Deep Reinforcement Learning in Real-Time Bidding

Author:
Oskar Stigland
Bachelor Thesis
Fall 2018

Supervisors:
Alexandros Sopasakis
Morten Arngren
Vlad Sandulescu

Lund University

Centre for Mathematical Sciences
Numerical Analysis

Abstract

This segment should describe the contents of the thesis, how the experiments etc have been carried out, what type of methods have been used and a short summary of the results of the project. Include `GitHub` link to repository.

Acknowledgements

I want to thank...

Contents

Chapter 1

Introduction

Today, online display advertisement is increasingly sold and bought through a process known as real-time bidding (RTB). In the last decade, spending on RTB has increased dramatically. The supposed reason for this impressive growth is the overall efficiency benefits from RTB (Yuan et al., 2014). By utilizing information from cookies, advertisers can target specific users who might be more susceptible to a given advertisement campaign. More specifically, an advertiser can target *only* these desirable users. In this sense, traditional display advertisement methods, e.g. billboards and newspaper ads, are extremely inefficient since the advertiser is paying the same amount for every impression, regardless of the effect. Even more recent innovations such as buying keywords or time-slots on websites appear inefficient compared to RTB.

Whenever a user logs onto a website with available ad slots, the owner of the domain (often referred to as a *publisher*) sends out a request to a so-called *ad exchange* (AdX). The AdX then sends out bid requests to a number of so-called *demand-side platforms* (DSPs) and holds an auction in which the DSPs submit bids to win the impression. The DSP that submits the highest price wins the ad slot and pays the second-highest price. A simple summary of the process is provided in a figure below.

So, why are we talking about DSPs and where are the actual advertisers? The process of auctioning out an ad slot and subsequently showing it to a user takes less than 100 milliseconds. To put this in perspective, blinking your eye takes about 300 to 400 milliseconds. Hence, the ad-buying procedure is entirely algorithmic. Bids submitted by DSPs have to be computed instantly when a bid request is received and a typical DSP handles billions of auctions on a daily basis.

There are many areas of research in the RTB ecosystem, ranging from bidding

strategy to auction design. Yuan et al. (2014) provide a good overview of different research problems. This thesis will focus on the problem of creating a bidding strategy for an ad campaign and, more specifically, on employing an algorithmic bidding agent which incorporates reinforcement-learning techniques to bid intelligently given campaign-relevant parameters.

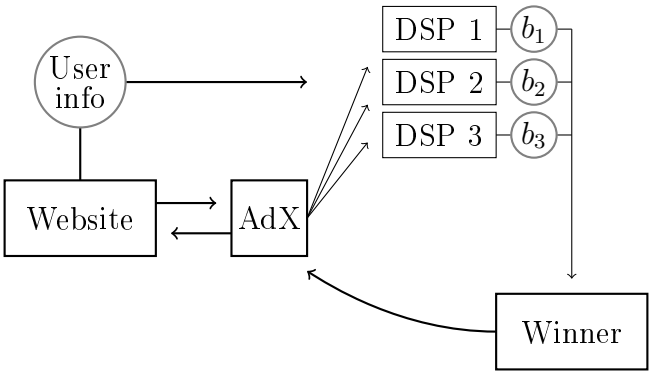


Figure 1.1: A simplified RTB ecosystem

Reinforcement learning has been introduced into RTB in the last two to three years, with two of the most prominent works being Du et al. (2017) and Cai et al. (2017). This thesis will focus on the approach used by Wu et al. (2018), where a bidding agent is built using a relatively recent innovation in combining reinforcement learning and deep learning. Chapter 2 will be devoted to an introduction to reinforcement learning, both conceptual and mathematical, while Chapter 3 will focus on how deep learning has been implemented in reinforcement learning. Then, Chapter 4 will give an introduction to bidding techniques used in RTB and, specifically, the paper by Wu et al. (2018), which Chapter 5 will be dedicated to replicating. Chapter 6 will then describe the experiments and their results, where the bidding agent will be compared to two benchmark techniques, along with a final discussion on the outcome of the project.

It should be noted that the content of this thesis is highly specialized and that the reader is thus expected to have some knowledge of stochastic processes and standard machine-learning techniques, specifically the neural network. While it would be desirable to include a section devoted to explaining the neural network in more detail, as well as adding more content on reinforcement learning, it is simply not feasible when also having to balance readability and the time constraint. However, I have done my best to present the material as clearly and pedagogically as possible and hope that this will suffice.

Chapter 2

Reinforcement Learning

The fundamental purpose of reinforcement learning is to design *agents* with the ability to successfully navigate through *environments* from which they have no prior experience. This does not necessarily mean that they have no prior knowledge of the environment whatsoever, although this is often the case as we shall see later, but rather that they haven't taken any actions in the environment previously; they have no idea of what kind of consequences or rewards follow from different actions.

Imagine a kid trying to learn how to ride a bike. The kid might understand how a bike works, e.g. that turning the handlebars to the right makes the bike turn right and that pushing the bike pedals makes the bike accelerate and go forward, and so on. However, there are a few things that only experience can teach. For example, it's difficult, if not impossible, to understand beforehand just how much the handlebar will make the front wheel turn. Similarly, it's hard to understand how much the bike will accelerate if we push the bike pedals forward or how harshly it will brake if we push the pedals backwards. Most importantly, it's impossible to know how much it will actually hurt to hit the ground, or if it will even hurt, when you fall off the bike if you haven't already done it, or how exhilarating it is to bike fast.

The latter example is of importance for reinforcement learning, since consequence and reward are how we make sure that an agent learns to behave in an optimal way in some environment. Just like the kid experiences pain and failure when it falls off the bike, we make sure our agent receives negative or low numerical rewards when choosing "bad" actions and, conversely, that it receives positive numerical rewards when it acts in a "good" way.

2.1 Markov Decision Processes

In any reinforcement-learning problem, we have an agent and an environment. These two interact with each other through *states*, *actions* and *rewards*. The environment provides a state, to which the agents responds with an action. Then, the action receives a numerical reward while the environment provides the next state, as a response to the agent's action. The goal of the agent is to maximize the cumulative reward over a number of states and actions, often referred to as an *episode*.

The most common way to model a reinforcement-learning problem is through the *Markov Decision Process* (MDP), which models state transitions using the Markov property. In this thesis, and in reinforcement learning in general, the finite MDP is of most importance, where there is a finite number of combinations of situations (or *states*) and actions as well as a finite (discretized) interval of rewards. In defining the general framework of a finite MDP, and the notation to be used later in this thesis, I will follow Sutton and Barto (2018).

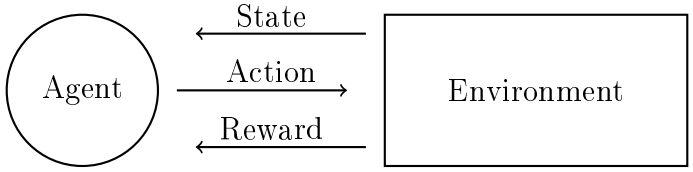


Figure 2.1: Illustration of a simple agent-environment relationship

We consider a finite series of time steps, $t = 1, 2, \ldots, T$, where T is the time of termination for the episode, and denote the action, state and reward at time t by A_t, S_t and R_t, respectively. We define a finite set for each: $A_t \in \mathcal{A}$, $S_t \in \mathcal{S}$ and $R_t \in \mathcal{R} \subset \mathbb{R}$. First, we want to consider the joint probability of some state, s', and some reward, r, following the choice of a certain action, a, in a certain state, s:

$$p(s', r, s, a) \triangleq P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

$\forall\, s', s \in \mathcal{S}$, $\forall\, r \in \mathcal{R}$ and $\forall\, a \in \mathcal{A}$. We have that $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \times \mathcal{R} \to [0, 1]$. That is, when considering the reward and the next state, we are only concerned with the previous state-action pair. We do not care about state-action pairs further back in the chain of transitions. Using $p(s', r, s, a)$, we want to define value of taking a certain action, a, in a certain state, s. In a probabilistic environment, this value should correspond to the sum of all possible immediate rewards, weighted by their

respective probabilities. Hence, we define the function

$$r(s,a) \triangleq \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a\right] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r, s, a)$$

such that $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Considering our biking kid, we might have a state in which the bike is on the crest of a hill and is just about to start rolling downwards. Our function, $q(s, a)$, then maps different actions, e.g. accelerating and breaking, to their perceived value. When finding these values, there's an important aspect to consider. For example, acceleration might yield some short-term exhilaration, but it also means some future risk as the kid will have less control over the bike.

