partial}{\partial x} f(x) \,, \text{ use it to rewrite.}\right] \\
&= \sum_{\tau} P(\tau; \theta) \, \nabla_\theta \log P(\tau; \theta) \, R(\tau) \,.
\end{aligned} \tag{2.7}$$

Equation 2.7 is equivalent to the expected value $\mathbb{E}\left[\nabla_\theta \log P(\tau; \theta) \, R(\tau)\right]$, which can be empirically estimated by following policy π_θ for n sample paths, and then calculating

$$\nabla_\theta U(\theta) \approx \frac{1}{n} \sum_{i=0}^{n} \nabla_\theta \log P(\tau^{(i)}; \theta) \, R(\tau^{(i)}) \equiv \hat{g} \,, \tag{2.8}$$

where sample path i is denoted by $\tau^{(i)}$. Intuitively one can think of Equation 2.8 as a gradient that tries to *increase the probability of paths with positive reward*, and decrease probabilities of paths with a negative $R(\tau)$.

We previously stated that one would not have to explicitly express $P(\tau^{(i)}; \theta)$, although Equation 2.8 might lead us to think that we have to. But it turns out we can circumvent this by assuming that the dynamics are Markovian, and then separating P into states and actions

in the following manner:

$$\begin{aligned}
\nabla_\theta \log P(\tau^{(i)};\theta) &= \nabla_\theta \log \left(\prod_{t=0}^{H} \underbrace{P(s_{t+1}^{(i)}|s_t^{(i)},u_t^{(i)})}_{\text{Dynamics model}} \cdot \underbrace{\pi_\theta(u_t^{(i)}|s_t^{(i)})}_{\text{Policy}} \right) \\
&= \nabla_\theta \Bigg(\underbrace{\sum_{t=0}^{H} \log P(s_{t+1}^{(i)}|s_t^{(i)},u_t^{(i)})}_{\text{Does not depend on } \theta!} + \sum_{t=0}^{H} \log \pi_\theta(u_t^{(i)}|s_t^{(i)}) \Bigg) \\
&= \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)}) \qquad (2.9)
\end{aligned}$$

Again, note the Markovian assumption in the first step. Equation 2.9 can be calculated by applying backpropogation to the policy network. By substituting Equation 2.9 into Equation 2.8, we acquire an unbiased[2] estimate of $\nabla_\theta U(\theta)$, which does not require us to explicitly express $P(\tau^{(i)};\theta)$:

$$\begin{aligned}
\nabla_\theta U(\theta) \approx \hat{g} &\equiv \frac{1}{n} \sum_{i=0}^{n} \nabla_\theta \log P(\tau^{(i)};\theta)\, R(\tau^{(i)}) \\
&= \frac{1}{n} \sum_{i=0}^{n} \left(\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)}) \right) R(\tau^{(i)}) \qquad (2.10)
\end{aligned}$$

Although this is not quite enough; even though $\hat{g}$ is unbiased, it is very noisy. To make it viable for most real-world applications, we need to reduce the noise. This will be achieved through two methods: by introducing a *baseline*, and by utilizing the temporal structure of the sample paths. Adding a baseline is simply a matter of adjusting the reward part of Equation 2.8, e.g. by subtracting some value b in the following manner:

$$\nabla_\theta U(\theta) \approx \hat{g} \equiv \frac{1}{n} \sum_{i=0}^{n} \nabla_\theta \log P(\tau^{(i)};\theta) \left(R(\tau^{(i)}) - b \right) . \qquad (2.11)$$

A simple and intuitive choice for b is the average reward for a sample path (empirically estimated). This choice modifies the PG algorithm to increase/decrease the probability of paths with *better/worse than average* reward, instead of considering purely positive/negative reward. Note that the estimate in Equation 2.11 is still unbiased (, p. 331). There are of course other more advanced (better) ways of choosing b. We will come back to these, but first we will show how one can utilize the temporal structure of the sample paths to further reduce the variance. Take $\left(R(\tau^{(i)}) - b \right)$ from Equation 2.11 and expand it (see Eq. 2.6):

$$\begin{aligned}
\nabla_\theta U(\theta) \approx \hat{g} &\equiv \frac{1}{n} \sum_{i=0}^{n} [\cdots] \left[R(\tau^{(i)}) - b \right] \\
&= \frac{1}{n} \sum_{i=0}^{n} [\cdots] \Bigg[\underbrace{\sum_{k=0}^{t-1} R(s_k^{(i)},u_k^{(i)})}_{\text{Does not depend on } u_t^{(i)}.} + \sum_{k=t}^{H-1} R(s_k^{(i)},u_k^{(i)}) - b \Bigg]
\end{aligned}$$

[2] Unbiased estimate means $\mathbb{E}[\hat{g}] = \nabla_\theta U(\theta)$.

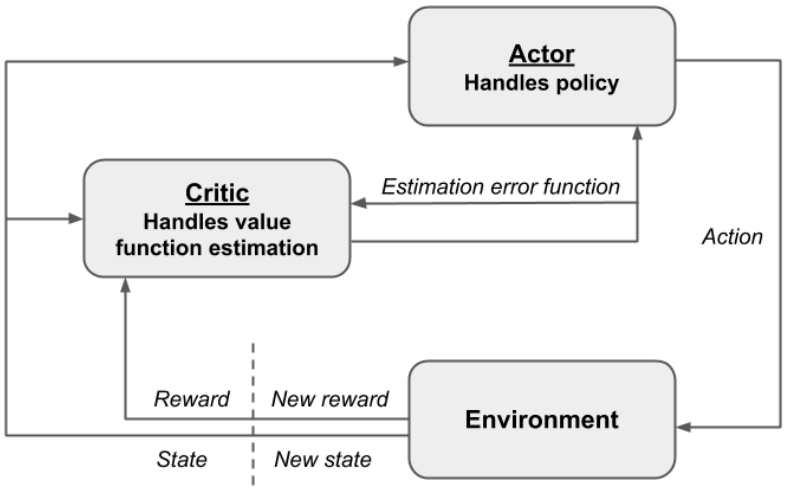


Figure 2.5: The general idea of the actor–critic policy gradient method. Evaluation of the current policy is handled by the critic. The actor implements and executes the current policy, and is trained with the help of the critic.

Removing the sum (part of the sample path) that does not depend on the current action can help in lowering variance. Doing so, we are left with (cf. Equation 2.10)

$$\nabla_\theta U(\theta) \approx \frac{1}{n}\sum_{i=0}^{n}\left(\sum_{t=0}^{H}\nabla_\theta \log \pi_\theta(u_t^{(i)}|s_t^{(i)})\right)\overbrace{\left(\underbrace{\sum_{k=t}^{H-1}R(s_k^{(i)},u_k^{(i)})}_{R_t} - b(s_t^{(i)})\right)}^{\text{Advantage, } A_t}. \tag{2.12}$$

Equation 2.12 is the full vanilla policy gradient (nb. definitions vary). As mentioned, there are better ways to choose the baseline $b(s_t^{(i)})$ than the empirical average return. To improve learning, we instead use a *value function estimation as the baseline*, i.e. $b(s_t^{(i)}) = V^\pi(s_t^{(i)})$. The modification carries us away from pure policy search methods to the realm of hybrid methods (see Figure 2.1).

As we stated in the beginning of this section, the step from here to the actor–critic policy gradients (ACPG) family of methods is rather small. The main characteristic of actor–critic methods is that they use more *advanced bootstrapping methods to estimate* R_t (compared to the sum in Equation 2.12). Figure 2.5 illustrates the (general) actor–critic policy gradient method.

There are various ways to estimate and bootstrap R_t and/or A_t. Some are fairly simple, e.g. advantage actor–critic (A2C), while other methods like A3C (Mnih et al., 2016) or GAE (Schulman et al., 2015) are more complex. We will (rather concisely) present A2C, since it is the method used in our implementation. In A2C, we use the following advantage

calculation:

$$A_t = A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)\ , \tag{2.13}$$

where $Q^\pi(s,a)$ is the standard Q-function, i.e. the value of taking the action a in state s and then following the policy π. Intuitively, Q is the value function V with the added option of choosing the action we take in the starting state. An action *and* state based value function, if you will.

One could be led to think that one would need two sets of estimation parameters (neural net parameters) to implement Equation 2.12 with the advantage function in Equation 2.13. But through the following astute observation, it turns out we only need one set:

$$A_t = A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) = \mathbb{E}_\pi\left[r + V^\pi(s') \,|\, s, a\right] - V^\pi(s)\ , \tag{2.14}$$

where r is immediate reward. Using Equation 2.12 with the advantage derived in Equation 2.14, we acquire the REINFORCE with baseline method from Sutton and Barto (2018, p. 330).

Limitations and comparison

When considering the value iteration solution algorithm for a MDP, one should keep a few limitations/disadvantages in mind. First off, the update equations in the value iteration algorithm require known transition dynamics. Also, to be able to store the computations in-memory, problems with small discrete state-action spaces are preferred. The action space also needs to be reasonably sized, since we iterate over it to solve the $\arg\max_a$. Lastly, one needs to balance the computation horizon with computational time and memory requirements. On the other hand, policy search algorithms in general have a disadvantage compared to value estimation algorithms: policy search algorithms are less sample efficient, i.e. one usually needs more data to find a good solution.

So the primary limiting factor of the value iteration method is computational complexity/time and memory requirements, while for the policy gradient method it is availability and quality of data. Also, note that we do not need to know the transition dynamics to implement the policy gradients algorithm. This could allow us to circumvent estimations and assumptions in the pre-processing step, resulting in an approach free from model assumptions ("model-free"). As a closing note, we want to point out that getting policy gradient methods to converge to a good solution can prove a tough challenge. Since value iteration is a so called exact method (we "just" solve Eq. 2.4), we do not have to worry about convergence but rather about the quality of the assumptions used to derive the MDP framework.

Chapter 3

Approach

3.1 Method

As mentioned in Section 2.3, multiple methods for finding the optimal policy exist. Therefore, we will begin this chapter by explaining our reasoning which led us to choose value iteration and actor–critic policy gradients.

The value iteration method is a classic and well-known approach (Sutton and Barto, 2018, p. 82), and it had previously been successfully applied to RTB (Cai et al., 2017). Thus, we saw it as a natural (and fairly safe) choice for the first solution method. It is also a so called exact solution method, meaning that as long as we are able to derive and solve the equations involved the acquired solution will be adequately good. In contrast, when utilizing ACPG the algorithm can converge to unsatisfactory solutions. While this source of uncertainty is worrying, value iteration has a major drawback of its own; it is not applicable in a large-scale setting, since the memory and calculation requirements would most likely be too demanding. It was this drawback that led us to look for more advanced methods, suited for large-scale application, in the first place. Since we need to respond to bid requests within a short time frame (< 100 ms), we specifically wanted to find an algorithm where it was possible to separate and parallelize the fast action step (i.e. bidding) from the slow learning step.

Deep Q-Learning (Mnih et al., 2015) can be implemented in such a parallel manner (Mnih et al., 2016). It had also successfully been applied to RTB, see for example Wang et al. (2017) or Wu et al. (2018). So why did we choose the actor–critic policy gradient method? First off, ACPG can also be implemented in a parallel manner that allows for large-scale application, see e.g. A3C (Mnih et al., 2016). Secondly, the action space (i.e. possible bid responses) of our environment is for all intents and purposes continuous. ACPG nat-

urally handles such continuous action spaces, unlike Deep Q-Learning where continuous action spaces have to be discretized. Thus, we thought that it could be advantageous to use ACPG over Deep Q-Learning. The final reason that led us to choose ACPG over Deep Q-Learning was that we could not find any papers applying a policy gradient approach to RTB. Therefore, we thought it would be interesting to provide an (apparent) first attempt at the policy gradient approach.

3.1.1 Solving the MDP with Value Iteration

MDP formulation

Section 2.2 presented the framework of a Markov decision process. In this section, we will specify and derive the objects required to formalize the bidding problem through an MDP (see Section 2.2). As a starting point we note that each advertising campaign usually contains a great amount of auctions, wherefore we will manage computation and memory requirements by treating the real time bidding process in batches/episodes of $H = 1000$ auctions at a time. The total budget will be divided and distributed over the episodes.

Each request for a bid contains various information about the request, e.g. approximate geographical location, site, ad slot size, timestamp, and so on. We refer to this as the *feature vector* representing a bid request, denoted by $\boldsymbol{x}$. In addition to $\boldsymbol{x}$, let us define b as the *remaining budget* and t as the number of *auctions left* in the current episode. The feature vector combined with the remaining budget and number of remaining auctions, form our states $s = (t, b, \boldsymbol{x}) \in S$. Our state space S is thus every possible combination of t, b, and $\boldsymbol{x}$, which is huge. This will be handled when we derive $V(s)$.

The action space A_s of our MDP is simply the available bids (in USD) we can send in response to a bid request. The bids we are able to offer do not depend neither on $\boldsymbol{x}$ nor on t; they are only upper bounded by either the remaining budget or a pre-set maximum bid. The lower bound is always approximately zero. A problem arises here: since we can offer bids with a resolution of less than 0.001 USD, the action space is so large that it is almost continuous. We handle this by discretizing the available bids into *integer values* $[0, b]$.

