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.

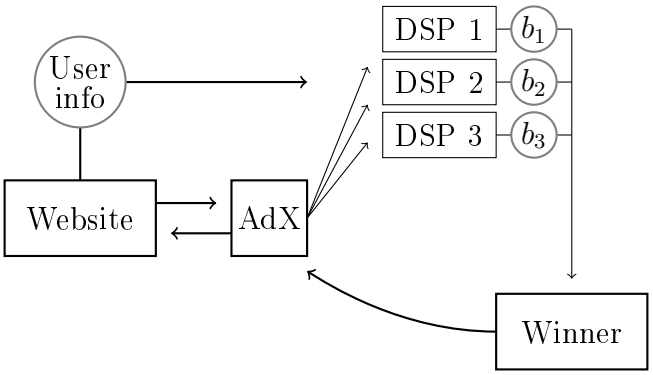

Figure 1.1: A simplified RTB ecosystem

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), **shortly discuss Markov process and 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).

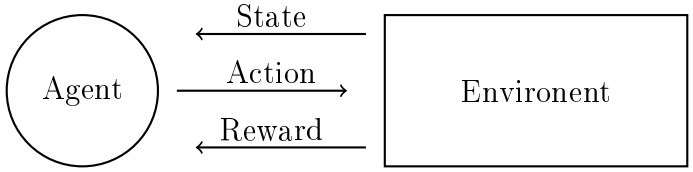

Figure 2.1: Illustration of a simple agent-environment relationship

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 $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \times \mathcal{R} \to [0, 1]$. Using this notation, we define value of taking a certain action, a, in a certain state, s, as

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

such that $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. 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 use this recursive relationship to re-define the function $q(s, a)$:

$$\vdots$$

$$\vdots$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. The reason for this re-definition will become clear shortly. 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 $t = 1, 2, \ldots, T$. If an agent follows a policy π, we say that $q_\pi(s, a)$ is the *action-value function for policy* π. 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 $t = 0, 1, 2, \ldots, T$. We define $R = G_0$, i.e. such that the expected return for a whole period is denoted by R.

2.1.2 The Bellman Optimality Equation

EXPLAIN MORE! We consider the definition of $q(s,a)$ above. Assuming that we are following the optimal policy, π_*, we can rewrite the action-value function as a recursive relationship:

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

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, 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. **Explanation why?**

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.

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

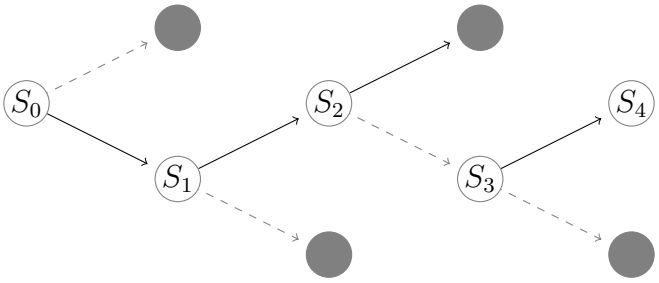

Figure 2.2: Illustration of an ϵ-greedy policy, where greedy actions are whole arrows and random actions are dotted arrows.

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

EXPLAIN MORE! 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

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

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. This is the update rule we will be using when constructing our agent later, with $\alpha = 0.001$. 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.

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 prin-

cipal 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 $S_t = (t, B_t, \text{ROL}_t, \beta_t, \text{CPM}_t, \text{WR}_t, r_{t-1})$. The network is illustrated in the figure below.

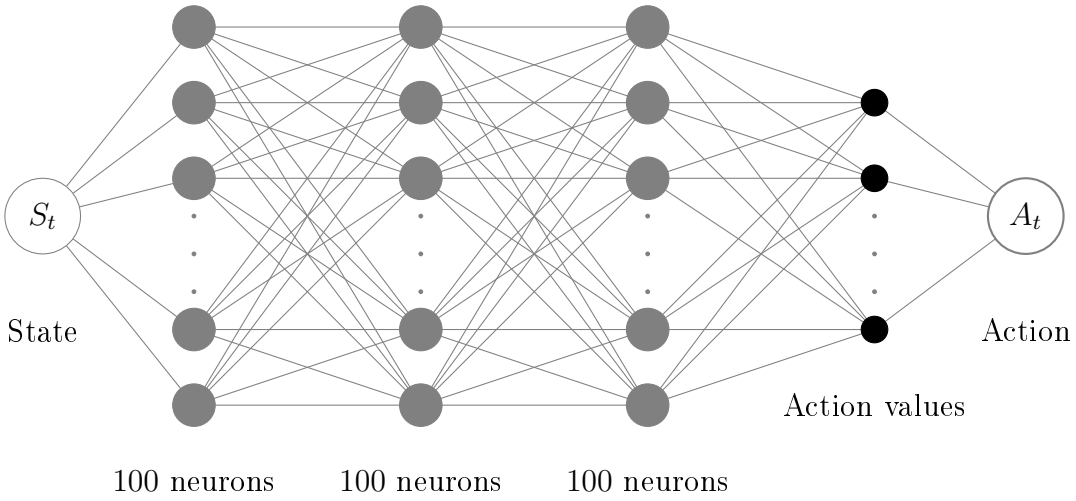

Figure 3.1: A feed-forward neural network with three hidden layers

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 $T = 96$, while the step length, i.e. how many auctions the agent participates in per time step, is unclear. The authors say that they use 7 days of bidding for training and 3 days of bidding for testing, using a number of different ad campaigns. The authors do not go into more detail as to what this means for how long a step should be.