Making a choice isn't just about weighing different immediate rewards against each other, it's also about weighing the present against the future, balancing the short-term and the long-term. This is also true for a reinforcement-learning agent, which we formalize using a *discount factor*, $0 \leq \gamma \leq 1$. A low γ means that the agent is *myopic*, prioritizing short-term rewards, while a high γ means that the agent will give consideration to future rewards.

Using the discount factor, we formulate the expected return at time t as

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

which gives us a recursive relationship, since

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

$$= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots + \gamma^{T-t-2} R_t) = R_{t+1} + \gamma G_{t+1}$$

such that $G_t = R_{t+1} + \gamma G_{t+1}$. We define $R = G_0$, i.e. such that the expected return for a whole period is denoted by R. Hence, the goal of an agent is to find the actions that maximize R. More specifically, instead of just considering the immediate reward form a state-action pair, $r(s, a)$, the goal of an agent should be to consider the entire reward following a state-action pair. For this purpose, we define

$$q(s,a) = \mathbb{E}\left[G_t|S_t = s, A_t = a\right] = \mathbb{E}\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \middle| S_t = s, A_t = a\right]$$

at time $t = 1, 2, \ldots, T$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The function $q(s, a)$ is called the *action-value function* and will be one of the most important conceptual features of this thesis.

2.1.1 An Optimal Policy

In reinforcement learning, a policy can be strictly defined as a mapping from states, s, to probabilities of selecting certain actions, a, in those states. A policy is usually denoted by π and we can hence express it as

$$\pi(a, s) = P(A_t = a | S_t = s)$$

$\forall\, a \in \mathcal{A}$, $\forall\, s \in \mathcal{S}$, and for $t = 1, 2, \ldots, T$. A policy can be strictly deterministic, meaning that the agent picks an action, a, with probability 1. If an agent follows a policy π, we say that $q_\pi(s, a)$ is the *action-value function for policy* π, since the subsequent state-action pairs are more or less deterministic, meaning that we can estimate the expected return if we're following a certain policy after (s, a). If a particular policy π_* has the property that $q_{\pi_*}(s, a) \geq q_\pi(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$ and for all other policies π, we call it the *optimal policy*. For this policy, we denote the action-value function by $q_*(s, a)$ and define it as

$$q_*(s, a) \triangleq \max_\pi q_\pi(s, a), \quad \forall\, a \in \mathcal{A}, \forall\, s \in \mathcal{S}$$

Hence, the purpose of the optimal policy is to maximize the expected return, G_t, at any time $t = 0, 1, 2, \ldots, T$.

2.1.2 The Bellman Optimality Equation

We consider the definition of $q(s, a)$ above. We also consider the recursive relationship for the expected return. Hence, we can also define the action-value function as

$$q(s, a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right]$$

Now, let's assume that we are following the optimal policy, π_*. If we have found the optimal policy, we are no longer exploring but only exploiting. In other words, we're only taking greedy action with probability 1. This means that, if $S_{t+1} = s'$, we know that $G_{t+1} = \max_a q(s', a)$. Hence, we can rewrite the action-value function as a recursive relationship:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r, s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned}$$

This equation says that the expected return from taking an action a in a state s when following the optimal policy corresponds to the immediate expected reward

and the expected return from following the optimal policy in the next state. This is known as the Bellman optimality equation for the action-value function. While this is analytically nice and solvable if we have knowledge of all transition probabilities and action values, it's applicability is constrained by computational complexity. Hence, many reinforcement-learning techniques, such as Monte Carlo methods, aim to approximate $q_*(s, a)$. This is also true for the method which will later be introduced as the *Deep Q-Network*.

2.1.3 Constrained Markov Decision Processes

So far, we've been concerned with an agent who's making decisions to maximize a single metric: the expected reward. In this case, we want to choose a policy π such that

$$\pi = \arg\max_{\pi} \mathbb{E}[R|\pi] = \arg\max_{\pi} \left[\sum_{t=1}^{T} \gamma^{t-1} R_t \middle| \pi \right]$$

which we've previously defined as the optimal policy, π_*. However, we've only been concerned with unconstrained maximization. It is not clear that π_* is an optimal policy when we impose constraints on the agents. We refer to such a case as a *Constrained Markov Decision Process* (CMDP). We define $C = \sum_{t=1}^{T} \gamma^{t-1} C_t$, where C_t is the cost at time t, and instead consider the problem of finding π such that

$$\begin{aligned} \max_{\pi} \quad & \mathbb{E}\left[R|\pi\right] \\ \text{s.t.} \quad & \mathbb{E}\left[C|\pi\right] \leq c \end{aligned}$$

where c is our cost constraint. How do we make this fit into the reinforcement-learning framework? Geibel (2007) discusses a number of methods fitted to different CMDP problems, one of which is to expand the state space, $\mathcal{S}$, to include the cost constraint. Instead of just considering the normal state-relevant parameters when taking an action, we also consider the cost incurred and if the constraint has been reached the agent will either be incapacitated or the period will be terminated. This is the approach that will be followed in this thesis.

2.2 Exploration and exploitation

As mentioned in the introduction to this chapter, reinforcement-learning techniques aim to deploy agents into new environments. This means that they have to *explore* the environment and essentially take random actions to see what happens, before being able to act intelligently. When the agent is acting intelligently and

taking the decisions that it knows maximizes the expected return, we say that it is *exploiting*. When the agent is only exploiting and not exploring, we call it *greedy*. Hence, the optimal policy, as defined by $q_*(s, a)$, is greedy since we're always choosing the actions that will maximize the expected return.

The exploration-exploitation trade-off is one of the most important aspects of reinforcement learning. On the one hand we want the agent to be as well-informed as possible about the actions it's taking, but on the other hand we want it to gain as much reward as possible; more exploration means less exploitation, and vice versa. This is usually solved by a so-called ϵ-greedy policy, which means that the agents exploits with a probability $1 - \epsilon$ and explores with a probability ϵ, where $0 \leq \epsilon \leq 1$. This is illustrated in the figure below with two possible actions.

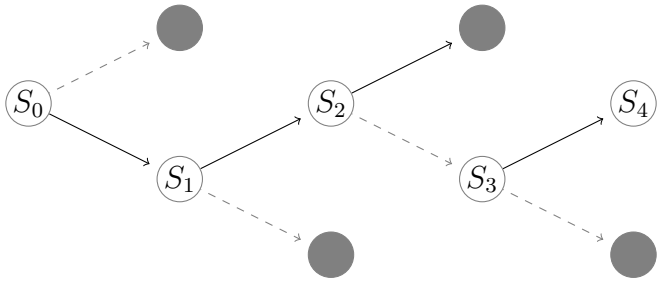


Figure 2.2: Illustration of an ϵ-greedy policy, where greedy actions are whole arrows and random actions are dotted arrows.

When the agent is learning, we want it to explore as much as possible, i.e. to have a high ϵ. Conversely, when the agent has finished learning, we want it to exploit, i.e. having a low ϵ or even $\epsilon = 0$. In practice, this can be solved by a number of ways. Often, it is the case that the agent starts out with $\epsilon \geq 0.9$ and then lets ϵ decay over time, e.g. linearly or exponentially, according to some fixed rate. Configuring an ϵ-greedy policy is by no means an exact science. Rather, it's a result of testing and applicability to the problem at hand.

In this thesis, we will be working with a limited action space with seven distinct actions, i.e. $|\mathcal{A}| = 7$, and $S_t \subset \mathbb{R}^7$ for $S_t \in \mathcal{S}$. We will attempt to use an adaptive ϵ which decays linearly according to some fixed rate but which tries to adjust itself if the agent seems to be in need of more exploration. This will be explained in more detail later on.

2.3 Q-Learning

One of the most famous RL methods is the model-free *Q-learning algorithm*, which, as the name suggests, is concerned with directly estimating the action-value function. While Q-learning deserves a longer theoretical background and discussion of its convergence properties, this will make for a shorter introduction as we will ultimately not be concerned with the Q-learning algorithm, but rather with a variant of Q-learning for which we can't necessarily make claims about stability and convergence.