After bidding for an impression, we either win or lose the auction. Thus we only have two "scenarios" for the state transition function (won/lost auction). The transition to one of them will depend only on the probability distribution of the market price for that particular impression auction. This probability distribution is unknown, and to estimate it we assume that the market price distribution only depends on the feature vector of the bid request. Let us denote this simplified market price distribution with $m(\delta, \boldsymbol{x})$, where δ is the price variable and $\boldsymbol{x}$ is the feature vector of a particular bid request.

Remember that we are dealing with second price auctions, therefore the price δ of an auction/impression is *not* the same as our bid response a: after winning an auction by responding with the highest bid, the actual price (cost) will be equal to the second highest bid. Thus, the price δ of a won auction can thus have values in the range $[0, a]$.[1] This

[1] We discretize into integer steps here too.

yields the following transition function for bidding on an auction that has a market price of δ:

$$P_a(s, s') = \begin{cases} P_a((t, b, \boldsymbol{x}), (t-1, b-\delta, \boldsymbol{x'})) = p_x(\boldsymbol{x'}) m(\delta, \boldsymbol{x}) \text{ if } a \geq \delta \ , \\ P_a((t, b, \boldsymbol{x}), (t-1, b, \boldsymbol{x'})) = p_x(\boldsymbol{x'}) \sum_{\delta=a+1}^{\infty} m(\delta, \boldsymbol{x}) \text{ if } a < \delta \ . \end{cases} \tag{3.1}$$

Since our value function will be derived as in Cai et al. (2017), an approximation $m(\delta, \boldsymbol{x}) \approx m(\delta)$ has to be made. Although it might seem so at first, one does not have to assume that the approximation holds for the whole space of possible feature vectors $\boldsymbol{X}$. We instead segment $\boldsymbol{X}$ by which advertising campaign the bid request belongs to, and assume that $m(\delta, \boldsymbol{x}) \approx m(\delta)$ holds with different distributions in each segment.

The only object left to specify is the expected immediate reward function $R_a(s, s')$. At first thought, it could be tempting to only reward the agent when it buys an impression (wins an auction) where the user clicks the ad. But with roughly one click for every 500–1000 auctions won, one would have a problem of *sparse rewards*. To combat this we will *predict the CTR* (probability of a click) for each auction, and use that as the reward received upon winning an auction. Losing an auction will yield zero reward. Denote the predicted CTR with $\theta(\boldsymbol{x})$. Just as with $m(\delta)$ in the previous paragraph, this requires additional steps/work in the data preprocessing. Again, due to the nature of the real time bidding process, we only have two general scenarios for s' and $R_a(s, s')$:

i. *Won the auction*: reward is $\theta(\boldsymbol{x})$.
   Decrease remaining auctions by one. Decrease budget left by the cost of the auction. Receive feature vector of next auction.

ii. *Lost the auction*: reward is 0.
   Decrease remaining auctions by one. Receive feature vector of next auction.

Since our goal is to simply acquire as many clicks as possible, there is no need to discount the rewards. We will hence set the discount factor γ to 1.

Predicting CTR for reward function

The reward function of our MDP formulation requires us to predict the CTR (i.e. the probability of a user click) for each impression auction. As suggested by Zhang et al. (2014), we use logistic regression (LR) to estimate the CTR. This model yields the logarithmic CTR probability as a linear combination of the input features.

The most important aspect of the logistic regression model is not necessarily the actual values of the probabilities it outputs, but rather that more valuable impression auctions (i.e. where the user clicked) consistently have a larger predicted CTR than the ones where the user did not click. Remember that we are using LR to solve the problem of sparse rewards, we are not trying to perfectly predict the CTR. Though it is probably good if the predicted CTR is fairly consistent with actual CTR values, e.g. by having the same mean.

Finding the optimal value function

In Section 2.3 we presented the value iteration algorithm for finding the optimal policy $\pi^*(s)$ of a MDP. In our real time bidding framework, a policy maps a state $s = (t, b, \boldsymbol{x})$ to an auction bid a. In the same manner, our optimal value function (see Eq. 2.3) is also a function of $s = (t, b, \boldsymbol{x})$. However, by following the derivation in Cai et al. (2017) (and utilizing $m(\delta, \boldsymbol{x}) \approx m(\delta)$ assumption) to integrate away $\boldsymbol{x}$, we acquire an approximate value function which is a function of only t and b:

$$V(t,b) \;=\; \max_{0\le a\le b}\left(\sum_{\delta=0}^{a} m(\delta)\,\theta_{\text{avg}} + \sum_{\delta=0}^{a} m(\delta)\,V(t-1,b-\delta) + \sum_{\delta=a+1}^{\infty} m(\delta)\,V(t-1,b)\right) \tag{3.2}$$

Compare Equation 3.2 with Equation 2.2. The sums over $m(\delta)$ arise from the transition probabilities in Equation 3.1 when integrating out $\boldsymbol{x}$. θ_{avg} relates to $R_a(s, s')$ and arises from the immediate reward $\theta(\boldsymbol{x})$, also after integrating out $\boldsymbol{x}$. The two sums containing $V(\cdot)$ arise from $\gamma V(s')$ in Equation 2.2: one for winning the auction, the other one for losing.

Once $V(t, b)$ has been calculated we can find the optimal policy, i.e. the optimal auction bid $a(t, b, \boldsymbol{x})$, through Equation 2.5:

$$\begin{aligned}
\pi^*(s) &= a(t,b,\boldsymbol{x}) \\
&= \arg\max_{0\le a\le b}\left(\sum_{\delta=0}^{a} m(\delta,\boldsymbol{x})(\theta(\boldsymbol{x}) + V(t-1,b-\delta)) + \sum_{\delta=a+1}^{\infty} m(\delta,\boldsymbol{x})V(t-1,b)\right) \\
&= \left[\text{Utilize } \sum_{\delta=0}^{\infty} m(\delta,\boldsymbol{x}) = 1 \text{ to change summation index and collect into one sum.}\right] \\
&= \arg\max_{0\le a\le b}\left(\sum_{\delta=0}^{a} m(\delta,\boldsymbol{x})\Big(\underbrace{\theta(\boldsymbol{x}) + V(t-1,b-\delta) - V(t-1,b)}_{\equiv \nu(\delta)}\Big)\right)
\end{aligned} \tag{3.3}$$

The keys to solving this equation are three important observations:

i. $\nu(0) = \theta(\boldsymbol{x}) > 0$

ii. $m(\delta, \boldsymbol{x}) > 0$

iii. Since $V(t-1, b)$ monotonically increases with respect to b, $V(t-1, b-\delta)$ monotonically decreases with respect to δ. This implies that $\nu(\delta)$ is monotonically decreasing with respect to δ.

So $m(\delta, \boldsymbol{x})$ is always positive, while $\nu(\delta)$ starts out positive at zero and decreases as we increase δ. This means that the sum in Equation 3.3 starts out positive, and will keep on decreasing as we increase δ. Thus, to find the a that maximizes Equation 3.3 we calculate $\nu(0), \nu(1), \ldots$ until $\nu(n)$ is negative, and then we pick $a(t, b, \boldsymbol{x}) = n - 1$ (n is an integer). Although, keep in mind that we can hit our remaining budget (or maximum bid) limit before $\nu(n) \le 0$. If that is the case, we will set $a(t, b, \boldsymbol{x})$ to our remaining budget (or maximum bid).

To summarize: we formulated our MDP framework and used it to derive the value function $V(t, b)$ and optimal bid $a(t, b, \boldsymbol{x})$ (Equations 3.2 and 3.3). To efficiently calculate $V(t, b)$

we will utilize bottom up dynamic programming (see section 2.3 for general algorithm), wherein we solve Equation 3.3 using the method described in the previous paragraph.

3.1.2 Evaluation and Baselines

We will evaluate the performance of our methods by comparing them to three different baseline approaches. The constant baseline is the bidding method most commonly used at/by Emerse, and is thus important to surpass. The other two baselines are supervised machine learning bidding methods commonly used in the industry. In addition to being commonly used in the industry, they are also often used as baselines in papers on RTB, see for example Wu et al. (2018) or Jin et al. (2018). Note that we hence refer to our method utilizing value iteration to solve the MDP as "Value Iteration MDP". We will now list and describe each of the baselines used:

Constant – Bids a pre-set constant value on each auction. When used in the industry, the DSP's client usually set the fixed bid themself. *This baseline is the main competing method we aim to beat.*

- Bid = b_0 , constant $b_0 \in \mathbb{R}^+$

Linear – This is a more advanced method which, just as our Value Iteration MDP, requires prediction of the CTR for each impression auction. It starts with a base bid b_0, which it then increases/decreases if it deems the auctioned impression is more/less likely to get a click.

- Bid = $b_0 \cdot \theta(\boldsymbol{x})/\theta_{\text{avg}}$, where $\theta(\boldsymbol{x})$ is a CTR prediction.

Max CPC – Also relies on CTR prediction. Tries to limit the cost per click by bidding in proportion to a pre-set CPC target:

- Bid = CPC $\cdot \theta(\boldsymbol{x})$, where $\theta(\boldsymbol{x})$ is a CTR prediction.

Note that Value Iteration MDP, Linear bidding, and Max CPC bidding all require CTR predictions. For a fair comparison, we will use the same linear regression based CTR prediction for all of them. Bidding performance can be gauged with a wide variety of measures, but we will mainly focus on the most common ones: number of clicks and cost per click. The number of clicks is exactly what it sounds like, just the total number of auctions won where the user clicked the ad. Cost per click (CPC) is simply the advertising cost divided by the number of clicks. So a good bidding method/algorithm will have a high number of clicks with a low CPC. Auxiliary performance measures for the full evaluation are presented in Table 1 in the appendix.

3.2 Implementation

We decided to implement our approaches in Python (version 3.6.7) due to the wide availability of packages for data preprocessing and machine learning. Also, we will implement actor–critic policy gradients with TensorFlow (1.10.0) which further supports our choice of Python.

3.2.1 Data and preprocessing

Emerse AB saves auction data in SQL databases, which allows us to fetch heaps of data with just a few lines of code. During the span of this project we have used auction data from week 4 and 5 of year 2019.[2] Week 5 is our main test data, while data from week 4 was used to estimate $m(\delta)$ and as training data to predict CTR for week 5.

The data consists of the bid request, plus some additional information, for each RTB auctions won by Emerse (in week 4 and 5). Some of the fields in the data are added retroactively, e.g. if the impression was clicked or for what duration the user watched a video ad. Since CTR prediction needs to take place as soon as a bid request arrives, those fields will not be used as input features. Note that extensive details on many of the data fields can be found in the OpenRTB specification from the Interactive Advertising Bureau (IAB).[3] We will now list and describe each input field/feature used:

User is the (scrambled) unique ID for the user who loaded the advertisement slot in question.

Exchange tells us from which ad exchange the bid request arrived.

Creative is the unique ID of the actual advertisement content.

Publisher is the publisher of the site (media) where the advertisement will be placed. Note that a single publisher can own/have multiple sites.

Site is the website where the ad will be placed. Originally contained the full page address, but to reduce unique values we replaced it with just the main site-name and top-level domain.

Platform is the platform where the ad will be placed: iPhone, iPad, Android, ChromeOS, Windows, MacOS, or Linux.

Browser specifies which browser the user is using, e.g. Safari, Chrome, Firefox, or AOL.

Position specifies the position of the ad, e.g. header, footer, sidebar, fullscreen, above or below the fold (i.e. initially visible (or not) after loading the page in question).

Device type indicates the type of device where the ad will be placed, e.g. mobile, tablet, personal computer, or smart TV.

[2] 2019-01-21 00:00:00 through 2019-02-03 23:59:59.

[3] IAB: OpenRTB API Specification Version 2.5. More on OpenRTB here.

Hour of day is exactly what it sounds like, i.e. the hour of the day the bid request arrived (indicated by 0 through 23). Extracted from exact arrival times (which has a resolution of seconds) to reduce unique values.

Click indicates if the served advertisement was clicked or not. Is retroactively added to auction data, and is (of course) not used in the CTR prediction.

As per tradition, a significant part of the time spent on our project went towards cleaning and wrangling the data. In this section, we will give an overview of the noteworthy steps:

- Load auction data with appropriate data types to minimize RAM usage. Make good use of the `categorical` data type in `pandas`.
- Segment the data by advertising campaign. For each campaign in week 5, calculate the following metadata: total impressions, total clicks, CTR, total price (sum of cost of auctions), average and median price, minimum price, maximum price.
- Estimate $m(\delta)$ for each campaign. We decided to take a purely empirical approach and calculated histograms over price data from week 4. We used 300 bins for each campaign's histogram. Also save the edges of the bins, since the edges will be different for each campaign. Note that the first bin is of a different size than the rest. Save the metadata, histograms, and bins to disk (e.g. as a CSV file).
- Split the raw auction data from week 5 into separate files and folders, one for each advertising campaign.

CTR prediction

The final step of the preprocessing is to predict the CTR, i.e. $\theta(\boldsymbol{x})$, for each auction in our test data (week 5). Our value iteration MDP method and two out of three baselines requires us to calculate $\theta(\boldsymbol{x})$. As described in Section 3.1.1, we decided to approach the the CTR prediction with logistic regression. Scikit-learn's LR model was used to perform this task.

Before we can use the LR model, we need to one-hot encode our categorical data. Note that when we calculated $m(\delta)$ and metadata, we really only used the `price` and `click` fields from the auction data. This time we will use 11 fields (see Table 3.1). In contrast to `price` and `click`, where NaN values are non-existent, the fields we are now using have plenty of NaN values. We handle this by encoding NaN values as one-hot vectors of only zeros. This way, the NaN values will not raise any errors when encoding and will not be taken into account by the LR when appearing in input.

