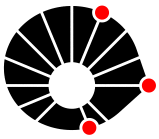

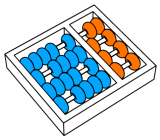

Sorting Permutations by Reversals and Transpositions with Reinforcement Learning

Guilherme Bueno Andrade *Andre Rodrigues Oliveira*
Zanoni Dias

Relatório Técnico - IC-PFG-18-13
Projeto Final de Graduação
2018 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Sorting Permutations by Reversals and Transpositions with Reinforcement Learning

Guilherme Bueno Andrade* Andre Rodrigues Oliveira† Zanoni Dias†

Abstract

TODO

1 Introduction

TODO

2 Reinforcement Learning

2.1 Overview

Reinforcement Learning (RL), like other branches of Machine Learning, has been drawing a lot of attention from the community in the recent years. Google DeepMind's AlphaGo victory over Lee Sedol [1] - world champion of the game Go, is one of many examples of recent astonishing applications of the technique. It is based an agent learning how to accomplish a certain goal based on interactions with the environment.

RL can be thought of as a sequence of episodes. Each of which consists of the agent initially at an initial state S_0. Then, based on a policy π, where a policy is a function that maps states to actions, it takes an action a, ending up at state S_1 and receiving some reward R_1. This process keeps going until the agent reaches a terminal state. Its goal is to find a function $\pi*$, known as optimal policy, that maximizes the cumulative discounted reward:

$$G(t) = \sum_{\tau=0}^{\infty} \gamma^{\tau} R_{t+\tau+1}, \tag{1}$$

Where:

$\gamma \in [0, 1)$: is a discount rate to highlight most likely short-term rewards

*gbuenoandrade@gmail.com

†Institute of Computing, University of Campinas, Brazil.

2.2 Exploitation vs. Exploration

A major concern in RL is the exploitation/explorarion trade-off. Exploration is about exploring new possibilities within the environment and finding out more information about it. Exploitation, on the other hand, is related to exploiting already known information so as to maximize the total reward.

Initially, the agent has no other option but to randomly explore the environment; however, as it learns about its surroundings, it can fall into the trap of sticking to safe known actions and miss larger rewards that depend on exploring unknown states.

This work uses the Epsilon-greedy strategy to address that problem. It specifies an exploration rate ϵ, which is set to 1 initially. This rate definies the ratio of the steps that will be done randomly. Before selecting an action, the agent generates a random number x. If $x > \epsilon$, then it will select the best known action (exploit); otherwise, it will select an action at random (explore). As the agent acquires more knowledge about the environment, ϵ is progressively reduced.

2.3 Q-table and the Bellman Equation

In value-based RL, the branch being considered in this work, the efforts are concentraded on maximizing the value function V_π:

$$V_\pi(s) = \mathbb{E}_\pi[G(t) \mid S_t = s] \tag{2}$$

It tells the agent the expected future reward it will get at each state. V_π can be generalized so as to also include actions, which is known as Q-table:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G(t) \mid S_t = s,\ A_t = a] \tag{3}$$

The previous definition is convenient because it allows the agent to pick the best action that can be performed from state s by simply finding $\arg\max_a Q^\pi(s, a)$.

Furthermore, Q can be expressed in terms of itself. An expression known as the Bellman Equation [ref]:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{a'} Q^\pi(s', a')], \tag{4}$$

Where:

s' : is the state reached after action a is taken from state s
a' : is an action that can be taken from s'

The above form can be used alongside dynamic programming to develop iterative approaches to solve the problem [ref].

2.4 Function Approximation

If one can calculate the Q-table from equation 3, they could trivially come up with a great policy. At each state s, the agent should simply be greedy and select the action a that maximizes $Q^{\pi}(s, a)$. As mentioned in the last section, finding the Q-table can be easily done with dynamic programming.

However, as the number of states grows largers, dynamic programming and other iterative approaches become unfeasible due to our computers' memory and time limitations. Fortunately, it turns out that the Q-table can be approximated instead of having its exact values determined, and it will still produce great results [ref]. This work tries to achieve that using both linear regression and deep neural networks.