We start by initializing some arbitrary action-value function $Q_0(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$ and as the agent is exploring the environment (as well as when it's exploiting), we're continually making incremental updates for the action-value function:

$$Q_{n+1}(S_t, A_t) = Q_n(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a) - Q_n(S_t, A_t) \right]$$

where α is the learning rate. That is, when observing S_t, A_t and R_{t+1}, we update the action-value function by the scaled difference between the old value, $Q_n(S_t, A_t)$, and the sum of the immediate reward and the discounted expected return for the next state under a greedy policy, $R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a)$. If we look closely, this is actually a familiar sight. Setting $\alpha = 1$, we get

$$\begin{aligned} Q_{n+1}(S_t, A_t) &= Q_n(S_t, A_t) + \left[R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a) - Q_n(S_t, A_t) \right] \\ &= R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a) \end{aligned}$$

which is analogous to the Bellman optimality equation for the action-value function for a greedy policy, i.e. where we choose the best action with probability 1. However, we're not concerned with transition probabilities anymore, as we're estimating action values with purely numerical methods (e.g. taking averages from large samples). It's important to note that the Q-learning update occurs at almost every step, meaning that the algorithm keeps updating the values for all state-action pairs even as ϵ decreases and the agent exits the exploring phase.

One of the problems with Q-learning is that we might run into trouble in large-scale systems since it's hard to visit every state-action pair a sufficient amount of times to get a good estimate of their values, especially when we have to balance exploration and exploitation. This is why we are now turning to the next chapter, where we will get a grasp of how we can approximate our action-value function and give our agent the power to generalize its experiences.

Chapter 3

Deep Reinforcement Learning

While traditional RL methods, e.g. the Q-learning algorithm, have nice properties with respect to stability and convergence, they're not always applicable when dealing have high-dimensional sensory inputs. Imagine that we want to create an agent with the purpose of playing 64-pixel arcade games, simply by "looking at" and analyzing the screen. This means that the input, i.e. the states, have dimensionality on the order of $64 \times 64 = 4096$. Hence, the number of unique states in the state space, $\mathcal{S}$, is potentially enormous, which means that Q-learning is likely to be computationally infeasible due to the number of state-action pairs.

Since the 90s, attempts have been made to find a more slick solution to estimating the value of action-value pairs in systems with large state-spaces. To date, the most prominent of these attempts is arguably the combination of deep learning and RL through the approximation of the action-value function with a neural network. This idea culminated in the projects by Google DeepMind, presented in Mnih et al. (2012, 2015), in which an RL agent, combined with a deep convolutional neural network, exceeded human-level performance in a number of arcade games by representing the state with 210×160 RGB visual inputs, i.e. dimensionality corresponding to $210 \times 160 \times 3 = 100800$.

3.1 Q-learning Powered by Deep Learning

While the approximation mechanism in a Q-learning algorithm is strictly local, i.e separate estimates for each state-action, the approximation mechanism in a deep neural network is global. In other words, combining an RL agent with a deep neural network gives it the ability to generalize its estimates. To consider an example, let's go back to our favorite biking kid. Imagine that the kid leans too much to the right, causing the whole bike to fall over and hit the ground. Now, if our young

biker's brain was wired like a Q-learning algorithm, the fall to the right, and the pain that came with it, would say nothing about what would happen in a similar situation where the bike was instead tilted to the left. This is of course absurd, and we should feel lucky we don't have Q-learning algorithms running the brain department, because the human brain has a powerful ability to generalize and draw comparisons. This is essentially the ability we want to give our RL agent; instead of having to thoroughly experience everything, we want it to be able to use limited experiences to get a comprehensive, general understanding of the environment.

One of the predecessors of DeepMind's arcade-game master was presented by Reidmiller (2005) under the name of *neural-fitted Q-iteration*. Reidmiller presented the problem as finding a tool to balance the positive and negative effects of using a global approximation, rather than a local one. While a global approximation might nullify the training from an older experience when adjusting the approximation based on a recent experience, it can also accelerate training significantly by exploiting generalization. The principal method proposed to achieve this goal is to store experiences and reuse them whenever the approximation of the Q-function is updated. This is based on the idea of *experience replay*, presented by Lin (1992)

In the previous chapter the update rule for the Q-learning algorithm was used, which was based on making incremental updates to the action-value estimates using the Bellman optimality equation:

$$Q_{n+1}(S_t, A_t) = Q_n(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q_n(S_{t+1}, a) - Q_n(S_t, A_t) \right]$$

Now, we consider similar update rule, but with a Q-function approximated by a neural network and parametrized by a set of weights, θ, such that we have $Q_n(s, a) = Q(s, a : \theta_n)$ and want to make updates by minimizing a loss:

$$\left(\left[r + \gamma \max_{a'} Q(s', a'; \theta_n) \right] - Q(s, a; \theta_n) \right)^2$$

with respect to the set of weights, θ_n, using e.g. a stochastic gradient descent (SGD) algorithm on previous experiences. In other words, we're concerned with finding a set of weights, θ, such that $Q(s, a; \theta) \approx Q_*(s, a)$. Reidmiller used this technique successfully on a number of simple control problems where the neural-fitted Q-algorithm found good policies relatively fast, compared to analytical model-based techniques. However, it was the introduction of two additional algorithmic features by the DeepMind team that eventually created the Atari-playing RL agent with superhuman game performance.

3.2 The Deep Q-Network

In 2012, a group of researchers from DeepMind, including the aforementioned Martin Reidmiller, released a paper which presented an algorithm that could successfully learn how to play a number of Atari games using a Q-learning agent powered by a deep, convolutional neural network (Mnih et al, 2012). The algorithm, called *Deep Q-Learning*, is very similar to Reidmiller's neural-fitted Q-iteration, except that it uses a convolutional neural network and a more efficient type of experience replay. It also incorporates a so-called *target network*, which is used to train the local network, i.e. the one used for decision making. The weights of the local network are then copied on to the target network at some pre-determined frequency.

Similarly to Reidmiller's algorithm, Deep Q-Learning fits an estimator to the Bellman optimality equation using the loss function

$$L_n(\theta_n) = \mathbb{E}_{s,a\sim\rho}\left[y_n - Q(s,a;\theta_n)\right]^2$$

where $\rho(s,a)$ is a probability distribution over states, s, and actions, a, and

$$y_n = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s',a';\theta_n^-)\middle| s,a\right]$$

where θ^- are the weights of the target network. Instead of using the entire replay memory, Mnih et al. (2012) take a mini-batch of samples from the experience replay memory and perform SGD, minimizing $L(\theta)$ with respect to θ using these samples. This is done at every step of the algorithm, using the target network. When a deep convolutional neural network is used, the estimator of Q is called a *Deep Q-Network* (`DQN`).

In 2015, Mnih et al. released another paper where the `DQN` was applied to 49 different Atari games. The agent achieved more than 75% of human score in more than half of the games, as well as beating humans in several games. To give an idea of how the problem was set up, and how the problem in this thesis will be set up, the authors used, for example, a mini-batch size of 32, a replay memory size of 100000, a target network update frequency of 10000, and a discount factor, γ, of 0.99.

3.2.1 Experience replay

The notion of experience replay is in no way new to Mnih et al. (2012, 2015), but the authors find a more efficient use of it by combining it with random sampling and SGD. However, the use of experience replay is primarily due to stability. In

games, and control problems in general, certain states are often interconnected, meaning that they are correlated and hence not independent (Mnih et al, 2012). In other words, a bias will appear when the agent is learning. Sampling from the replay memory remedies this by continuously letting the agent re-experience old state-action pairs. More specifically, we never let the agent improve it's estimations with immediate experiences; we always sample from the replay memory when updating the network, at every step. Reidmiller (2005) noted significant improvements in stability and training time from using the replay memory, making the learning process more stable and data efficient.

It should be noted that in the version of experience replay used by Mnih et al. (2012, 2015), there is a pre-determined size of the experience replay memory, whereas Reidmiller (2005) and Lin (1992) use all of the previous transitions. The main reason for this seems to be memory usage and computational feasibility; the simulations run on the Atari games can have millions of transitions. In the `DQN`-based RTB algorithm introduced later, we will also be using a pre-determined size for the experience replay memory. Whenever the memory is full, we remove the earliest experience when adding new experiences.