The more unique categorical values we one-hot encode, the larger (wider) the matrix containing our encoded data will be. To determine whether or not it was feasible to include certain fields, we investigated the percentage of unique values in relation to the total number of values (see Table 3.1).

We decided to one-hot encode all of the columns in Table3.1, except for the user column. Being able to identify the user would most likely improve the accuracy of our CTR predictions. However, due to RAM restriction, we could not one-hot encode it as is. To

Table 3.1: Table over unique values in each column for the auctions from week 4, both the raw numbers and as a percent of total values in the column. We used this determine if it was viable to one-hot encode the columns.

Column name	User	Exchange	Creative	Publisher
Unique values	2868584	28	1727	7844
(percent of total)	54%	0%	0.03%	0.15%
Column name	Site	Position	Device Type	Campaign ID
Unique values	24432	8	9	112
(percent of total)	0.46%	0%	0%	0%
Column name	Platform	Hour of Day		
Unique values	11	24		
(percent of total)	0%	0%		

circumvent this, we decided to replace the unique user IDs with a scalar value which we would not need to one-hot encode: the historical CTR for the user. We calculated CTR per user from our training data (week 4 data). If the user was previously unknown, we set its CTR to the average CTR for the whole of week 4. We also adjusted outlier CTR values (which usually arose from too few data points) by squeezing user CTR that was 10 times larger or smaller than the average into the range $[10 \cdot \theta_{\text{avg}}, 0.1 \cdot \theta_{\text{avg}}]$.

So we one-hot encoded all input columns except for the calculated user CTR, which we scaled to zero mean and unit variance. The encoder and scaler used was Scikit-learn's OneHotEncoder and StandardScaler. For ease of use, these two transforms were combined into one single transform with Scikit-learn's ColumnTransformer.[4] The ColumnTransformer was fit on data from week 4. Then that same data was encoded and scaled (transformed), whereafter it was used to fit the Logistic Regression model. The final step was then to predict the CTR for auctions from week 5. We transformed the week 5 data (ColumnTransformer still only fitted to week 4 data), and predicted CTR with our LR model (also still only fitted to week 4 data). We checked the feasibility of our predictions by studying the accuracy, mean, min/max, and quantiles. Once satisfied, we split the CTR predictions by advertising campaign and saved them in separate files and folders (matching the way we split the raw auction data).

3.2.2 Main program, Value Iteration MDP, and baselines

As stated in the beginning of Section 3.2, we decided to use Python (version 3.6.7). Figure 3.1 is an overview of how the algorithms (except for ACPG) were implemented and evaluated. Everything except the main script (`main.py`) was implemented as classes. The execution begins when one runs the main script. (Various command-line options exist, which were added with the Python package `argparse`.) The `Preprocess` class handles the

[4] Documentation: www.scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing.

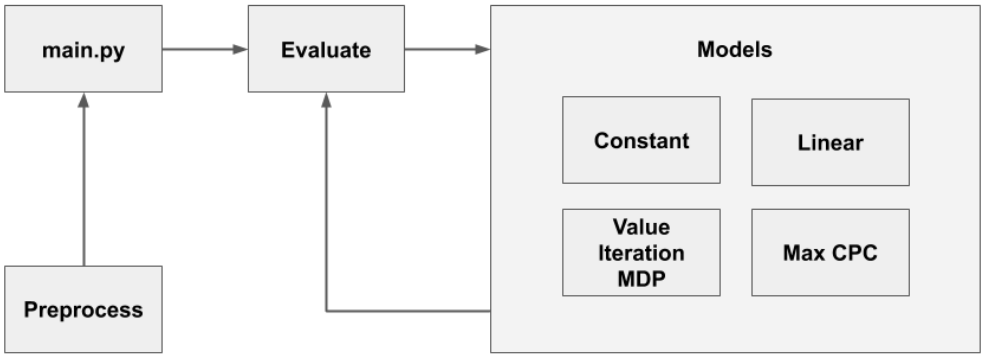


Figure 3.1: A simple implementation diagram. The actor–critic policy gradient method was implemented in a stand-alone manner. The model objects are initialized in `main.py` and are then passed to the evaluation class/object, which executes them and saves the results.

first step, namely the loading and final preparation of the input data. The `Evaluate` class is then initialized. To evaluate a model, we simply instantiate the model's corresponding class (with desired hyperparameters) and then pass it to the `Evaluate` object. `Evaluate` runs and evaluates the model, then logs (and/or displays) the result. Separating the model implementation from the evaluation allowed us to easily swap out and test different models.

3.2.3 Actor–Critic Policy Gradients

The actor–critic policy gradient method was implemented in Python using TensorFlow (version 1.10.0).[5] This solution algorithm was implemented as a stand-alone module, and was due to time constraints not integrated into the framework shown in Figure 3.1. However, it still shares much of the workflow, e.g. the one-hot encoding of the input is the same as in the LR CTR prediction. Also, to evaluate the ACPG model we used the same performance measures as before. Nevertheless, we added a few extra measures to better monitor the learning progress, e.g. mean and variance of output distribution, average bid placed in each episode, and at which step 20% budget left was reached.

As for the size and depth of the neural network layers composing the actor and critic, many variations were experimented with. We had the best (or least-worst) results using a single fully-connected layer of size 30 with a ReLU activation function for both the actor and the critic. Those layers were then connected to output layers/nodes of size one; two in the case of the actor (one for the mean, one for the standard deviation) and a single one in the critic. After the actor outputs a mean μ and a standard deviation σ, actions (bids) are drawn from the normal distribution $\mathcal{N}(\mu, \sigma^2)$.

[5] See www.tensorflow.org.

The actor–critic variation used is the A2C version (reduced to REINFORCE with baseline) presented in Equations 2.12 and 2.14 in Section 2.3. For further implementation details, see Sutton and Barto (2018, p. 330). As a closing note to this section, we would like to comment on the TensorFlow version used (1.10.0). TensorFlow 2.0 Alpha was announced at the TensorFlow Dev Summit on March 6-7 (2019), i.e. during the early stages of this thesis. Even though it would have been nice to be able to take advantage of the improvements in version 2.0, we decided to opt for version 1.10.0 to minimize the risk of running into stability and/or compatibility problems.

Chapter 4

Evaluation

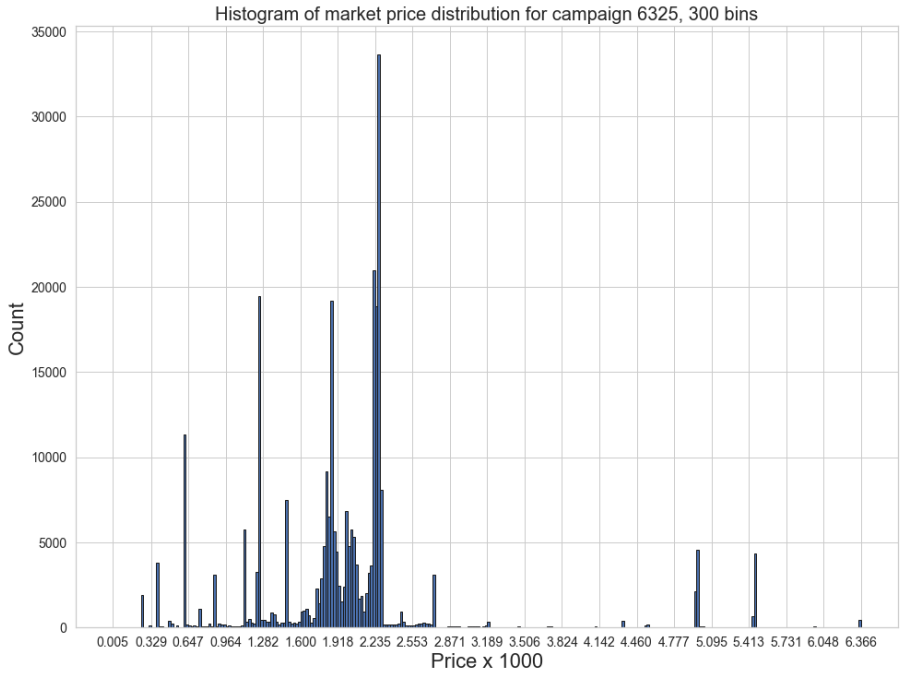
4.1 Results

In this section, we will present our results. To keep things manageable, this section will only contain visualizations of subsets of the full test results. The full test results are instead presented as a table in the appendix. We will briefly discuss the individual results, while a more thorough and overarching discussion will take place in Section 4.2.

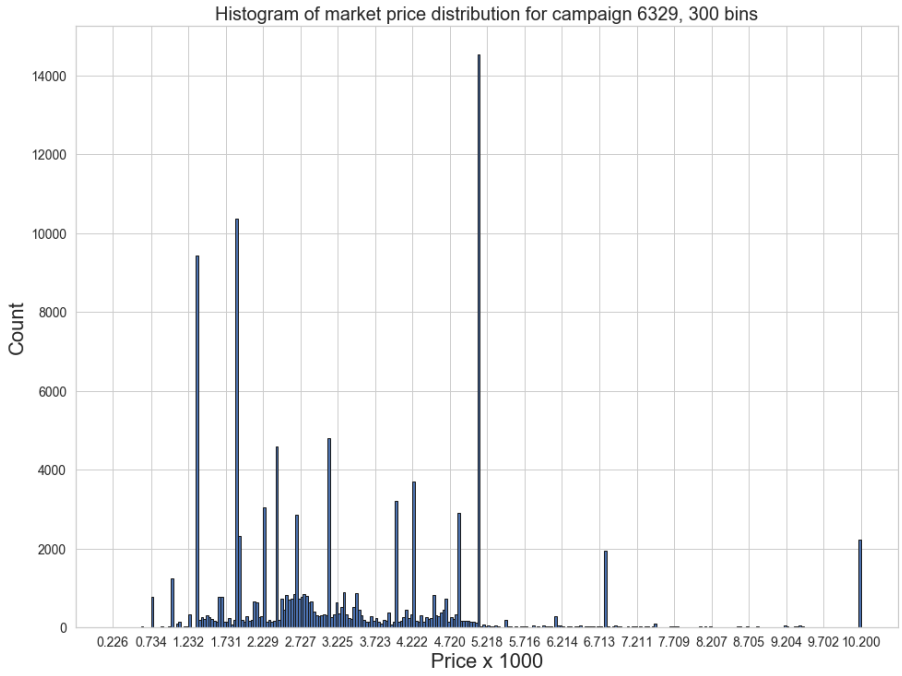
4.1.1 Market price estimation

The market price estimation $m(\delta)$ does not have to be perfect, although it is preferable if the resulting histogram at least somewhat resembles a distribution. Figure 4.1 shows two examples of a good, or at least acceptable, $m(\delta)$ estimation. Barring occasional peaks, the histograms somewhat resemble a continuous distribution. They are far from perfect though, as gaps and outliers are present.

Shown in Figure 4.2 are two examples of bad $m(\delta)$ estimations. Figure 4.2a is fragmented into a few small peaks, with a large single peak at $\sim$ 13 USD. Figure 4.2b is even worse, the estimated $m(\delta)$ almost entirely consists of a single peak at 1.2 USD. This is due to the way some advertising campaigns are set up in Emerse's DSP; a single fixed bid is set by the customer, which is adjusted downwards when possible. But like in this example, there is not always room for downward adjustment and almost all impressions in the campaign end up at the same price. In the final evaluation the campaign of the histogram in Figure 4.2b is ignored due to the bad $m(\delta)$ estimate, which arose from a lack of ample price information.

