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

1 Reinforcement Learning **2**
1.1 Markov Decision Processes 3
1.1.1 An Optimal Policy 4
1.1.2 The Bellman Optimality Equation 4
1.1.3 Constrained Markov Decision Processes 5
1.2 Exploration and exploitation 6
1.3 Q-Learning 6

2 Deep Reinforcement Learning **8**
2.1 Q-learning Powered by Deep Learning 8
2.2 The Deep Q-Network 10
2.2.1 Experience replay 10
2.2.2 Target network 11
2.3 Summary 11

3 Real-Time Bidding **12**
3.1 Modeling a bidding agent 13
3.2 Reinforcement Learning in Real-Time Bidding 13
3.3 Real-Time Bidding with a Deep Q-Network 14

4 Method **17**

Chapter 1

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.

1.1 Markov Decision Processes

The most common way to model a reinforcement-learning problem is through the Markov Decision Process (MDP). 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).

We consider a finite series of time steps, $t = 1, 2, \ldots, T$, where T is the time of termination for the so-called *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.

1.1.1 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.

1.1.2 The Bellman Optimality Equation

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:

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

$$= \sum_{s',r} p(s',r,s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

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*.

1.1.3 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?**

1.2 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 before being able to act intelligently, essentially taking random actions to see what happens. 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 $0 \leq \epsilon \leq 1$.

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 start out with $\epsilon \geq 0.9$ and then lets ϵ decay over time, e.g. linearly or exponentially, according to some fixed rate. In this thesis, several approaches will be tried later on, but focus will be on a ϵ-greedy policy with an exponentially decaying ϵ. **Discussion on why?** The figure below illustrates an agent following an ϵ-greedy policy.

1.3 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. 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 2

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 $210 \times 160 \times 3 = 100800$.

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 Target network

Together with experience replay, the target network is what really makes the deep Q-network 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. Similarly to experience replay, the target network makes the learning process more stable and efficient. Mnih et al. (2015) show the effect of using experience replay and a target network for a number of games. The effect on the agent's performance is astounding and 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.

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 in this thesis.

Chapter 3

Real-Time Bidding

During recent years, online display advertisement has been revolutionized by a process now referred to as *Real-Time Bidding* (RTB). Zhang, Yuan and Wang (2014) provide a great introduction to RTB. Rather than advertisers buying specific keywords in search engines or certain fixed time slots on websites, many websites and advertisers have now instead turned to real-time auctions where the display advertisement slot is sold while a website is loading. Due to cookie technology, this means that advertisers can target internet users, or *impressions*, in specific demographic groups and direct their messages to the most appropriate group of people - when they want.

In terms of efficiency, this is quite the innovation. When advertising was dominated by newspaper ads and billboards, an advertiser would basically have to target a very large group, and only under a limited amount of time (as billboards and, especially, newspaper ads could be expensive). With RTB, advertisers can spend their budgets only on the type of people and impressions they actually want to reach.

However, there is a constraint: the process of auctioning out an ad slot and subsequently posting the ad from the successful advertiser often takes less than 100 milliseconds. To put this in perspective, blinking your eye takes about 300 to 400 milliseconds. Hence, buying impressions through RTB auctions is a purely algorithmic procedure and it's easy to see how RL fits into such an algorithmic setting: we have an agent which we want to maximize a reward, i.e. the number of impressions or clicks, under some constraint, i.e. the campaign budget, given a finite set of states and actions. The rest of this chapter will be devoted to describing how a bidding agent can be modeled using RL and, specifically, using the DQN described in the previous chapter.

3.1 Modeling a bidding agent

When an advertiser wants to buy an impression, they usually don't just go right ahead and access an online ad exchnage (usually called an *AdX*), but instead they enlist the help of a so-called *demand-side platform* (DSP). A DSP participates in auctions on behalf of companies or advertisement agencies who want to access markets for online display advertisement through RTB. This has become immensely popular in recent years, and spending on RTB has grown explosively, with a typical DSP handling billions of auctions on a daily basis (Yuan et al., ???, Survey on RTB...).

However, 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 often restricted to a technique known as *linear bidding*, while 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 so-called *click-through rate* (CTR), denoted here as 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.

3.2 Reinforcement Learning in Real-Time Bidding

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 observe campaign-relevant parameters and adjust its behavior in real time and hence achieve more strategic granularity.

Both RL-based methods use the demographic user information and the CTR estimations to capture the state dynamics, s.t. $\mathcal{S} \subset [0, 1]$, while the agent acts by setting bid prices. They 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 both of these methods outperform state-of-the-art algorithms using linear bidding, 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 DQN; altough in this case there is no convolution, but a feed-forward neural network with three hidden layers, each having 100 neurons.

3.3 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 discussed 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. Each of these time-steps means that 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 $\beta_t = \frac{B_t - B_{t-1}}{B_{t-1}}$,
- 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

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

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 to 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 $\mathcal{A} = \{-0.8, -0.3, -0.1, 0, 0.1, 0.3, 0.8\}$, 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 (???)**

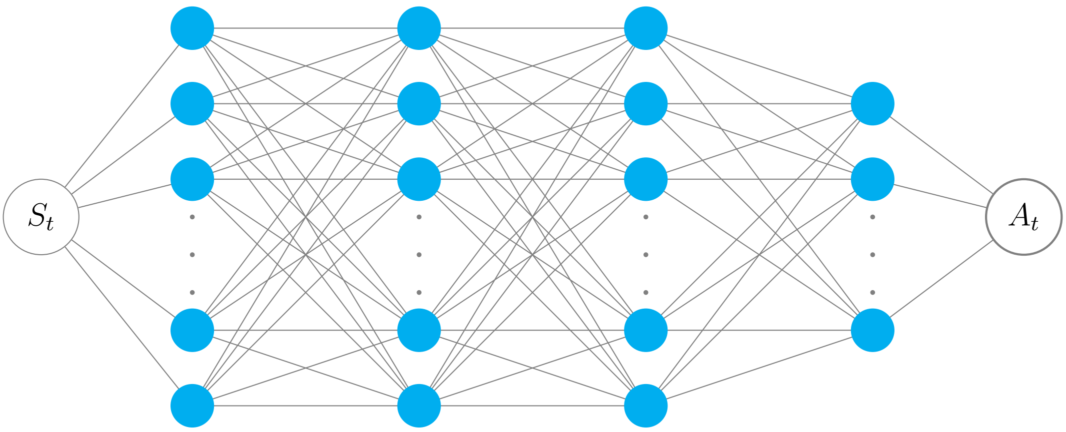
$$R_{t+1} = \sum_{k=1}^{K} X_{t,k}\phi_{t,k}$$

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 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-value 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 below:



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 $(y_n - Q(s, a; \theta))^2$. They then use a target network update frequency of 100, a time-step length of 96, i.e. $K = 96$, and an episode length corresponding to the length of an entire advertisement campaign. **DISCUSS ϵ-greedy and RewardNet??**

Chapter 4

Method