3.2.2 Target network

Together with experience replay, the target network is what really makes the `DQN` an efficient RL agent. When updating the decision-making network at every time-step, the risk of policy divergence and instability increases. If an update increases $Q(s_t, a_t)$, it is likely that $Q(s_{t+1}, a)$ also increases for all a, even though it's not a good estimate of the optimal policy (Mnih et al., 2015). Similarly to experience replay, the target network makes the learning process more stable and efficient. The authors show the effect of using experience replay and a target network for a number of games and where the effect on the agent's performance is astounding. While experience replay accounts for the greatest part, the use of a target network improves the performance of the agent many times over in several cases.

3.2.3 Summary

As mentioned previously, we refrained from discussing convergence and stability properties of Q-learning since the method we'd be using cannot make guarantees on convergence. Boyan and Moore (1995) were early to discuss this problem and showed that the combination of function approximation and certain RL methods could lead to serious instabilities and bad policies. It's evident that this problem is still pervasive, but in using the contributions by Lin (1992), Reidmiller (2005), and Mnih et al. (2015), we arrive at the `DQN` which does a good job in maintaining

both stability and efficiency. With exception for the convolutional neural network used by Mnih et al. (2012, 2015), this is the approach which will be followed when we construct a bidding agent.

3.3 Deep Reinforcement Learning in RTB

Reinforcement Learning was recently introduced into RTB when Du et al. (2017) and Cai et al. (2017) independently proposed RL-based bidding agents. The principal reason for this is the need for dynamism; instead of setting parameters for a whole batch of bids, e.g. which average bid and CTR estimation to use, we want a bidding agent which can achieve more strategic granularity, e.g. by observing campaign-relevant parameters and adjust its behavior in real time or by being able to make more fine-grained valuations of individual impressions.

Both RL-based methods use the demographic user information and the CTR estimations to capture the state dynamics, while the agent acts by setting bid prices. For example, Du et al. (2017) use historical data to estimate the winning probability of a certain bid price and the CTR, which then gives the probability of a click given a certain bid price. The goal of the agent is then to maximize the number of clicks under the budget constraint. While the methods from both papers outperform state-of-the-art linear bidding algorithms, they will not be the focus of this thesis. They use model-based training, which has problems with scalability due to computational complexity and with non-stationarity - and RTB is a highly non-stationary environment (Wu et al., 2018).

We will focus on creating a bidding agent using the approach of Wu et al. (2018), which uses historical data to train a bidding agent with a variant of the `DQN`; although, in this case there is no convolution, but a feed-forward neural network with three hidden layers, each having 100 neurons. Going forward, we will use the name '`DQN`' to describe the approximated Q-function used by the bidding agent.

3.3.1 Real-Time Bidding with a Deep Q-Network

In *Budget Constrained Bidding by Model-free Reinforcement Learning in Display Advertising* (2018), researchers from Alibaba Group discuss how they try to solve the problem of optimal bidding by using a `DQN`. This approach is completely different from the other RL-based approaches mentioned above. Instead of formulating bids by creating a model of the click probability from a given bid and then using this model to solve the constrained optimization problem of maximizing the

number of clicks under a given budget, the bids are formulated as:

$$b_{t,k} = \frac{\phi_{t,k}}{\lambda_t}$$

Let's consider what this expression means and how it explains the model. $b_{t,k}$ is the bid at step k, for $k = 1, 2, \ldots, K$, in time-step t, for $t = 1, 2, \ldots, T$. That is, one episode has T time-steps, where T is the time of termination. In each of these time-steps, the agent participates in K different auctions. $\phi_{t,k}$ is the CTR estimation for a particular auction, while λ_t is the bid-scaling parameter for that particular time-step. That is, all the bids in one time-step are scaled with the same parameter. The action of the agent is then to regulate the scaling parameter λ_t at each time step, t, depending on the state, S_t, which is described by

- the time step, t,
- the remaining budget, B_t,
- the number of regulation opportunities left at time t, ROL_t,
- the budget consumption rate, β_t, where $\beta_t = \frac{B_{t-1} - B_t}{B_{t-1}}$,
- the cost of the impressions won between time $t-1$ and t, CPM_t,
- the auction win rate, WR_t, and
- the total value of winning impressions, e.g. the number of clicks, at time step $t-1$, r_{t-1}.

Hence, the state can be describe as

$$S_t = (t, B_t, \mathrm{ROL}_t, \beta_t, \mathrm{CPM}_t, \mathrm{WR}_t, r_{t-1})$$

The agent thus considers campaign-relevant parameters rather than how the auction environment will react to it placing a bid and eventually winning an impression. For example, if the remaining budget is low and and the number of remaining budget regulation opportunities is high, the agent should ideally increase λ in order to decrease the bid scaling and, hence, bid less aggressively.

The auction space, $\mathcal{S}$, then consists of all of the possible values of the tuple, S_t. Since we're aiming for approximating our Q-function with a deep neural network, we don't have to consider discretization of the continuous parameters in S_t. The actions, i.e. the possible adjustments to λ_t, are $\mathcal{A} = \{-8\%, -3\%, -1\%, 0, 1\%, 3\%, 8\%\}$, such that

$$\lambda_t = \lambda_{t-1} \times (1 + A_t), \quad A_t \in \mathcal{A}$$

The reward, R_t, after some action A_t in some state S_t, is then given by

$$R_{t+1} = \sum_{k=1}^{K} X_{t,k}\phi_{t,k}$$

where $X_{t,k} = 1$ if the kth impression was won and $X_{t,k} = 0$ otherwise, while $\phi_{t,k}$ is the aforementioned CTR estimation for the kth bid. Similarly, the cost after some action A_t in some state S_t, is given by

$$C_t = \sum_{k=1}^{K} X_{t,k}c_{t,k}$$

where $c_{t,k}$ is the cost for the kth impression during time t. As mentioned previously, the cost constraint will be incorporated into the MDP by expanding the state when including the cost incurred and the budget, i.e. since $B_t = B_{t-1} - C_{t-1}$. Finally, we set the discount factor, γ, to $\gamma = 1$, since we consider all impressions to be equally important over the course of one episode.

Given S_t, the agent will estimate the value of taking different actions, i.e. using the Q-function. As previously mentioned, the Q-function will be approximated using a feed-forward neural network with three hidden layers, each having 100 neurons. It is not explicitly stated by Wu et al. (2018) which type of activation function is used in the network, but it's assumed here that they are using *ReLU activation functions*, as this is common for deep reinforcement learning applications, and for machine learning in general. The goal of the network is thus to evaluate all possible adjustments to λ given $S_t = (t, B_t, \mathrm{ROL}_t, \beta_t, \mathrm{CPM}_t, \mathrm{WR}_t, r_{t-1})$. The network is illustrated in figure 3.1.

The authors follow the same procedure as Mnih et al. (2015), using experience replay and a target network. They use a replay memory size of 100000 together with a mini-batch size of 32. For all of the samples in the mini-batch, they take the tuple (s, a, s', r) and set

$$y_n = \begin{cases} r & \text{if n+1 is terminal} \\ r + \gamma \max_{a'} Q(s', a'; \theta_n^-) & \text{otherwise} \end{cases}$$

and perform a gradient descent step on $(y_n - Q(s, a; \theta))^2$ with respect to θ. They use a target network update frequency of 100. The episode length is set to $T = 96$, with one day of bidding constitutes one episode. The authors use 7 days of bidding for training and 3 days of bidding for testing, using a number of different ad campaigns. The bids are thus allocated to the different time steps, most likely based

on their time stamps in the auction data. However, it's unclear what the "normal" step length is for any given campaign. It's also unclear whether or not they let their agent train on all of the campaigns and then test it on separate campaigns, or if the agent is trained and tested on separate campaigns.