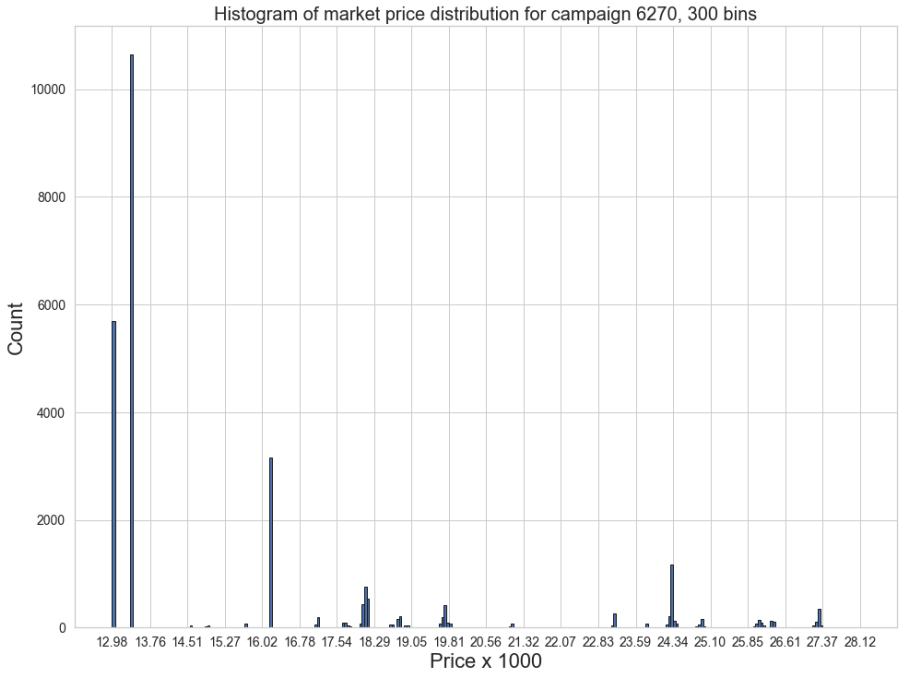


(a)

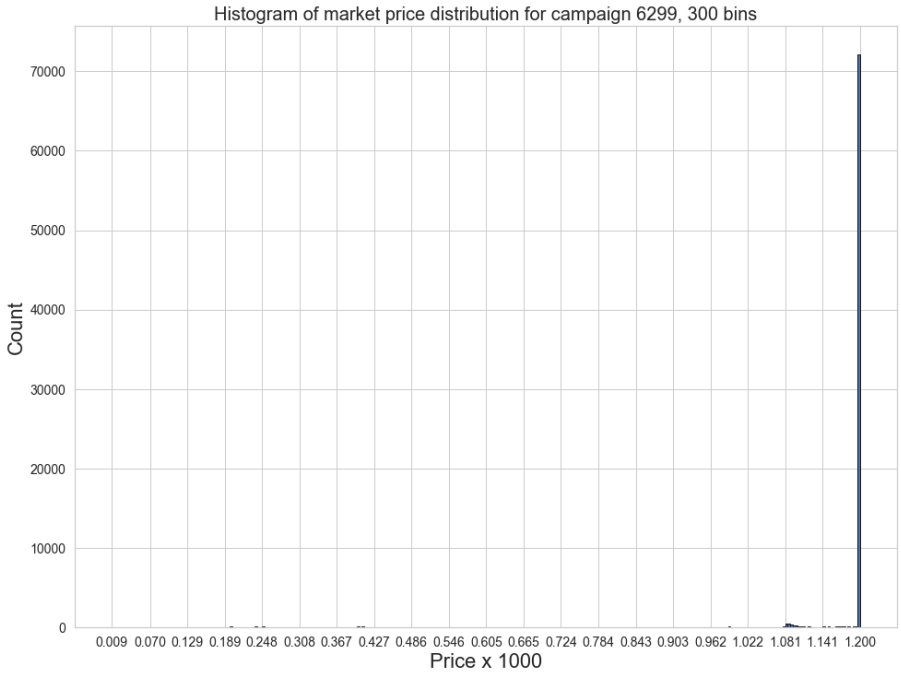


(b)

Figure 4.1: Two examples of a good $m(\delta)$ estimation. Except for a peak here and there, the histograms have some resemblance of a continuous distribution.



(a)



(b)

Figure 4.2: Two examples of a bad $m(\delta)$ estimation. The histogram is fragmented or consists of (almost) only a single peak. They both have low resemblance of a continuous distribution.

Figure 4.3 is a general overview of the $m(\delta)$ estimation for 40 out of the 41 (removed one to even out subfigures) campaigns of week 5 with more than 10000 impressions. A few suffer from the problems described above. But in the end, 31 of them were usable.

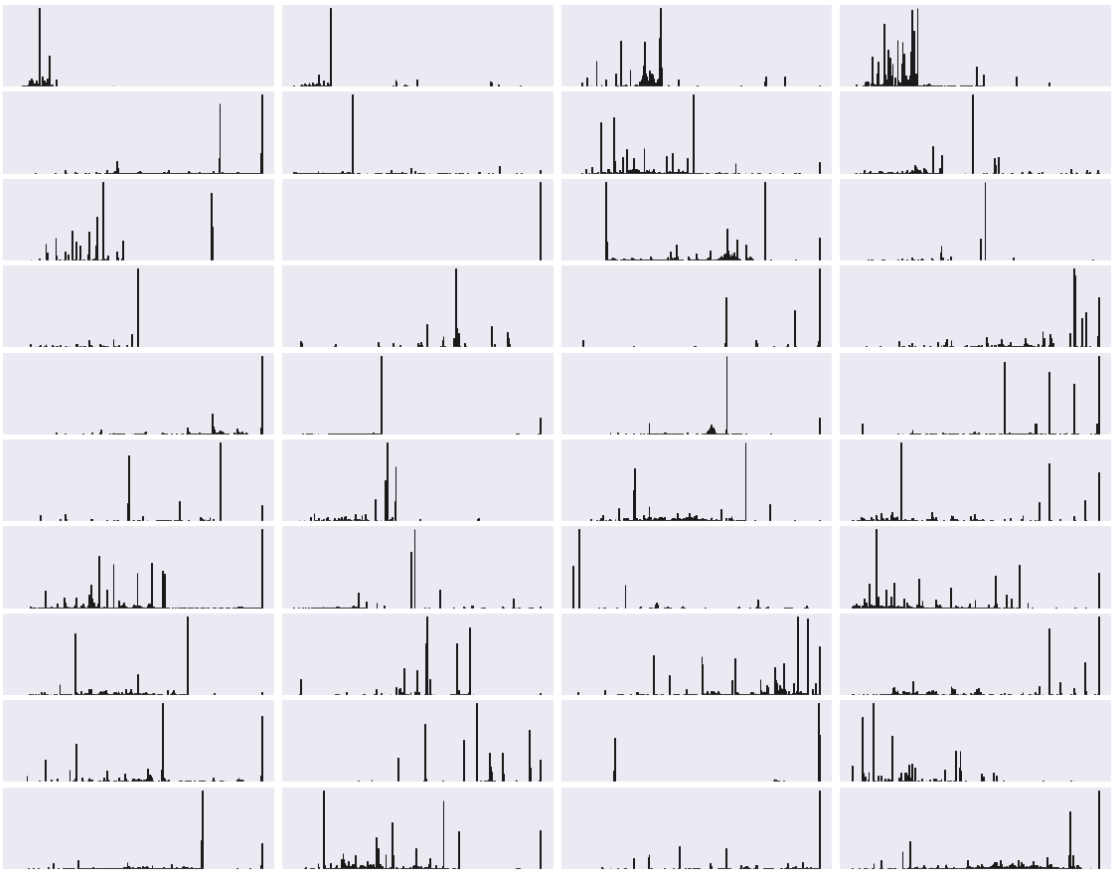


Figure 4.3: Overview of the $m(\delta)$ estimation for 40 out of the 41 (removed one to even out subfigures) campaigns of week 5 with more than 10000 impressions. Axes are the same as in Figure 4.1 and 4.2, i.e. Price×1000 on the x-axis and Counts on the y-axis.

4.1.2 CTR prediction

Figure 4.4 is an overview of the predicted CTR probabilities for auctions from campaign 6228 in week 5. The top figure shows the CTR predictions divided into two histograms; one for auctions where the user did not click the impression, and one where they did click. The bottom figure shows, for clarity, the best fitting normal distribution for each histogram. Figure 4.5 shows similar histogram-fitted normal distributions for 30 out of the 31 usable campaigns. As stated in Section 3.1.1, our goal is not to perfectly predict the actual click probability but rather to introduce some kind of relative value between the auctions. We think that this has been achieved: there is some (greater or smaller depending on campaign) distinction between auctions where the user clicked and where they did not.

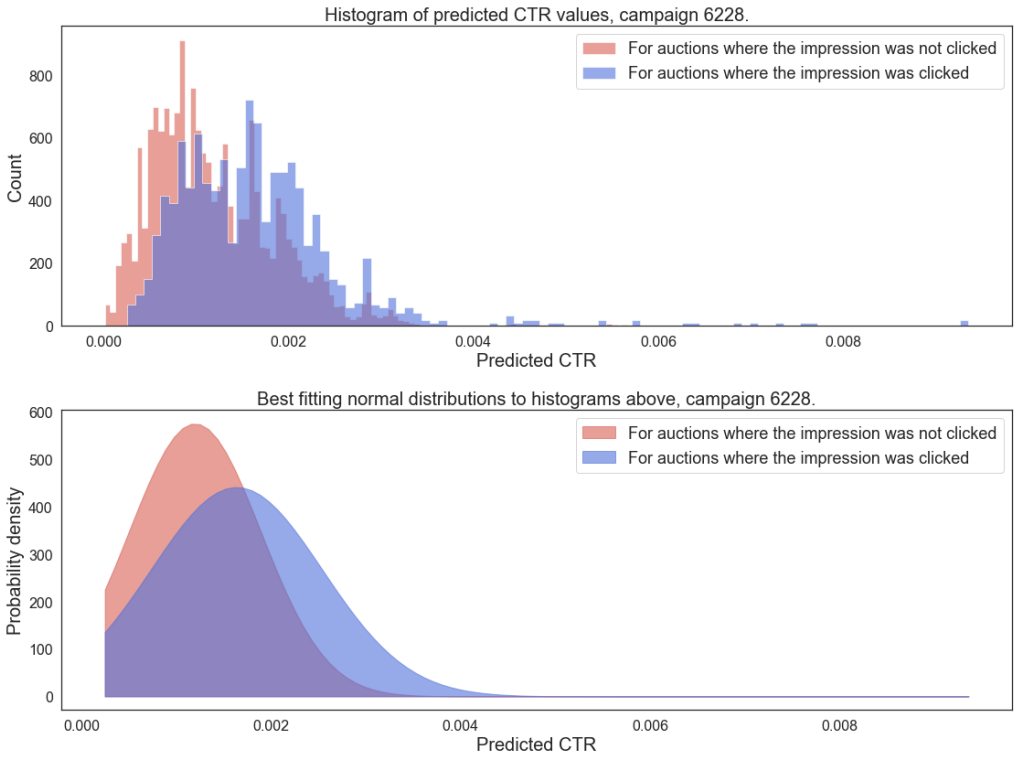


Figure 4.4: A visualization of the CTR probabilities predicted by the logistic regression for advertising campaign with ID 6228. The top figure shows the CTR predictions divided into two histograms; one for auctions where the user did not click the impression, and one where they did click. The bottom figure shows, for clarity, the best fitting normal distribution for each histogram.