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 $\epsilon_{\text{max}} = 0.9$ which decays linearly to $\epsilon_{\text{min}} = 0.05$. 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.

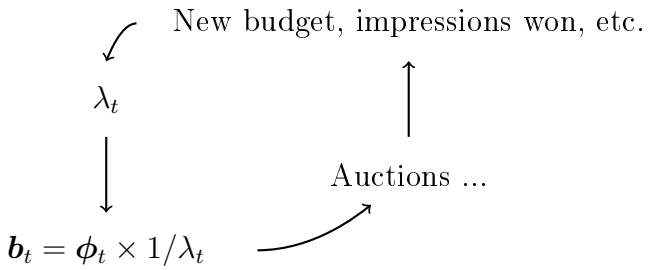

Figure 3.2: Illustration of how the bidding agent interacts with the RTB environment

Algorithm - `DRLB`

Initialize replay memory $\mathcal{D}$ to capacity N
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.

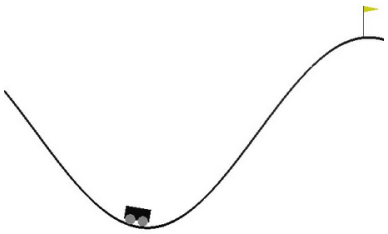

Figure 4.1: OpenAI's `MontainCar-v0` environment

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.

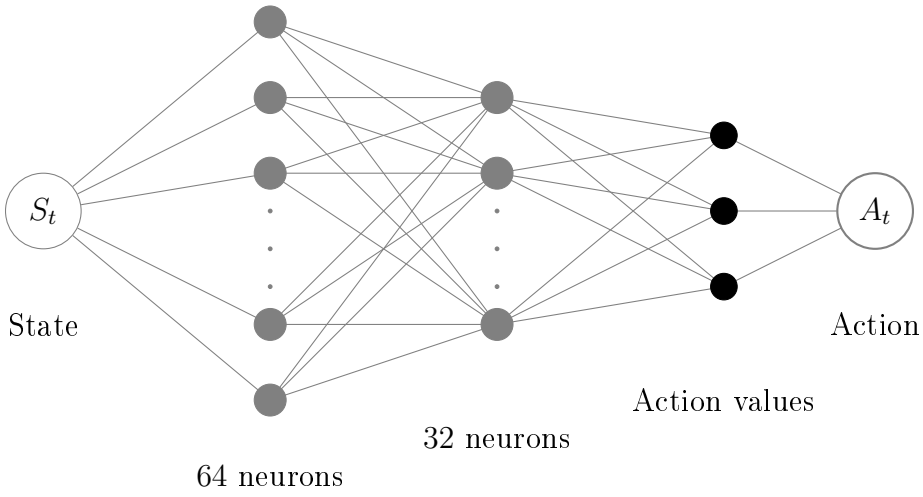

Figure 4.2: A feed-forward neural network with two hidden layers

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
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).

4.2 Results

With $r_\epsilon = 0.00005$, $\gamma = 1$, $C = 100$, $N = 100000$, $\lambda_0 = 0.0001$, $\sigma^2_{\text{budget}} = 5000$, $T = 96$, $K = 500$ and a budget-scaling parameter of 1/32, I got

Table 4.1: `DRLB`

Campaign	Impressions	Clicks	Cost	Win rate	eCPC	eCPI
2261	39691	9	844936	0.11	93881.78	22.86
3386	32010	66	1358975	0.06	20590.53	42.45
2259	42656	8	1046099	0.10	130762.38	24.52
2997	24548	62	378319	0.16	6101.92	15.41
3358	11103	202	673646	0.04	3334.88	60.67
3476	26871	165	1311514	0.05	7948.57	48.81
1458	53350	351	1540002	0.09	4387.47	28.87
2821	72677	32	1659420	0.11	51856.88	22.83
3427	28806	282	1278764	0.05	4534.62	44.39

Table 4.2: `LinBid`

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

4.3 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. The bid, b, is then 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:

$$b = b_{\text{average}} \times \frac{\phi_{\text{current}}}{\phi_{\text{average}}}$$

This model will be used as a benchmark when testing and evaluating the performance of a new model incorporating reinforcement learning and a `DQN`, along with random uniform bidding.