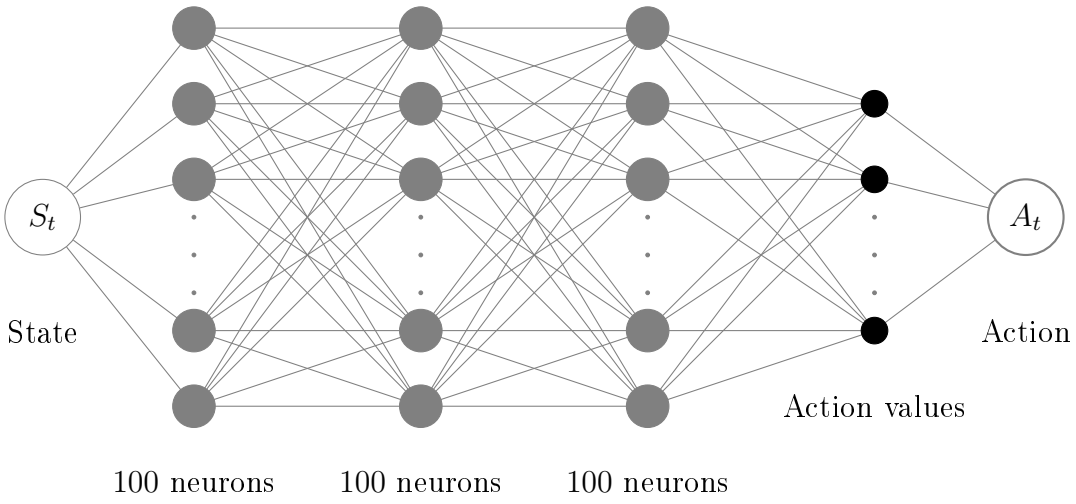


Figure 3.1: A feed-forward neural network with three hidden layers

The learning rate for the gradient descent algorithm when training the `DQN` is set to 0.001. For the ϵ-greedy policy, the starting value is set to $\epsilon_{\max} = 0.9$ which decays linearly to $\epsilon_{\min} = 0.05$. The authors also introduce two additional algorithmic features: an adaptive ϵ-greedy policy and an innovative reward function. Both of these features aim to increase the efficiency of the training and hence the performance of the network. The reward function also has the purpose of preventing the agent from getting stuck in suboptimal policies where it's just depleting the budget instantly. We will discuss both of these in more detail later in the methods and experiments sections. The authors call this method *Deep Reinforcement Learning to Bid* (`DRLB`). The interaction between the agent and the environment is illustrated below along with the pseudocode for `DRLB`.

Make illustration: the agent regulates the lambda, runs through a number of bids, observes the new budget, clicks won, impressions won, etc, sets new lambda, etc. Also include illustration of Q-learning update.

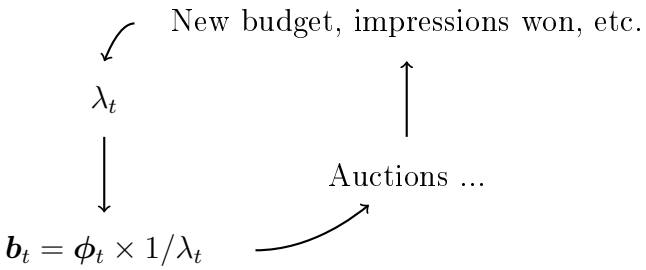


Figure 3.2: Illustration of how the bidding agent interacts with the RTB environment

Algorithm - `DRLB`

Initialize replay memory $\mathcal{D}$ to capacity $n_{\mathcal{D}}$
Initialize Q_{local} with random weights θ
Initialize Q_{target} with weights $\theta^- = \theta$
for Episode = 1 to N **do**
    Initialize λ_0
    for k = 1 to K **do**
        Bid with λ_0, s.t. $b_{0,k} = \frac{\phi_{0,k}}{\lambda_0}$
    end for
    for t = 1 to T **do**
        Observe state s_t
        Update ϵ
        Get action a_t from ϵ**-greedy policy**
        Set $\lambda_t = \lambda_{t-1} \times (1 + a_t)$
        for k = 1 to K **do**
            Bid with λ_t, s.t. $b_{t,k} = \frac{\phi_{t,k}}{\lambda_t}$
        end for
        Observe reward r_t and next state s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in replay memory $\mathcal{D}$
        Sample mini-batch from replay memory $\mathcal{D}$
        for j = 1 to 32 **do**
            if s_{j+1} is terminal **then**
                set $y_j = r_j$
            else
                set $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-)$
            end if
            Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
        end for
        Every 100 steps set $\theta^- = \theta$
    end for
end for

Figure 3.3: Pseudocode for the `DRLB` algorithm

Chapter 4

Method

While it might seem straightforward to set up and solve the problem given the lengthy descriptions in the previous chapter, this is far from the case. As machine learning is not an exact science, the difference between success and failure can lie in small, seemingly innocuous, details. Many of these kinds of details are not provided by Wu et al. (2018) for the `DRLB` algorithm and, in some cases, where they are provided they give rise to more confusion than clarity. For example, they do not disclose how they initialize the bid-scaling parameter λ, e.g. whether they use a fixed value for every episode or if they initialize the parameter randomly. They also omit seemingly important details, such as what kind of activation function they use in their neural network, or if they have had to deal with typical training problems, such as diverging action values. Most importantly, they haven't released any code for the article, meaning that I've had to build the project from scratch.

This chapter will be devoted to describing how I've tried to replicate the `DRLB` agent. First, I will describe how I set up the problem in terms of creating the RL agent and the RTB-auction environment. Then, I will describe the parts of the `DRLB` algorithms which are not entirely clear in the article and what kind of problems I've had with them, as well as how I've tried to solve them. Then, I will describe the methods that will be used for comparison and benchmarking in order to evaluate the bidding agent and, finally, I will describe the data used.

4.1 Setting up the problem

The first step towards creating and implementing a `DRLB`-inspired bidding agent was to create a working § agent. This basically meant creating an RL agent which has a deep neural network as a function approximator for the action-value function, $Q(s, a)$, an experience replay memory, a target network and an ϵ-greedy policy

with which it picks actions. The agent also needs to have the ability to train its deep neural network, as well as the ability to manage the target network. In order to create an agent with these properties, I used `python` programming. For all of the machine-learning aspects, I relied on Google's `tensorflow` library, especially `tf.layers` and `tf.train`.

However, much of the challenge in creating a functioning `DQN` agent is in making the whole machine work together. Another dimension of complexity is added when the agent also has to function together with a simulated environment. Thus, I started by creating a `DQN` agent for a much simpler problem, to get a better grasp of how it works and how to incorporate it into a more complicated setting. Once I had succeeded in creating an agent for this simple environment which was learning and completing the given task, I moved on to building the auction simulation environment and trying to replicate the `DRLB` agent.

4.1.1 Mountain Car test

The simple problem I chose to work on was OpenAI's environment '`MountainCar-v0`', where an agent has to learn how to drive a small car up a hill such that the car reaches a flag on top of the hill. However, in order to get up the hill, the car needs momentum. Hence, the agent has to learn to generate momentum before attempting to ascend the hill.

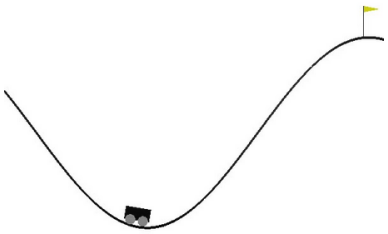


Figure 4.1: OpenAI's `MontainCar-v0` environment

There are three possible actions for the agent: to push left, to push right and to not push at all. The state is two-dimensional since it includes the horizontal position and the velocity. Additionally, we're concerned with a completely stationary and deterministic problem; for example, the position of the flag won't move and we know what happens if we push left (the car will go to the left). This is quite different from the RTB environment, where for example the market price for impressions can change dramatically over the course of a day.

Consequently, driving the mountain car doesn't require a neural network like the `DRLB` network illustrated previously. Instead, I used a much simpler architecture with two hidden layers, the first of which had 64 neurons while the second one had 32 neurons. The reason for this is to manage an accuracy-efficiency trade-off. Since we're working with a simple, low-dimensional problem, it's unlikely that we're going to reap any benefits in terms of accuracy by adding extra layers. At the same time, adding layers and neurons also means that we're adding training time. The network architecture is illustrated in the figure below.