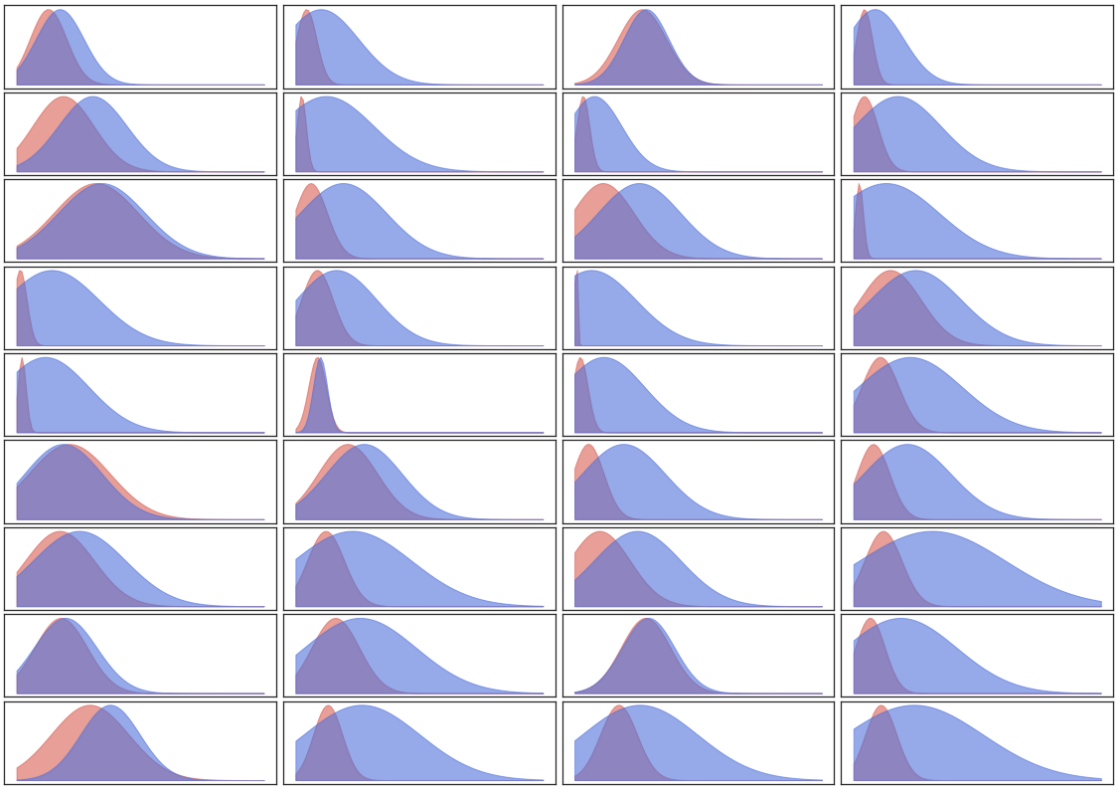


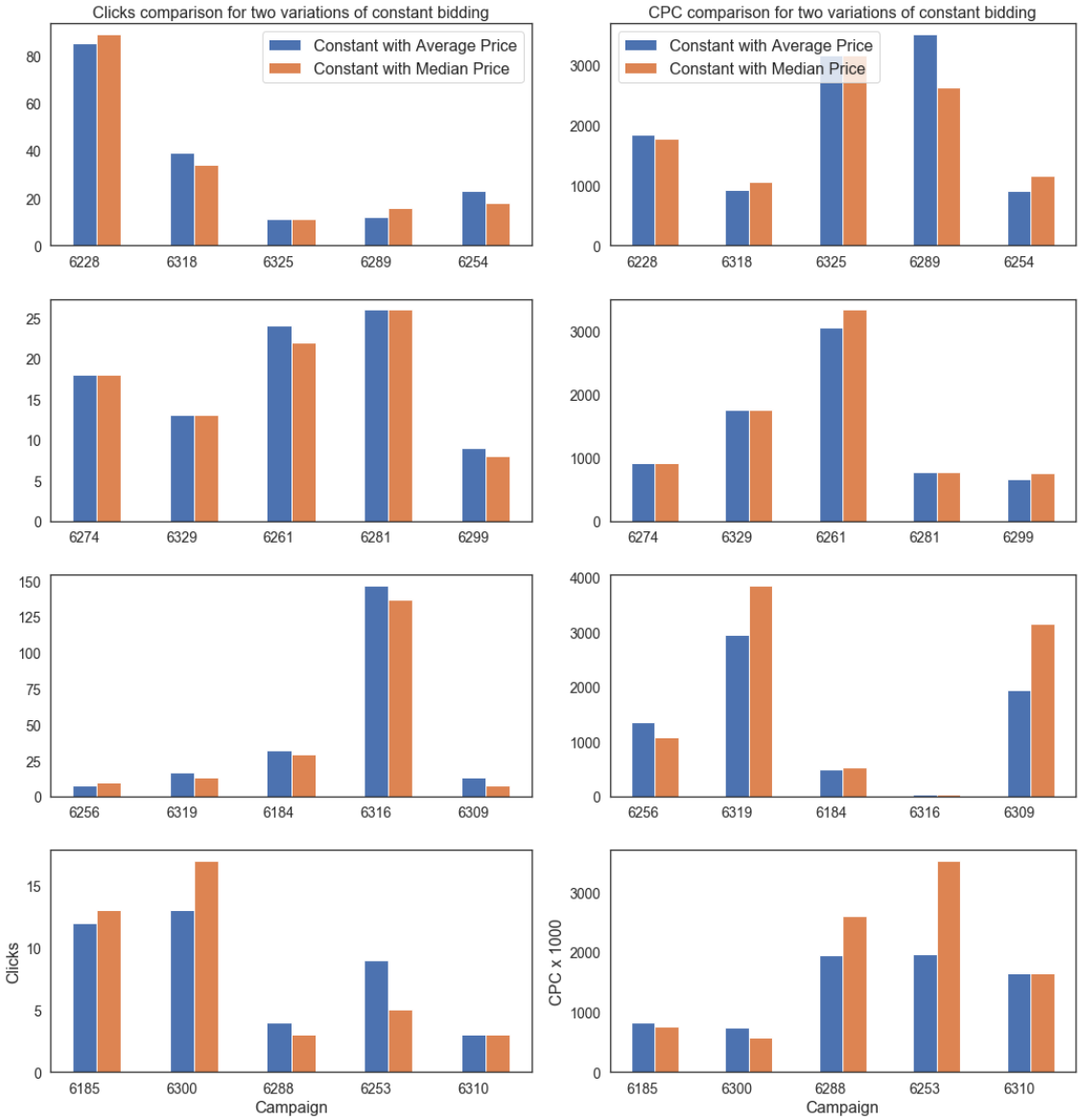
Figure 4.5: Overview of the CTR predictions for 30 out of the 31 (removed one to even out subfigures) usable campaigns of week 5 with more than 10000 impressions. Since plotting the histograms (as in Figure 4.4) did not yield a legible enough overview, we instead show normal distributions fitted to the histograms' values. Axes are the same as in Figure 4.4, i.e. predicted CTR on the x-axis and probability (density) on the y-axis. Note that we, for visual clarity, normalized the height of the probability distributions.

4.1.3 Value Iteration MDP and baselines

As stated in Section 3.1.2, we used three different baseline methods: constant bidding, linear bidding, and max CPC (cost per click) bidding. But for both constant and linear bidding, we tested two different parameter values. To avoid cluttering the final click and CPC figures with multiple variations of the same baseline, we will begin with comparing baseline alternatives and decide on only one version of each method.

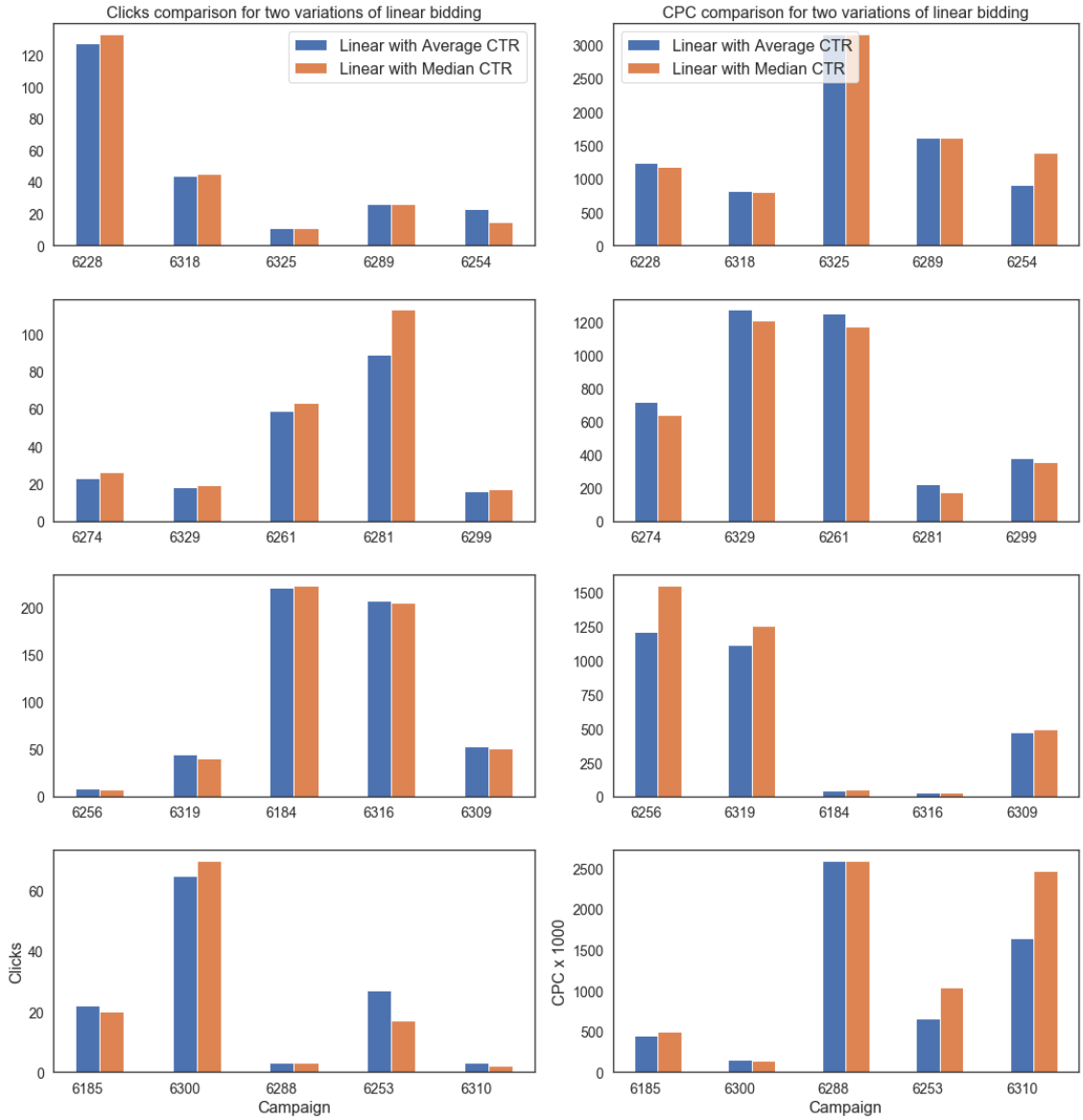
In the constant bidding baseline, we tried setting the fixed bid to either the average price or the median price for each campaign. Figure 4.6 is a comparison of the results for the top 20 campaigns sorted by number of impressions. Figure 4.6a shows the acquired clicks, and Figure 4.6b shows the cost per click. The results are similar enough that we can safely say that picking the average or median price will not significantly alter the results.

We did the same comparison for the linear bidding baselines. This time we compare the standard method, which uses θ_{avg}, to using the median CTR. As above, Figure 4.7 is a comparison of the results for the top 20 campaigns sorted by number of impressions. Figure 4.7a shows the acquired clicks, and Figure 4.7b shows the cost per click (CPC). Here too, the results are similar enough that we can safely say that picking the average or median CTR will not significantly alter the results.



(a) Clicks. **(b)** CPC.

Figure 4.6: Comparison of two variations of the constant bidding baselines: one using the average price as the fixed bid, and the other using the median price. The results are similar enough that we can safely say that picking the average or median price will not significantly alter the results. We decided to proceed with the average price version.



(a) Clicks. **(b)** CPC.

Figure 4.7: Comparison of two variations of the linear bidding baselines: one using the average CTR as the fixed bid, and the other using the median CTR. The results are similar enough that we can safely say that picking the average or median CTR will not significantly alter the results. We decided to proceed with the average CTR version.

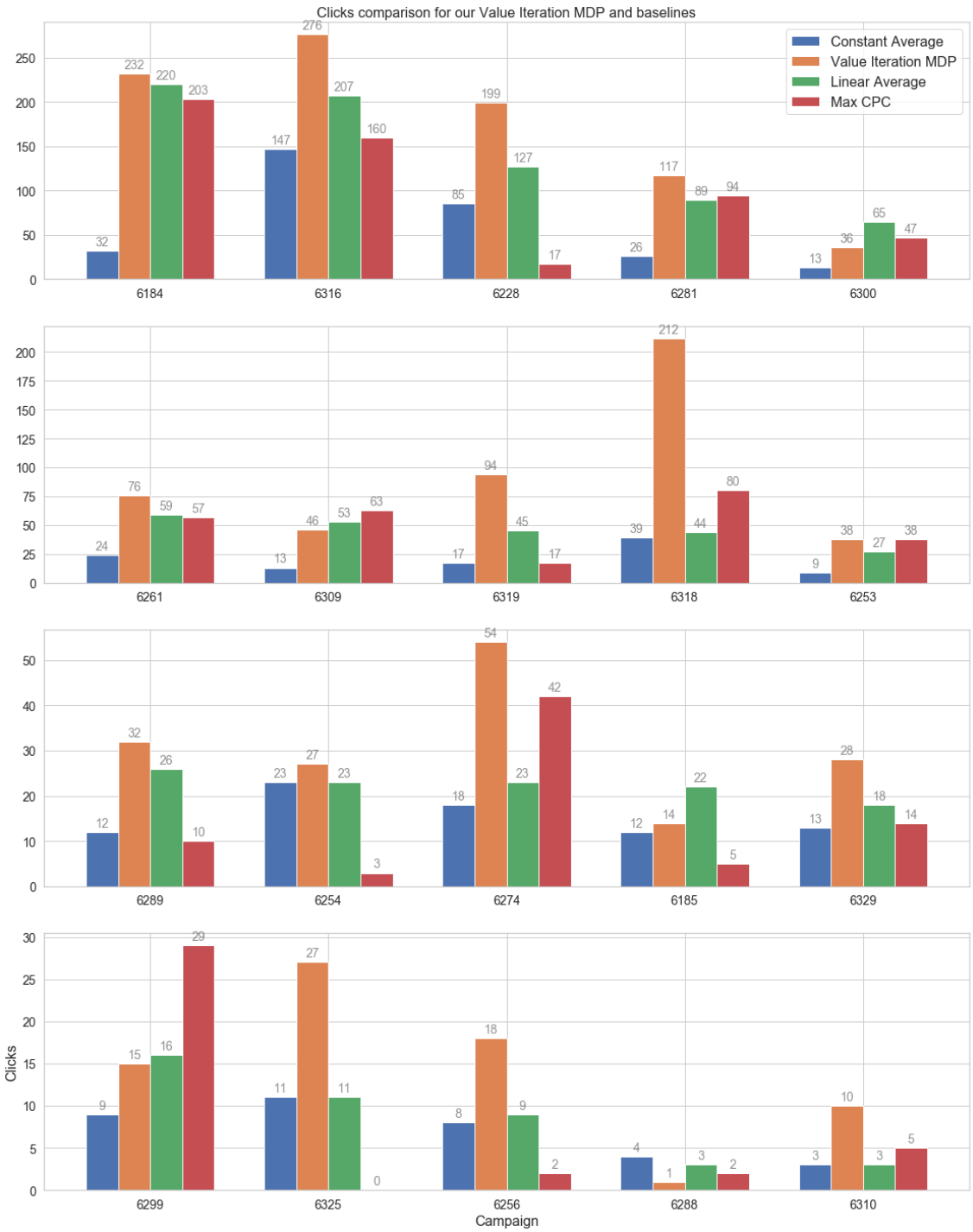


Figure 4.8: Comparison of acquired clicks performance between our Value Iteration MDP method and three baseline methods. The top 20 campaigns (by total impressions) are shown, sorted by linear bidding clicks in descending order.

