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

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 whatever we're doing, 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.

TIKZ-PICTURE FIGURE OF AN E-GREEDY POLICY

1.3 Q-Learning

One of the most famous reinforcement-learning methods is the *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, **probably??** 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

Refer to some general case.

2.1 Deep Q-Learning

2.2 The Deep Q-Network

Refer to Mnih et al. (2015).

2.2.1 Experience replay

2.2.2 Target network

Chapter 3

Real-Time Bidding

Refer in principal to Zhang, Yuan and Wang (20XX), and to Du et al. (2017), Wu et al. (2018), Cai et al. (2017), etc.

3.1 Modeling a bidding agent

3.1.1 Considering the framework in chapter 1

For example, how do we handle transition probabilities?? What does the action-value function look like?

3.2 Reinforcement Learning in Real-Time Bidding

3.2.1 Model-based approach

3.2.2 Model-free approach

Chapter 4

Method

4.1 Setting up the problem

Describe what was needed to actually consider the problem and experiment, i.e. in terms of programming. Mention that everything had to be reproduced from scratch. Specify use of tensorflow and so on.

4.1.1 Building the agent

Describe building the agent, in terms of programming, i.e. class-based solutions and how these were integrated into one agent.

4.1.2 Mountain Car test

4.1.3 Building the environment

Describe how environment was built and why.

4.2 Reproducing Wu et al. (2018)

Describe hyperparameters from the paper and how the agent and problem was used together with these to reproduce the model in the paper. Mention not attempting (yet) to reproduce RewardNet and adaptive ϵ-greedy policy. Also mention unclarity around budget etc.

4.3 Data

4.3.1 iPinYou

COULD use processed data from both Cai et al. (2017) AND Du et al. (2017)?? Discuss the data in Wu et al. (2018) as well.

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???

4.4 Modeling bias and constraints

Check the article sent by Morten!! Perhaps this should be in the discussion/results section??

4.5 Stability testing

Run simulations with exact same hyperparameters and compare results, using some metric, e.g. clicks won or winning rate. Then vary the hyperparameters and see how these changes affect the outcomes.

Chapter 5

Experiments

5.1 Importance of budget

It could be interesting to test the variance of the reward and win rate with different budgets, i.e. to check how an increased budget influences the results, if the relationship is linear, "logarithmic", etc.

5.2 Initializing the λ-parameter

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.

5.3 RewardNet and reward functions

ϵ-greedy policies

5.4 Dueling Deep Q-Networks

Chapter 6

Results

References

Appendix A

Proofs

Appendix B

Code