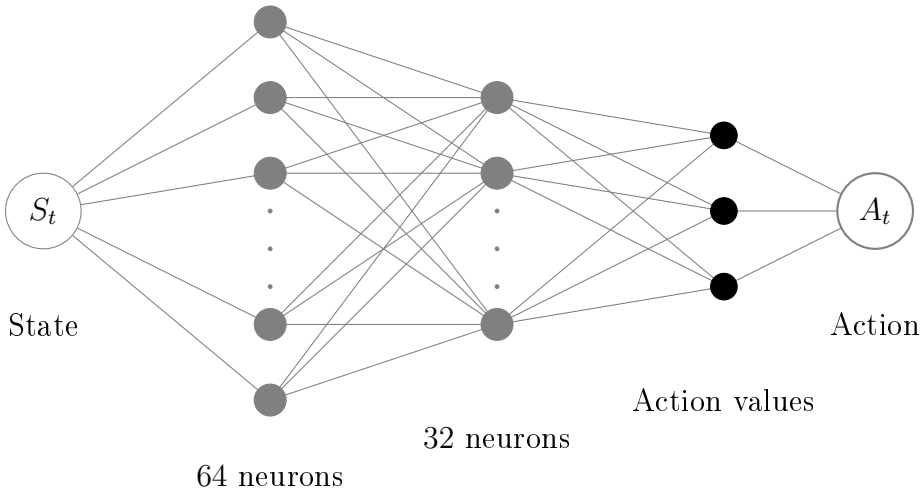


Figure 4.2: A feed-forward neural network with two hidden layers

Every episode is 200 steps and for each episode the code is similar to that presented above for the `DRLB` algorithm. I used an exponentially decaying ϵ, with $\epsilon_{\max} = 1.0$ and $\epsilon_{\min} = 0.01$. I also used a replay memory size of 50000, a mini-batch size of 50, a target network update frequency of 200 and a completely random selection of actions for 25000 simulated steps (in order to fill the replay memory). For every step that the car does not pass the flag, it gets a reward of -1. Below are the results from using the algorithm, as well as the pseudocode. The actual code has also been included in the `GitHub` repository.

It should be noted that the exponential decay used in the above example is relatively slow to converge. It is possible to create a `DQN`-based agent which learns how to master the mountain car a lot faster. However, the example was primarily for seeing that the algorithm worked in a simple environment and since it did, I

Algorithm - `DQN` for `MountainCar-v0`

Initialize replay memory $\mathcal{D}$ to capacity $n_{\mathcal{D}}$
Initialize Q_{local} with random weights θ
Initialize Q_{target} with weights $\theta^- = \theta$
for Episode = 1 to N **do**
  Observe the initial state s_0
  for t = 1 to 200 **do**
    Choose action a_t from ϵ-**greedy policy**
    Observe reward r_t and next state s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in replay memory $\mathcal{D}$
    Sample a mini-batch from replay memory $\mathcal{D}$ as (s_j, a_j, s'_j, r_j)
    for j = 1 to 50 **do**
      if s'_j is terminal **then**
        set $y_j = r_j$
      else
        set $y_j = r_j + \gamma \max_{a'} Q(s_j, a'; \theta^-)$
      end if
      Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
    end for
    Every 200 steps set $\theta^- = 0.05 \times \theta^- + 0.95 \times \theta$
  end for
end for

Figure 4.3: Pseudocode for a `DQN`-inspired algorithm

moved on to creating the RTB auction simulation environment.

4.1.2 Building the environment

The ultimate goal of this project is to use data from real auctions to simulate an environment in which the performance of a bidding agent can be tested. I solved this by creating a class which mimics the structure of environments in OpenAI's `Gym` library. The environment defines all of the possible actions, tracks all of the state-relevant parameters, the time step, and so on. In creating the state and how to make the environment respond to the agent's actions, I had to start using the paper by Wu et al. (2018).

First, the environment defined $\mathcal{A} = \{-8\%, -3\%, -1\%, 0\%, 1\%, 3\%, 8\%\}$ as the set of possible actions. Then, the environment tracks the values defining the state, i.e. the tuple $S_t = (t, B_t, \text{ROL}_t, \beta_t, \text{CPM}_t, \text{WR}_t, r_{t-1})$. The environment also manages

all of the data, fetching the necessary bids and CTR estimations from the dataset when a new step is taken, as well as tracking the step and time of termination. I formalized all of this in a `python` class, which has the functions of resetting the environment with an initial budget and λ_0 when a new episode is started, taking steps given a certain action, and returning results for a period, e.g. when testing the performance of the agent on a certain campaign. The class is itself initialized by taking pre-determined episode and step lengths and a `python` dictionary containing campaign-relevant information (or information on several campaigns), e.g. the bid data, the available budget, the number of impressions, and so on.

Wu et al. (2018) explicitly describe the state using the tuple above. However, as mentioned in the previous chapter, the problem with equipping RL agents with function approximators is that a global approximation might complicate the training process, rather than to make it more efficient. Let's consider how the state dynamics are described. The time step, t, will always be between 0 and 96. The budget, B_t, can be in the hundreds of thousands, or even millions. The budget consumption rate, β_t, will of course always be between 0 and 1; as will the winning rate. It's also the case that the remaining regulation opportunities, ROL_t, is just inversely proportional to the time step. That is, $\mathrm{ROL}_t = 96 - t$. Similarly, the cost, CPM_t, is already incorporated into the budget and the budget consumption rate. Hence, we have two considerations to make:

(i) are some parameters unnecessary?

(ii) are the different parameter ranges prone to create instabilities in the approximation?

Intuitively, the answer to both questions is a clear *yes*. In the experiments section, I will thus compare $S_t = (t, B_t, \mathrm{ROL}_t, \beta_t, \mathrm{CPM}_t, \mathrm{WR}_t, r_{t-1})$ to another tuple: $S_t = (B_t/B_0, \mathrm{ROL}_t, \beta_t, \mathrm{WR}_t, r_{t-1})$. Instead of including the absolute budget, we consider the current budget in relation to the initial budget. We also discard the cost and the time step. Hence, we will see how using fewer parameters with more uniform ranges will affect the performance of the agent. Finally, it's not entirely clear how many bids are processed at each time step. Wu et al. (2018) most likely use the time stamps in the auction data to allocate different bids to time steps and episodes. This information is not available here, meaning that the step length has to be fixed. However, even when adapting the step length to how the bids might be allocated, it's unclear how the authors manage to have an ϵ-greedy policy that explores sufficiently. In my experiments, I will let the agent train on all the campaigns and then test the performance on separate campaigns, using and experimenting with different step lengths.

4.2 Comparisons and benchmark

While RTB spending has grown explosively, the development of bidding strategies has not been as explosive. For all of the progress being made in machine learning and big data, DSPs are usually restricted to a relatively simple technique known as *linear bidding*, while often estimating values of different impressions with logistic regression models.

There are several reasons for this. For example, there hasn't been any publicly available data for research and benchmarking until a few years ago when a Chinese RTB company, iPinYou, released a large dataset for research purposes (Zhang et al., 2015). There are also some natural constraints in an RTB setting, e.g. the time constraint which means that any bidding algorithm has to be able to formulate a bid within 100 milliseconds of receiving a bid request from an AdX, as well as the non-stationarity of impression markets which make it even more difficult to design efficient, general models.

One of the most common techniques, the aforementioned linear bidding, uses the CTR, denoted here by ϕ, which tries to capture the probability of a given user clicking on a display advertisement. I will evaluate the performance of the `DRLB`-based bidding agent using a simple form of linear bidding, `LinBid`. The kth bid, b_k, is formulated by taking the average CTR over a large number of historical cases, as well as the average bid used in these auctions, and the current CTR estimation, ϕ_k:

$$b_k = b_{\text{average}} \times \frac{\phi_k}{\phi_{\text{average}}}$$

Together with random uniform bidding, `RandBid`, this model will be used as a benchmark. For random uniform bidding, we take the historical minimum bid and the historical maximum bid and sample bids from a uniform distribution, s.t.

$$b_k \sim \text{Uniform}(b_{\min}, b_{\max})$$

For both `RandBid` and `LinBid`, the set of training bids for a specific campaign will be used as historical data when evaluating the performance on that campaign. We will evaluate the algorithms by considering the number of impressions won, the number of clicks, the winning rate, the total budget spent, the effective cost per click (eCPC) and the effective cost per impression (eCPI).

4.3 Data

The principal source of data in this thesis will be the aforementioned dataset released by the chinese RTB company iPinYou in 2015. The primary reason for this is the ability to compare results to other papers and methods, since all of the three previously mentioned papers incorporating RL uses the iPinYou dataset. Zhang, Yuan and Wang (2015) released a paper together with the dataset, describing the data in detail together with some summary statistics. The dataset contains 9 different campaigns, each running over the course of a couple of days. Each campaign is divided up into one dataset for training and one for testing. In total, there are 15395258 (i.e. ~ 15 million) impressions in the training data and 4100716 (i.e. ~ 4 million) impressions in the testing data. The dataset is described in more detail below.