2.5 Monte Carlo and Temporal Difference Learning

In Monte Carlo Approaches, the agent plays an entire episode, keeping track of the rewards received at each timestep so it can calculate the cumulative discounted reward. After that, it updates the value function for each visited stated based on the expression [ref]:

$$V(S_t) \leftarrow V(S_t) + \alpha[G(t) - V(S_t)], \tag{5}$$

Where:

$$\alpha : \text{is the learning rate}$$

Therefore, the agent only learns after an entire episode has been played. In Temporal Diference Learning, however, the value of V is updated after each timestep. At time $t + 1$, the observations made during time t are already being considered. In its simplest form, the method is called TD(0) or 1-step TD, and its update equation is as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{6}$$

The right-hand side of the previous equation is referred to as the TD(0) error.

2.6 TD-Lambda

TD(0) is biased as it relies on information from a single timestep to perform its updates. It does not take into account the fact that the action that caused a reward might have happened several timesteps earlier, which can lead to slow convergence [ref]. Monte Carlo methods, although not biased, have a lot of variance since they use the rewards of an entire episode in order to perform updates [ref].

To address that, from equation 6, we can define the 1-step return, $G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$. We can extend the concept further to 2-step return, $G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$, and, generically, to, $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^n V(S_{t+n})$.

TD-Lambda methods use a mathematical trick to average all the possible n-step returns into a single one. This is done by introducing a factor $\lambda \in [0, 1]$ and weighting the nth-return

with γ^{n-1}. It can be shown [ref] that when $\lambda = 0$, the method is equivalent to TD(0), and when $\lambda = 1$, equivalent to Monte Carlo. So, intuitively, by setting $0 < \lambda < 1$, we can get a mixture of both methods [ref].

2.7 Q-learning

Q-learning is another technique based on Temporal Difference Learning to learn the Q-table. The main difference between it and the previous shown techniques is that Q-learning is off-policy, while TD(0) and TD-Lambda are on-policy [ref]. This is reflected in its update equation, derived from the Bellman Equation [ref]:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R_{t+1} + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{7}$$

The fact that there is no constraint regarding the action a', only that it must optimizes Q, makes it an off-policy method [ref].

2.7.1 Deep Q-learning

In order to approximate the Q-table to make using it feasible even when the number of states is very large, since Google AlphaGo's paper [ref], it has becoming common the use of a deep neural network. Even though standard Q-learning is proven to converge when there are finite states and each pair state-action is presented repeatedly [ref], the same proof does not hold when neural networks are being used to calculate Q. To face with this issue, this work makes use of several ideas found in the literature [ref] to stabilize the training. They are presented in the following subsections.

2.7.2 Experience Replay

During each step of an episode, our estimation can be shifted according to equation 7. It is reasonable to think that as more truth is being introduced into the system, it will eventually converge; however, this is often not true. One of the reasons is that samples arrive in order they are experienced and as such are highly correlated. Also, by throwing away our samples immediately after we use it, we are not extracting all the knowledge they carry.

Experience replay address that by storing samples into a queue and during each learning step, it samples a random batch and performs gradient descend on it. As samples get old, they are gradually discarded.

2.7.3 Target Network

The points $Q(s,a)$ and $Q(s',a')$ in equation 7 are very close together as $s'a$ is a state reachable from s. That being so, updates to one of them influence the other, which in turn can lead to instabilities.