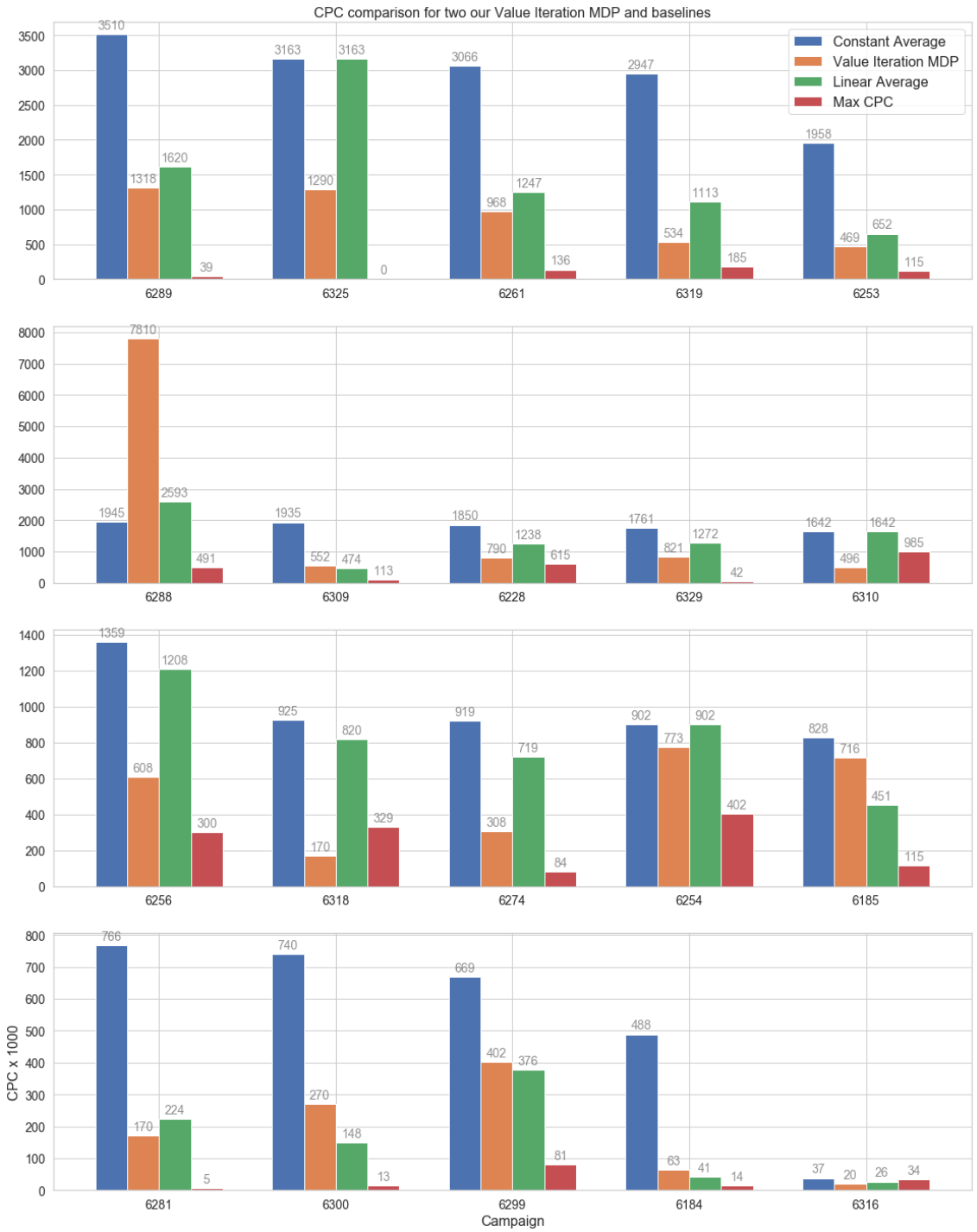
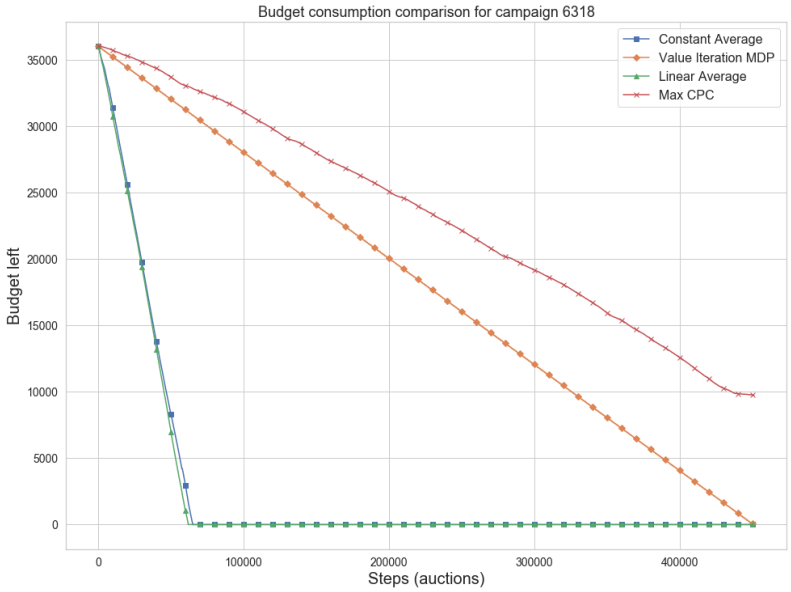
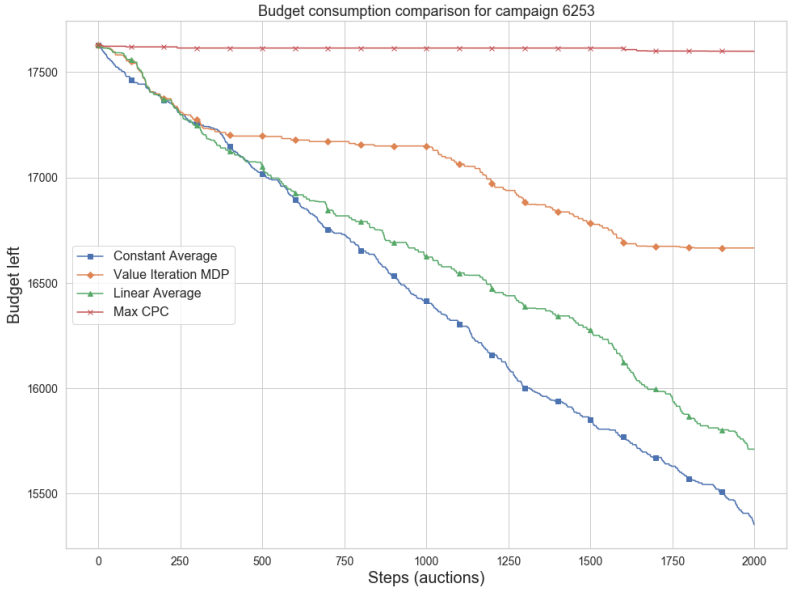


Figure 4.9: Comparison of CPC performance between our Value Iteration MDP method and three baseline methods. The top 20 campaigns (by total impressions) are shown, sorted by constant bidding CPC in descending order.



(a)



(b)

Figure 4.10: Figure 4.10a shows comparison of the budget consumption comparison for an entire campaign. Figure 4.10b is a zoomed view and shows budget consumption for the first two episodes of a campaign. Spending the budget evenly across all auctions would put us at ~16600 budget left after 2000 auctions. Value Iteration MDP (approximately) hits this mark, while constant and linear bidding consume the budget too fast. The consumption rate of Max CPC will vary wildly depending on which CPC goal we use. Here the goal results in a too low of a bid, wherefore Max CPC wins few auctions and fails in spending the entire budget. Sidenote: markers every 10000 steps in (a), while every 100 steps in (b).

After comparing the two constant and linear bidding baselines, we settled on using the average price and CTR for our final evaluation. Again, a description of the baseline methods can be found in Section 3.1.2. To reduce clutter, we chose to plot the top 20 campaigns by number of impressions. The campaigns are sorted in the manner that yielded the best visual clarity (e.g. similar height scaling). For clicks, that turned out to be by the linear average baseline in descending order. For CPC sorting by the constant bidding baseline in descending order resulted in the best visual clarity.

Figure 4.8 shows the acquired clicks for the baselines and our Value Iteration MDP method. First off, note that the constant bidding method was clearly bested by our method and the linear bidding baseline. The performance of the linear bidding model is fairly consistent, and comes close to Value Iteration MDP on several occasions. The Max CPC baseline's performance varies massively. Sometimes it performs surprisingly well (e.g. campaign 6299), and other times it lags far behind. One reason is that it is sensitive to the choice of CPC target. Also, Max CPC does not rely on relative CTR predictions like linear bidding, but rather on that the actual CTR probability predictions are correct. This increases its sensitivity to the accuracy of the CPC predictions, having less error leeway than the linear bidding model. Value Iteration MDP seems to be the most stable of the models, with click performance ranging from good to excellent.

Figure 4.9 shows cost per click (CPC) for the baselines and our Value Iteration MDP method. Constant bidding is the worst performing model here too (high CPC with few clicks). Linear bidding has higher CPC than Value Iteration and Max CPC, with a CPC almost as high as constant bidding in a few cases. Although remember that linear bidding acquires a relatively large amount of clicks, so maybe the high CPC can be forgiven. Most of the time, Max CPC manages to stay at or below its target of \$0.3 CPC (= 300 [CPC $\times$ 1000]). However it does not achieve this by smart bidding, but rather by only winning cheap (often very few) impressions. This can be seen in the budget consumption graph, and often results in low click performance. For all campaigns (except the 6288 outlier), our Value Iteration Method beats the constant bidding baseline. Also when it does not perform much better than the linear bidding, it has roughly equal amounts of clicks, but with a lower CPC. The outlier campaign 6288 has very few clicks across the board (and only one with Value Iteration MDP), which may help to explain the abnormally high CPC value.

Reasons behind similarities and differences in the performance of the methods will be discussed in the next section. As a closing note to this section, we humbly suggest that Value Iteration MDP has the best across-the-board performance: high amount of acquired clicks, low CPC, and good budget pacing.

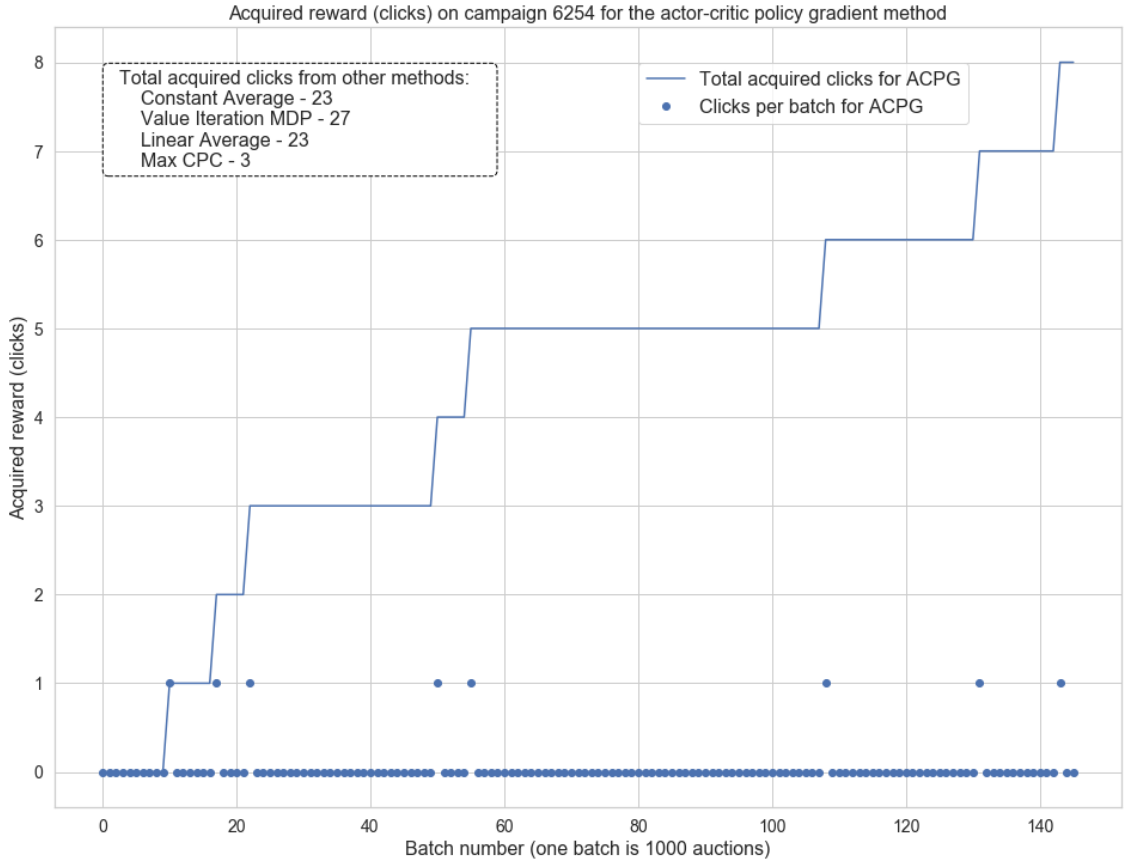


Figure 4.11: Illustrates the poor click performance of the actor–critic policy gradient approach. We believe this is mainly due to the sparse rewards. Results for campaign 6254 is presented here, but the performance is equally poor on all campaigns.