Once the primary testing and benchmarking has been done on the iPinYou dataset, I will be testing the agent on data from Adform, which has been retrieved and processed explicitly for this thesis. The Adform dataset contains XXX ($\sim Y$ million) bids, which has been divided into testing and training. This dataset is described in more detail below.

4.3.1 iPinYou

Since the `DRLB`-based agent formulates bids using the CTR estimations and then takes actions according to the budget, the budget consumption rate, and so on, we only need the winning bids, the CTR estimations, the total budgets and the total number of impressions. I have used the processed data by Du et al. (2017), where CTR estimations have already been made using impression-specific information and logistic regression, according to Zhang, Yuan and Wang (2014, 2015).

Wu et al. (2018) also use the iPinYou dataset when testing their `DRLB` algorithm. However, they have not released their processed data and they use a more complicated metric when evaluating the performance of their agent. We will be using the processed data from Du et al. (2017) as this allows performance comparisons with several other methods with respect to the number of clicks won and the eCPC. The iPinYou dataset is partially described in table 4.1.

4.3.2 Adform

What's in the Adform data? Is it the same as iPinYou or different in any way, e.g. w.r.t. the CTR estimations??? Describe how CTR estimations have been made.

Table 4.1: iPinYou Dataset

Campaign	Train impressions	Train clicks	Test impressions	Test clicks
1458	3083056	2454	614638	543
2259	835556	280	417197	131
2261	687617	207	343862	97
2821	1322561	843	661964	394
2997	312437	1386	150063	533
3358	1742104	1358	300928	339
3386	2847802	2076	545421	496
3427	2593765	1926	536795	395
3476	1970360	1027	523848	302

Describe division between training and testing. Give some background on where the bids are from.

4.4 Training the agent

The challenge of this project, or any reinforcement learning task, is to train the agent. We have the data and have created the environment; the next step is to make the agent navigate through it successfully. This is, of course, easier said than done. There are many details to consider and changing seemingly unimportant parameters can have a huge effect on the agent's learning process and subsequent performance. We will be working with and making test on several different hyperparameters, e.g. the step length, K, the decay rate for the ϵ-greedy policy, r_ϵ, the initial bid-scaling, λ_0, and the learning rate, α.

As mentioned previously, Wu et al. (2018) incorporate two additional algorithmic features into the `DRLB` procedure as described above: a new type of reward function, the `RewardNet` and an adaptive ϵ-greedy policy. The `RewardNet` uses a neural network with the same architecture as the Q-function approximator to consider the reward from entire episodes when taking a certain action in a certain state, rather than considering the immediate reward from a state-action pair. The point of this is to prevent the agent from immediately depleting the budget by decreasing λ. We will not be using the `RewardNet`, but instead show that inefficient budget depletion can be avoided by considering the ϵ decay rate, the step length and the learning rate, as well as by initializing the budget for respective episodes randomly during the training phase.

The adaptive ϵ-greedy policy checks if the distribution of the action values, i.e.

the outputs from the DQN, is unimodal, e.g. that the graph connecting the action values is first strictly increasing, then nonincreasing and then strictly decreasing. If the distribution of action values is found not to be unimodal, the ϵ is set to $\epsilon = \max(\epsilon_t, 0.5)$. The authors do not make it clear exactly how they test for unimodality. I tried creating a function which checked if the action values had a unimodal distribution, but it always found that this was false, i.e. basically setting $\epsilon = \max(\epsilon_t, 0.5)$ permanently. This, of course, led to inefficient and excessive exploration. Hence, I've discarded the use of this feature as well.

Wu et al. (2018) do not make it clear how they initialize the budget during training. In the iPinYou dataset, there is a specified cost for each campaign. It is not specified anywhere in the article by Wu et al. (2018) if they use this as their budget, or if they use some other, pre-specified budget. Neither is it clarified how they partition and initialize the budget for separate episodes during the same campaign. This is far from ideal, as the goal of the DRLB algorithm is partly to create an agent which can manage the spending of the budget in an optimal way. Hence, budget partition and initialization seem like important considerations to make in both training and testing. I have partly followed Du et al. (2017) in this respect. They use the specified training budget, B_{train}, in the iPinYou dataset during the training phase and then they scale the training budget using the proportion of impressions in the testing data, N_{test}, to the number of impressions in the training data, N_{train}. Then, they use a budget-scaling parameter, b_0, to see how the agent performs with different budget sizes. The testing budget is then defined by

$$B_{\text{test}} = b_0 \times \frac{N_{\text{test}}}{N_{\text{train}}} \times B_{\text{train}}$$

This is the testing budget I will use when running experiments and performance comparisons, which allows comparisons to the results presented by Du et al. (2017). However, during the training phase I will initialize the budget randomly using a normal distribution, setting the fraction of the total budget as the mean and testing for different variances:

$$B_{\text{episode}} \sim \mathcal{N}\left(\frac{N_{\text{episode}}}{N_{\text{train}}} \times B_{\text{train}}, \sigma_B^2\right)$$

where σ_B^2 is the variance for the budget initialization. We will see how this variance affects the performance of the agent its ability to deplete the budget in an efficient way.

The bulk of the experimentation will be devoted to finding a good set of parameters which lead to efficient training and a good performance with respect to

how the budget is spent, how many impressions are acquired and so on. While there are many hyperparameters to consider, we will focus on

(i) the step length, K,

(ii) the learning rate, α,

(iii) the ϵ decay rate, r_ϵ, and

(iv) the variance for the episodic budget initialization, σ_B^2.

We till test the stability of the agent by initializing it with different bid-scaling parameters, λ_0. We will primarily be running performance tests using the budget-scaling parameter $b_0 = 1/32$, since this is the same value used by Du et al. (2017). Hence, using $b_0 = 1/32$ will allow for making broader performance comparisons, in addition to benchmarking with `LinBid` and `RandBid`. Finally, the remaining hyperparameters will be set exactly as by Wu et al. (2018). That is, we'll use

- the starting value for ϵ, $\epsilon_{\text{max}} = 0.9$,
- the end value for ϵ, $\epsilon_{\text{min}} = 0.05$,
- the target network update frequency, $C = 100$,
- the experience replay memory size, $n_{\mathcal{D}} = 100000$,
- the episode length, $T = 96$, and
- the discount factor, $\gamma = 1$.

In order to be able to compare results, I've used `np.random.seed(1)` together with `tf.set_random_seed(1)` in every test. When training and testing for different hyperparameters, I've been able to use one of Adform's 32-core machines, which has been incredibly helpful and has allowed for parallel testing.

4.5 Modeling bias and constraint

In terms of wanting to know how well our agent could perform in an actual RTB setting, our methodology is far from ideal. The main problem is that offline evaluations inevitably impose a modeling bias. Li et al. (2012) discuss this for a similar problem. That is, it's unrealistic to assume that if we introduce a new bidder to an auction environment, who bids aggressively and depletes its budget, all of the other bidders will remain passive in changing their bidding strategy and simply accept winning fewer bids. Obviously, other bidders will react by increasing their

bids, since they also want to deplete their budgets and win as many impressions as possible.

However, this is the approach followed by Du et al. (2017), as well as Wu et al. (2018). While this type of offline evaluation might be problematic, it can at least give an idea of how the bidding agent will manage the budget and it's ability to form reasonable bids. It would be desirable to run a more complete simulation with several bidding agents interacting with each other, but we will settle for an offline test where the agent is interacting with a passive environment. This is partly due to comparability with Du et al. (2017) and partly due to the time constraint imposed on this thesis, since creating and evaluating a sufficiently realistic environment for simulations would take a considerable effort.

4.6 Stability testing

The bulk of the experimentation will be devoted to finding good parameters for the agent's performance. Then, we also want to test the stability of the agent. As previously mentioned, we will do this by testing its performance for different initial bid-scaling parameters, λ_0. We'll consider

$$\lambda_0 \in \{5 \cdot 10^{-2}, 10^{-2}, 5 \cdot 10^{-3}, 10^{-3}, 5 \cdot 10^{-4}, 10^{-4}, 5 \cdot 10^{-5}, 10^{-5}, 5 \cdot 10^{-6}, 10^{-6}\}$$

When searching for good parameters, we'll use $\lambda_0 = 10^{-4}$.

Chapter 5

Experiments and Results

This chapter will be devoted to optimizing the agent's performance and then comparing its performance to the simple benchmark algorithms presented above, `RandBid` and `LinBid`, as well as the results presented by Du et al. (2017), which includes a more sophisticated linear bidding algorithm, the RL algorithm presented by Cai et al. (2017), `RLB`, and their own model-based RL algorithms, `CMDP` and `Batch-CMDP`. For these latter ones, we only have information on the number of clicks and the eCPI.