To overcome that, a second network, called target network, can be used to provide more stable $\widetilde{Q}$ values. This second network is a mery copy of the first one; however, it does not get updated every simulation step. Instead, it stays frozen in time, and only after several

steps it is updated by copying the weights from the first network. The introduction of the target network changes our update equation to the following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R_{t+1} + \gamma \max_{a'} \widetilde{Q}(s',a') - Q(s,a)] \tag{8}$$

2.7.4 Double Learning

Due to the max term presented in both equations 7 and 8, the approximation tends to overstimate the Q function value, which can severely impact on stability of the algorithm [ref]. A solution proposed by Hado van Hasselt (2010) [ref], called Double Learning, consists of two Q functions, Q_1 and Q_2, that are independently learned. One function is then used to determine the maximizing action and second to estimate its value. As we are already making use of two different networks, Hado van Hasselt's approach can be easily introduced to the update equation 8. Its augmented version is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R_{t+1} + \gamma \widetilde{Q}(s', \operatorname*{argmax}_{a'} Q(s',a')) - Q(s,a)] \tag{9}$$

3 Experiment

3.1 Modeling

3.1.1 State Representation

Three different state representations were considered. They are as follows.

- *One-Hot Encoding*: Each number is treaded as a category. By doing so, each p is mapped to a matrix m, where $m_{ij} = 1$ if $p_i = j$, and 0 otherwise.

- *Min-Max Normalization*: P is mapped to an array V,
  where $v_i = p_i/(|p| - 1)$.

- *Permutation Characterization*: p is mapped to an array v of 30 features, where the features are the ones described in the work by Flavio (2017) [ref].

3.1.2 Proposed Architecture

The experiment was based on two different architectures. The first one consisted of a Double Deep Q-Network (DDQN), with a main network and a secondary target network. Also, it included both experience replay (see 2.7.2) and double learning (see 2.7.4) optimizations.

Both main and secondary networks used the following architecture:

Layer	Number of Units	Activation	Type
Input layer	-	-	-
Hidden layer 1	256	ReLU	Fully-connected
Hidden layer 2	256	ReLU	Fully-connected
Output layer	|Actions|	Linear	Fully-connected

Table 1: Main and secondary networks architecture

Furthermore, the hyperparameters selected were as follows:

Hyperparameter	Value
γ	0.99
α	0.001
|Batch|	32
ϵ_{min}	0.1
ϵ_{decay}	0.993
|Replay queue|	2000
Loss	logcosh
Optimizer	Adam
Step reward	-1

Table 2: Hyperparameters used in the DDQN

Nevertheless, we tried a second different approach. An agent based on the TD-Lambda algorithm was also built. In order to approximate the Q-table, it relied on a much simpler linear regressor. Also, since Q is highly nonlinear (see equation 4), the radial basis function kernel was used as a pre-processing step. The agent's hyperparameters were the following:

Hyperparameter	Value
γ	0.999
α	0.01
λ	0.25
ϵ_{min}	0.05
ϵ_{decay}	0.99
RBF Components	4
RBF γ array	$[5.0, 2.0, 1.0, 0.5]$
Step reward	-1

Table 3: Hyperparameters used to implement TD-Lambda

3.1.3 Speeding up the Convergence

Despite being finite, the state space is still enormous given that it is in $\mathcal{O}(|p|!)$. Being that the case, several techniques were tried so as to decrease training time. The ones that seemed more promising are mentioned below:

- *Greedy Pre-Training*: Fulano (1990) [ref] presented a 2-approximation for sorting a permutation using reversals, its output is represented below:

$$d(p) = (p,\ a(p),\ a'(a(p)),\ \ldots,\ \iota), \tag{10}$$

Where:

a : is the resultant permutation after applying action a on p
ι : is the identity permutation

From expression 10, let $\hat{V}$:

$$\hat{V}(s) = (\gamma^{|d(s)|} - 1)/(\gamma - 1) \tag{11}$$

Furthermore, we can define function $\hat{Q}$, which can be shown to be lower bound on Q:

$$\hat{Q}(s, a) = -1 + \gamma \hat{V}(s'), \tag{12}$$

Where:

s' : is the state reached by taking action a at state s

Having defined those, we can talk about our pre-training strategy. It consisted of partially fitting our neural network and linear regressor to function $\hat{Q}$ before start learning from proper episodes.

3.2 Results and Discussion

TODO

References

[1] T. Guardian. Alphago seals 41 victory over go grandmaster lee sedol. https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol, 2016. accessed June 05, 2018.