As we have mentioned multiple times, we were unable to produce a well-performing actor–critic policy gradient bidding agent. Figure 4.11 illustrates the click performance of this (unsuccessful) bidding agent. Note that click results from Value Iteration MDP and the baselines are listed in the a box in the figure. The ACPG method outperforms Max CPC here, but only since this is one campaigns where Max CPC performed the worst. The figure also illustrates what we believe is the main problem: the sparse rewards (clicks). Suggestions on how to handle this are brought up in the next section. We chose to only present one example of the ACPG agent's performance, since the performance on all the campaigns is as poor as shown in Figure 4.11.

4.2 Discussion

In this section we will discuss the data, the methods, and the results. We will address issues and suggest changes and improvements. But first, let us address a point we touched upon in the previous section: the poor performance of constant (fixed price) bidding. If the value of every auction were known, every bidder would of course try to win the most valuable ones. But the value is unknown, and we suggest that it is in trying to uncover it one can gain an advantage and build a good bidding algorithm. Value Iteration MDP, Linear Bidding, and Max CPC all have something in common: they utilize the CTR predictions $\theta(\boldsymbol{x})$. We take this as an indication that a very important factor in building a good bidding algorithm is to somehow assign a value to each impression auction/opportunity. We have done so through CTR prediction with LR, but we are sure other approaches are viable too. Linear bidding's superior performance when compared to Max CPC leads us to suggest that ranking CTR predictions in a relative manner reduces sensitivity to CTR prediction errors. At its core, the importance of ranking auction values in relation to each other stem from the fact that we have a resource limit: the advertising budget.[1] Since we can not buy impressions in a limitless manner we need to concentrate our resources where they count, i.e. on auctions for impressions with high *relative* value.

Data Quality

The first issue when working with real time bidding is one of data availability. The final price for an auction is only revealed to the one who placed the winning bid. All other bidders are left in the dark, only knowing that they lost. The competitors' bids are never revealed to any of the bidders. (Except for the second highest one in the case of winning a second price auction. Although we do not know who placed it.) Unless the bidders share data (which they generally do not), each bidder can only test their algorithms offline by reusing the auctions they have previously won. Are those auctions representative of the auction landscape as a whole? The answer is that it depends on how the auctions were won. If they were won by using only a few different bid values, they will not properly reflect the auction landscape no matter how many impressions we acquire. Instead, we argue that to gain a good representation of the auction landscape one needs to win auctions, and thus acquire data, for a wide variety of bids. Only after exploring many different bid levels we gain a dataset of auctions that can be used for effective offline historical testing. Checks for this type of non-representative data are naturally incorporated into the Value Iteration MDP method: when estimating $m(\delta)$ and studying the histograms, we uncover the described non-representative datasets.

Speed, complexity, and scalability

As shown in the introduction (Section 1), speed is of the essence when working with real time bidding; the bidders need to respond in 100 ms. In practice, due to overhead from the

[1] We would like to highlight the similarity of the bidding problem and the age-old knapsack problem.

rest of the system, our algorithms only have roughly 40 ms to input a state and output a bid. This inhibits the viability of Value Iteration MDP: calculating the value matrix takes us about a second. However once that is done, using it to serve a bid only takes a millisecond or so. The actor–critic policy gradient method has an advantage here; it can be built to be fully online, with fast prediction and slow learning happening asynchronously (i.e. in parallel). For our implementation, it takes the actor 24(±4) milliseconds to take a state as input and output a bid.

The problem we face is a common trade-off: speed versus performance. Larger and more complex methods often perform better, but are slower than simple baseline approaches. This brings us onto the issue of scalability: is it possible to apply a model in a large-scale environment and still retain speed and performance? As mentioned in Section 3.1.1, the calculation and memory requirements for the Value Iteration MDP method grow with the budget and episode length. Therefore, a workaround is needed for large-scale viability. As suggested in Sutton and Barto (2018) (chapter 9), one can approximate the value function (even non-linear ones) with a neural network. Cai et al. (2017) demonstrated large-scale viability of this value function approximation solution. As for the actor–critic policy gradient method, it yet again has the theoretical advantage. Due to its online nature and the possibility to implement in an asynchronous manner, it should have no problem scaling. For implementations, see e.g. Mnih et al. (2016). We would also like to quickly address the scalability of the baseline methods. Fixed (constant) bidding has, provided adequate supporting infrastructure, no scalability issues. Linear and max CPC bidding both utilize CTR prediction. At first though this may seem like a limiting factor, but since the CTR is predicted auction-by-auction this is not the case. Provided that the predictor does not get significantly slower as we scale up (e.g. by limiting the size of the lookback horizon for training), the model is large-scale viable as long as the prediction is fast enough for a single auction. Also, training of the predictor can take place offline and thus does not really need to be taken into account.

Improvements

Since our efforts to stop time were to no avail, we had to limit the scope of the project. In this subsection, we will suggest some improvements which we would have liked to implement.

As a starting point, we would have liked to investigate the effects of improving the CTR prediction and $m(\delta)$ estimation. We did not prioritize quantifying exactly (in a numerical manner) how much the accuracy of these two estimations affects the final result. One could most likely improve the CTR prediction by including more user-specific data in features, e.g. user (group) data from an external database. One could also try using XGBoost instead of LR, since it has had good performance in past CTR prediction contests.[2] By regularly updating the $m(\delta)$ as we gain more price information (by winning auctions), the estimation could probably be improved. Given enough auction volume, one might be able to base $m(\delta)$ on the previous day's data instead of the previous week's. The more recent data the better, provided we have enough price data to form a robust estimate.

[2] See e.g. this Kaggle contest (and this interview with the top contestant, Owen Zhang).

We would also have liked to include an additional baseline, namely "Budget Smoothed Linear Bidding" (BSLB, see Wu et al. (2018)). It is exactly what it sounds like; the linear bidding baseline with an additional (time left ratio / budget left ratio) factor. Since a more balanced budget consumption seems to positively impact the results, BSLB could have served as a tough baseline to beat.

Lastly, we would like to suggest improvements to our model-free actor–critic policy gradients approach which might help in achieving convergence and a successful result. A model-free reinforcement learning approach was successfully developed by Wu et al. (2018). They used a Deep Q-Learning method, combined with an additional neural network to combat the sparse rewards problem. We would also have liked to try a similar solution to the sparse rewards problem in our actor–critic policy gradient approach. Perhaps this could have been implemented through improvements to the critic. In addition to improving the reward function design, we would have liked to engineer more features to add to our input state. We suggest including continuously updated performance measures, e.g. the current budget consumption rate, the current CPM, and the current win rate.

Chapter 5

Conclusions

In this thesis, we modeled the real time bidding process with a Markov decision process and solved for the best bid using a value iteration algorithm. An actor–critic policy gradient solution was also implemented, it was however unable to converge. The value iteration MDP approach and three other baseline methods were evaluated on real-world data. Our value iteration MDP approach proved to be (compared to the baselines) successful in increasing clicks, lowering CPC, and smoothing budget consumption. However for it to be viable in a real-world large-scale settings, one would need to increase its speed, e.g. through function approximation of the state-value matrix.

While carrying out our literature study, we were not able to find any papers applying a policy gradient based approach to RTB. Thus, we aimed to provide an (apparent) first exploration of the viability of a policy gradient based real time bidding algorithm. While our actor–critic policy gradient approach was unable to converge to a satisfactory solution, we still hope that our efforts will facilitate the success of any further attempts. Should such an attempt be made, we suggest that one considers the following lesson learned: rewarding the algorithm only when a click is acquired will yield too sparse rewards, which is likely the main issue preventing convergence to an adequate solution. To further increase likelihood of success, we also suggest utilizing feature engineering to including more information in the state than just the data from the bid request. Barring these problems, we believe the policy gradient approach to be a good fit for RTB, since it naturally handles the almost continuous action (bidding) space and is applicable to a large-scale environment.

Chapter 6

Bibliography

Bellman, R. (1957). A Markovian Decision Process. *Indiana Univ. Math. J.*, 6(4):679–684.

Cai, H., Ren, K., Zhang, W., Malialis, K., Wang, J., Yu, Y., and Guo, D. (2017). Real-time bidding by reinforcement learning in display advertising. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 661–670. ACM.

Fischer, T. G. (2018). Reinforcement learning in financial markets - a survey. FAU Discussion Papers in Economics 12/2018, Erlangen.

Huang, S. H., Zambelli, M., Kay, J., Martins, M. F., Tassa, Y., Pilarski, P. M., and Hadsell, R. (2019). Learning gentle object manipulation with curiosity-driven deep reinforcement learning. *arXiv preprint arXiv:1903.08542*.

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., and Hutter, M. (2019). Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872.

Jin, J., Song, C., Li, H., Gai, K., Wang, J., and Zhang, W. (2018). Real-time bidding with multi-agent reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2193–2201. ACM.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

Puterman, M. (1994). *Markov decision processes : discrete stochastic dynamic programming*. Wiley-Interscience, Hoboken, N.J. Great Britain. ISBN 0471619779.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: an Introduction*. The MIT Press, Cambridge, MA. Second edition. ISBN 9780262039246.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063.

Wang, Y., Liu, J., Liu, Y., Hao, J., He, Y., Hu, J., Yan, W. P., and Li, M. (2017). Ladder: A human-level bidding agent for large-scale real-time online auctions. *arXiv preprint arXiv:1708.05565*.

Wu, D., Chen, X., Yang, X., Wang, H., Tan, Q., Zhang, X., Xu, J., and Gai, K. (2018). Budget constrained bidding by model-free reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1443–1451. ACM.

Xiong, Z., Liu, X.-Y., Zhong, S., Walid, A., et al. (2018). Practical deep reinforcement learning approach for stock trading. *arXiv preprint arXiv:1811.07522*.

Yu, P., Lee, J. S., Kulyatin, I., Shi, Z., and Dasgupta, S. (2019). Model-based deep reinforcement learning for dynamic portfolio optimization. *arXiv preprint arXiv:1901.08740*.

Zhang, W., Yuan, S., Wang, J., and Shen, X. (2014). Real-time bidding benchmarking with ipinyou dataset. *arXiv preprint arXiv:1407.7073*.

Appendices

Table 1: Full test results from week 5. As described in Section 3.2, we only treated campaigns with more than 10000 impressions. This leaves us with 41 campaigns from week 5, where 10 of them were ignored due to non-representative price data (see Section 4.2).