First, I will present the results for the benchmarks and comparisons. Then, I will display some of the results from testing different parameters, as well as discuss the results. Finally, I will check the stability of the agent with respect to λ_0. Since we have deviated a bit from the `DRLB` algorithm as it is presented by Wu et al. (2018), I will instead call the algorithm we're using here `DRLB-0` (**OR MAYBE** `Batch-DRLB`???).

5.1 Comparative results

When testing `LinBid`, `RandBid` and `DRLB-0`, I've used

$$B_{\text{campaign, test}} = b_0 \times \frac{N_{\text{campaign, test}}}{N_{\text{total, train}}} \times B_{\text{total, train}}$$

where $N_{\text{campaign, test}}$ is the total number of impressions in the test data for the specific campaign, $N_{\text{total, train}}$ is the total number of impressions in all of the training data and $B_{\text{total, train}}$ is the total budget available in all of the training data. Since the testing data is divided up into different episodes for `DRLB-0`, any budget that has not been spent during an episode spills over to the next episode. The results for `LinBid` and `RandBid` are presented in table 5.1 and table 5.2, respectively. In

table 5.3, I'm presenting the results from Du et al. (2017) as they present it themselves, i.e. as clicks (eCPC), for the tuned linear bidding algorithm (`tLinBid`), `RLB`, `CMDP` and `Batch-CMDP`. The eCPC is measured in thousands.

Table 5.1: `LinBid`

Campaign	Impressions	Clicks	Cost	Win rate	eCPC	eCPI
1458	37351	105	1541903	0.06	14684.79	41.28
2259	16177	4	1046594	0.04	261648.50	64.70
2261	20306	9	862623	0.06	95847.00	42.48
2821	44596	15	1660619	0.07	110707.93	37.24
2997	20425	36	391497	0.13	10874.92	19.17
3358	11383	48	754918	0.04	15727.46	66.32
3386	27672	34	1368258	0.05	40242.88	49.45
3427	26117	47	1346624	0.05	28651.57	51.56
3476	22793	39	1314143	0.04	33695.97	57.66

Table 5.2: `RandBid`

Campaign	Impressions	Clicks	Cost	Win rate	eCPC	eCPI
1458	28024	20	1541902	0.05	77095.10	55.02
2259	15388	3	1046594	0.04	348864.67	68.01
2261	15346	0	862624	0.04	N/A	56.21
2821	23679	14	1660627	0.04	118616.21	70.13
2997	8794	24	391501	0.06	16312.54	44.52
3358	9202	3	754919	0.03	251639.67	82.04
3386	22191	18	1368261	0.04	76014.50	61.66
3427	21931	13	1346623	0.04	103586.38	61.40
3476	19110	10	1314143	0.04	131414.30	68.77

All of the four algorithms presented from Du et al. (2017) outperform `LinBid` and `RandBid` by some distance. This is no surprise, since they are all state-of-the-art algorithms. However, we will still mainly be focusing on `LinBid` and `RandBid` as our primary comparisons, since there is no information on total cost or number of impressions won for the algorithms in table 5.3.

Table 5.3: Results from Du et al. (2017)

Campaign	tLinBid	RLB	CMDP	Batch-CDMP
1458	464 (1.09)	424 (3.09)	464 (2.71)	462 (2.8)
2259	7 (173.52)	12 (101.2)	13 (89.47)	10 (119.16)
2261	9 (105.67)	11 (87.39)	8 (118.10)	7 (116.46)
2821	40 (40.26)	47 (39)	39 (45.32)	41 (45.05)
2997	64 (2.73)	82 (3.7)	71 (2.95)	71 (2.98)
3358	189 (3.77)	199 (4.29)	208 (3.38)	203(4.28)
3386	55 (5.52)	61 (21.21)	92 (12.99)	91 (14.26)
3427	203 (6.55)	261 (5.14)	292 (4.47)	292 (4.36)
3476	162 (5.92)	131 (9.87)	181 (7.16)	188 (6.82)

5.2 Parameter testing

When considering different parameters, we will start by looking at campaign 1458, as this is the campaign with the most training and testing data. However, we will still be training the agent using all of the training data from all of the campaigns. I considered

$$K \in \{1000, 500, 250\}$$

$$\alpha \in \{0.001, 0.0001, 0.000075\}$$

$$r_\epsilon \in \{0.00025, 0.0001, 0.000075, 0.00005, 0.000025\}$$

$$\sigma_B^2 \in \{0, 2500, 5000, 7500, 10000, 12500, 15000\}$$

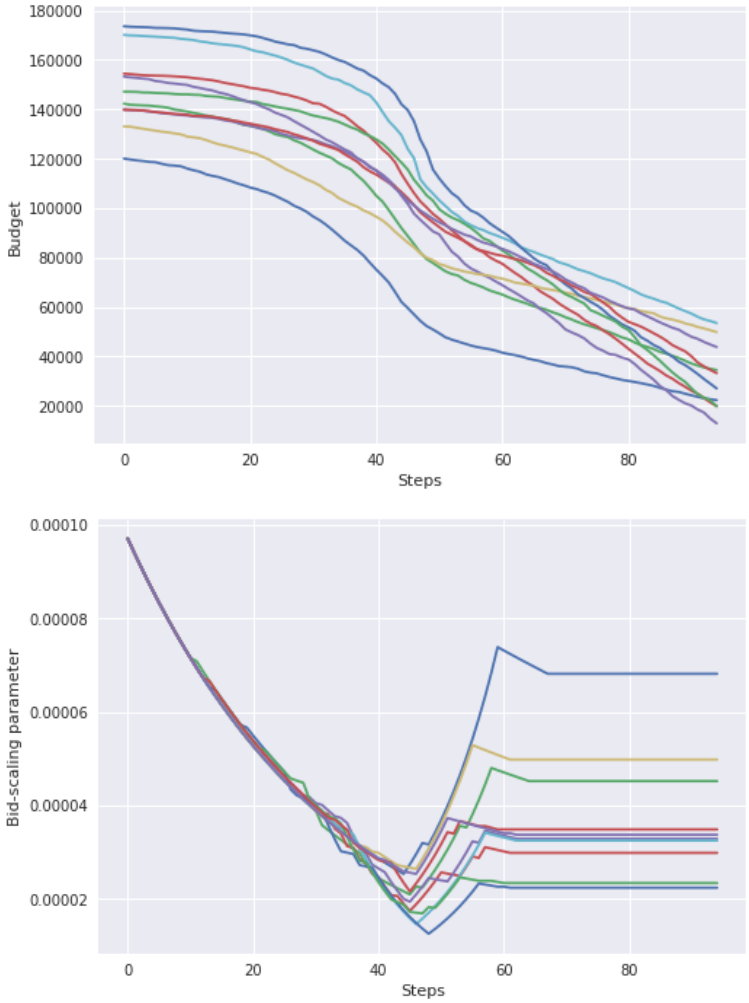
For $K = 500$, $\alpha = 0.0001$, $r_\epsilon = 0.0001$ and $\sigma_B^2 = 2500$, I got the following results:

Table 5.4: DRLB-0 for camp 1458

Campaign	Impressions	Clicks	Cost	Win rate	eCPC	eCPI
1458	54085	465	1499421	0.09	3224.56	27.72

Hence, in this case the DRLB-0 algorithm spent 97.2% of the budget and outperformed all other algorithms. During this test, I remodeled the state dynamics as discussed in section 4.4, i.e. discarding the cost and the absolute budget, as well as the time step, and instead using relative measures to decrease the dimensionality and to make the parameter ranges more uniform. I've plotted the spending of the budget as well as the bid-scaling parameter in the figures below.

These results give us a number of things to consider. For example, it's evident



that the initial bid-scaling should be even lower, since the agent is obsessed with decreasing λ to increase its bids in every episode and then has to increase λ so as not to deplete the budget too fast. This is actually not the case for many other parameter tests.

5.3 Stability tests with λ_0

Here, we want to discuss how the performance changes when the Lambda is initialized in certain ways, especially if the Lambda is initialized randomly with some consideration of the historical ratio between winning bids and CTR estimations.

Chapter 6

Results

References

Appendix A

Code