Campaign ID	Algorithm	Auctions	Bought	Clicks	Cost	CPM	CPC	Win Rate	Budget
6228	Const. Average	1131312	88917	85	157.2524	1.7685	1.8500	0.0786	157.2524
6228	Value Iter. MDP	1131312	138606	199	157.3480	1.1352	0.7907	0.1225	0.1390
6228	Max CPC	1131312	9421	17	10.4701	1.1114	0.6159	0.0083	157.2524
6228	Linear Average	1131312	72625	127	157.2524	2.1653	1.2382	0.0642	157.2524
6318	Const. Average	451051	48297	39	36.0841	0.7471	0.9252	0.1071	36.0841
6318	Value Iter. MDP	451051	355853	212	36.1590	0.1016	0.1706	0.7889	0.0800
6318	Max CPC	451051	59637	80	26.3410	0.4417	0.3293	0.1322	36.0841
6318	Linear Average	451051	40974	44	36.0841	0.8807	0.8201	0.0908	36.0841
6325	Const. Average	280671	24747	11	34.8032	1.4064	3.1639	0.0882	34.8032
6325	Value Iter. MDP	280671	46202	27	34.8440	0.7542	1.2905	0.1646	0.1240
6325	Max CPC	280671	700	0	0.5698	0.8140	0.0000	0.0025	34.8032
6325	Linear Average	280671	18400	11	34.8032	1.8915	3.1639	0.0656	34.8032
6289	Const. Average	236696	22252	12	42.1319	1.8934	3.5110	0.0940	42.1319
6289	Value Iter. MDP	236696	27371	32	42.1860	1.5413	1.3183	0.1156	0.1780
6289	Max CPC	236696	754	10	0.3966	0.5260	0.0397	0.0032	42.1319
6289	Linear Average	236696	19720	26	42.1319	2.1365	1.6205	0.0833	42.1319
6254	Const. Average	146248	14207	23	20.7672	1.4618	0.9029	0.0971	20.7672
6254	Value Iter. MDP	146248	18785	27	20.8740	1.1112	0.7731	0.1284	0.1420
6254	Max CPC	146248	1339	3	1.2067	0.9012	0.4022	0.0092	20.7672
6254	Linear Average	146248	10825	23	20.7672	1.9184	0.9029	0.0740	20.7672
6274	Const. Average	131359	13836	18	16.5512	1.1962	0.9195	0.1053	16.5512
6274	Value Iter. MDP	131359	33539	54	16.6320	0.4959	0.3080	0.2553	0.1260
6274	Max CPC	131359	8583	42	3.5533	0.4140	0.0846	0.0653	16.5512
6274	Linear Average	131359	13997	23	16.5512	1.1825	0.7196	0.1066	16.5512
6329	Const. Average	107540	9347	13	22.9058	2.4506	1.7620	0.0869	22.9060
6329	Value Iter. MDP	107540	11406	28	23.0040	2.0168	0.8216	0.1061	0.2130
6329	Max CPC	107540	162	14	0.5983	3.6934	0.0427	0.0015	22.9060
6329	Linear Average	107540	8388	18	22.9060	2.7308	1.2726	0.0780	22.9060
6261	Const. Average	104100	10546	24	73.5987	6.9788	3.0666	0.1013	73.5987
6261	Value Iter. MDP	104100	8197	76	73.6250	8.9819	0.9688	0.0787	0.7070
6261	Max CPC	104100	1206	57	7.7931	6.4619	0.1367	0.0116	73.5987
6261	Linear Average	104100	4767	59	73.5987	15.4392	1.2474	0.0458	73.5987
6281	Const. Average	100700	9489	26	19.9386	2.1012	0.7669	0.0942	19.9386
6281	Value Iter. MDP	100700	13023	117	19.9400	1.5311	0.1704	0.1293	0.1980
6281	Max CPC	100700	589	94	0.5453	0.9259	0.0058	0.0058	19.9386
6281	Linear Average	100700	10008	89	19.9386	1.9923	0.2240	0.0994	19.9386
6299	Const. Average	84823	7535	9	6.0224	0.7993	0.6692	0.0888	6.0224
6299	Value Iter. MDP	84823	13152	15	6.0350	0.4589	0.4023	0.1551	0.0710
6299	Max CPC	84823	2606	29	2.3519	0.9025	0.0811	0.0307	6.0224
6299	Linear Average	84823	5615	16	6.0224	1.0726	0.3764	0.0662	6.0224
6256	Const. Average	75555	9062	8	10.8799	1.2006	1.3600	0.1199	10.8799
6256	Value Iter. MDP	75555	21525	18	10.9440	0.5084	0.6080	0.2849	0.1440
6256	Max CPC	75555	1185	2	0.6006	0.5068	0.3003	0.0157	10.8799
6256	Linear Average	75555	8596	9	10.8799	1.2657	1.2089	0.1138	10.8799
6319	Const. Average	72723	6829	17	50.1061	7.3373	2.9474	0.0939	50.1061
6319	Value Iter. MDP	72723	5356	94	50.2290	9.3781	0.5344	0.0736	0.6890
6319	Max CPC	72723	617	17	3.1588	5.1197	0.1858	0.0085	50.1061
6319	Linear Average	72723	4353	45	50.1061	11.5107	1.1135	0.0599	50.1061
6184	Const. Average	70714	7635	32	15.6278	2.0469	0.4884	0.1080	15.6278
6184	Value Iter. MDP	70714	14411	232	14.7580	1.0241	0.0636	0.2038	0.2210
6184	Max CPC	70714	2128	203	2.9593	1.3906	0.0146	0.0301	15.6278
6184	Linear Average	70714	7541	220	9.1955	1.2194	0.0418	0.1066	15.6278
6316	Const. Average	70285	5606	147	5.5525	0.9905	0.0378	0.0798	5.5525
6316	Value Iter. MDP	70285	14794	276	5.6090	0.3791	0.0203	0.2105	0.0790
6316	Max CPC	70285	4508	160	5.5525	1.2317	0.0347	0.0641	5.5525
6316	Linear Average	70285	4667	207	5.5525	1.1897	0.0268	0.0664	5.5525
6309	Const. Average	63207	5471	13	25.1564	4.5981	1.9351	0.0866	25.1564
6309	Value Iter. MDP	63207	6065	46	25.4160	4.1906	0.5525	0.0960	0.3980
6309	Max CPC	63207	1747	63	7.1470	4.0910	0.1134	0.0276	25.1564
6309	Linear Average	63207	5032	53	25.1564	4.9993	0.4746	0.0796	25.1564

Campaign ID	Algorithm	Auctions	Bought	Clicks	Cost	CPM	CPC	Win Rate	Budget
6185	Const. Average	57449	5062	12	9.9386	1.9634	0.8282	0.0881	9.9387
6185	Value Iter. MDP	57449	5779	14	10.0340	1.7363	0.7167	0.1006	0.1730
6185	Max CPC	57449	356	5	0.5786	1.6252	0.1157	0.0062	9.9387
6185	Linear Average	57449	4679	22	9.9387	2.1241	0.4518	0.0814	9.9387
6300	Const. Average	56296	5052	13	9.6266	1.9055	0.7405	0.0897	9.6266
6300	Value Iter. MDP	56296	8768	36	9.7470	1.1117	0.2707	0.1557	0.1710
6300	Max CPC	56296	887	47	0.6555	0.7390	0.0139	0.0158	9.6266
6300	Linear Average	56296	5822	65	9.6266	1.6535	0.1481	0.1034	9.6266
6288	Const. Average	54793	4848	4	7.7806	1.6049	1.9452	0.0885	7.7806
6288	Value Iter. MDP	54793	6617	1	7.8100	1.1803	7.8100	0.1208	0.1420
6288	Max CPC	54793	877	2	0.9822	1.1199	0.4911	0.0160	7.7806
6288	Linear Average	54793	4283	3	7.7806	1.8166	2.5935	0.0782	7.7806
6253	Const. Average	36424	3321	9	17.6292	5.3084	1.9588	0.0912	17.6292
6253	Value Iter. MDP	36424	3626	38	17.8440	4.9211	0.4696	0.0995	0.4840
6253	Max CPC	36424	1062	38	4.3712	4.1160	0.1150	0.0292	17.6292
6253	Linear Average	36424	2366	27	17.6292	7.4511	0.6529	0.0650	17.6292
6310	Const. Average	33760	5295	3	4.9290	0.9309	1.6430	0.1568	4.9290
6310	Value Iter. MDP	33760	11962	10	4.9640	0.4150	0.4964	0.3543	0.1460
6310	Max CPC	33760	6080	5	4.9290	0.8107	0.9858	0.1801	4.9290
6310	Linear Average	33760	3646	3	4.9290	1.3519	1.6430	0.1080	4.9290
6304	Const. Average	30870	3563	6	24.0477	6.7493	4.0080	0.1154	24.0477
6304	Value Iter. MDP	30870	2115	33	24.1230	11.4057	0.7310	0.0685	0.7790
6304	Max CPC	30870	324	15	2.2274	6.8747	0.1485	0.0105	24.0477
6304	Linear Average	30870	1794	31	24.0477	13.4045	0.7757	0.0581	24.0477
6270	Const. Average	26919	2023	7	26.7021	13.1993	3.8146	0.0752	26.7037
6270	Value Iter. MDP	26919	1414	27	26.6920	18.8769	0.9886	0.0525	0.9920
6270	Max CPC	26919	94	14	1.9936	21.2081	0.1424	0.0035	26.7037
6270	Linear Average	26919	1561	24	26.6927	17.0998	1.1122	0.0580	26.7037
6314	Const. Average	25917	3732	2	24.7767	6.6390	12.3883	0.1440	24.7767
6314	Value Iter. MDP	25917	3235	3	24.8560	7.6835	8.2853	0.1248	0.9560
6314	Max CPC	25917	201	1	0.0445	0.2216	0.0445	0.0078	24.7767
6314	Linear Average	25917	1756	3	24.7766	14.1097	8.2589	0.0678	24.7767
6290	Const. Average	21055	1498	45	1.7897	1.1947	0.0398	0.0711	1.7897
6290	Value Iter. MDP	21055	3638	115	1.8290	0.5027	0.0159	0.1728	0.0850
6290	Max CPC	21055	1440	50	1.7897	1.2428	0.0358	0.0684	1.7897
6290	Linear Average	21055	1628	120	1.7897	1.0993	0.0149	0.0773	1.7897
6293	Const. Average	21003	1840	1	1.7642	0.9588	1.7642	0.0876	1.7643
6293	Value Iter. MDP	21003	7115	10	1.7660	0.2482	0.1766	0.3388	0.0840
6293	Max CPC	21003	811	8	0.6742	0.8313	0.0843	0.0386	1.7643
6293	Linear Average	21003	1700	4	1.7643	1.0378	0.4411	0.0809	1.7643
6311	Const. Average	20433	2259	1	3.7392	1.6553	3.7392	0.1106	3.7392
6311	Value Iter. MDP	20433	3848	7	3.8430	0.9987	0.5490	0.1883	0.1830
6311	Max CPC	20433	2408	6	2.7384	1.1372	0.4564	0.1178	3.7392
6311	Linear Average	20433	1194	0	3.7392	3.1317	0.0000	0.0584	3.7392
5801	Const. Average	20133	1881	19	3.3823	1.7981	0.1780	0.0934	3.3823
5801	Value Iter. MDP	20133	4064	45	3.4930	0.8595	0.0776	0.2019	0.1680
5801	Max CPC	20133	232	67	0.3805	1.6402	0.0057	0.0115	3.3823
5801	Linear Average	20133	1082	68	1.0373	0.9587	0.0153	0.0537	3.3823
6282	Const. Average	18080	1377	3	2.0611	1.4968	0.6870	0.0762	2.0611
6282	Value Iter. MDP	18080	2052	2	2.1610	1.0531	1.0805	0.1135	0.1140
6282	Max CPC	18080	91	1	0.1021	1.1221	0.1021	0.0050	2.0611
6282	Linear Average	18080	1332	3	2.0610	1.5473	0.6870	0.0737	2.0611
6224	Const. Average	15236	1938	5	6.2467	3.2233	1.2493	0.1272	6.2468
6224	Value Iter. MDP	15236	2184	7	6.5550	3.0014	0.9364	0.1433	0.4100
6224	Max CPC	15236	1040	9	1.2968	1.2470	0.1441	0.0683	6.2468
6224	Linear Average	15236	1726	8	6.2466	3.6191	0.7808	0.1133	6.2468
6327	Const. Average	13551	1197	3	1.6939	1.4151	0.5646	0.0883	1.6939
6327	Value Iter. MDP	13551	4298	6	1.7500	0.4072	0.2917	0.3172	0.1250
6327	Max CPC	13551	237	6	0.2561	1.0804	0.0427	0.0175	1.6939
6327	Linear Average	13551	1551	3	1.6939	1.0921	0.5646	0.1145	1.6939
4751	Const. Average	10023	951	2	1.8842	1.9813	0.9421	0.0949	1.8843
4751	Value Iter. MDP	10023	1192	4	1.9450	1.6317	0.4863	0.1189	0.1880
4751	Max CPC	10023	7	0	0.0022	0.3100	0.0000	0.0007	1.8843
4751	Linear Average	10023	1210	3	1.8843	1.5573	0.6281	0.1207	1.8843

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-06-11

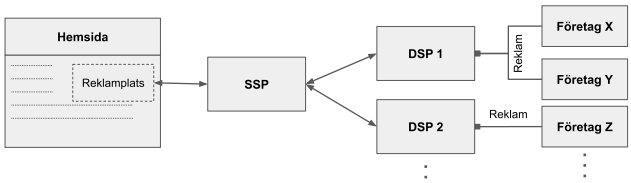
EXAMENSARBETE Reinforcement Learning for Real Time Bidding
STUDENT Erik Smith
HANDLEDARE Pierre Nugues (LTH), Rasmus Larsson (Emerse AB), Carl-Johan Grund (Emerse AB)
EXAMINATOR Elin Anna Topp (LTH)

Reinforcement Learning för Real Time Bidding

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Smith**

Majoriteten av den internet-reklam du ser idag har blivit utvald genom ett blixtsnabbt auktionsförlopp: så fort du börjar ladda en reklamplats, lägger företag som vill visa reklam för dig bud på den. Högstbjudande får visa sin reklam i platsen, men vilket bud är bäst att lägga?

I själva verket är det sällan företagen själva som hanterar reklambudgivningen, då de saknar nödvändig teknik-infrastruktur. Av samma anledning är det inte heller de som har reklamplatserna (t.ex. tidningar eller bloggar) som genomför själva auktionen. Istället hanteras de tekniskt komplicerade stegen av mellanhänder. En så kallad supply side plattform (SSP) tar hand om ut-auktioneringen för de som vill visa reklam på sin sida. I samma anda läggs budgivningsansvaret över på så kallade demand side plattformar (DSPs). En DSP representerar och sköter budgivning åt de som har själva reklam-materialet, t.ex. företag eller intresseorganisationer. Företaget vi har gjort detta examensarbete hos är/driver en sådan DSP.



Reklamkampanjer har en begränsad budget och löper över en begränsad tidsperiod. Man har långt ifrån råd att köpa alla reklamplatser som auktioneras ut, så för att en kampanj ska påverka så mycket som möjligt måste budgeten spenderas strategiskt. Hur bra är reklamplats X, och är det är värt att lägga ett högt bud på den eller ska vi vänta och hoppas på att en ännu bättre dyker upp? I vårt examensarbete skapade vi, genom en typ av maskininlärning som kallas reinforcement learning, en budgivnings-algoritm som presterar bättre än den som används i nuläget.

Genom att lägga bud på ett smartare sätt, så får företagen mer valuta för sina reklam-pengar. Det kan även eventuellt leda till att internetanvändare får se mer relevant reklam. Till exempel skulle algoritmen möjligtvis kunna identifiera att vissa sportintresserade personer aldrig klickar på spel-reklam, och därigenom sluta visa (köpa) onödig reklam för de personerna.

Vi skapade två budgivningsalgoritmer. Den ena, en så kallad Markov decision process löst via value iteration, visade lovande resultat. Med samma budget som andra jämförelse-algoritmer lyckades den köpa fler reklamplatser som blev klickade på, medan den samtidigt spenderade mindre budget för att "få ett klick". Den lyckades även spendera budgeten i en jämnare takt än de andra jämföreselse-algoritmerna.