UPPSALA
UNIVERSITET

IT 19 053
Examensarbete 30 hp
Augusti 2019

Warehouse Vehicle Routing using Deep Reinforcement Learning

Johan Oxenstierna

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Warehouse Vehicle Routing using Deep Reinforcement Learning

Johan Oxenstierna

In this study a Deep Reinforcement Learning algorithm, MCTS-CNN, is applied on the Vehicle Routing Problem (VRP) in warehouses. Results in a simulated environment show that a Convolutional Neural Network (CNN) can be pre-trained on VRP transition state features and then effectively used post-training within Monte Carlo Tree Search (MCTS). When pre-training works well enough better results on warehouse VRP's were often obtained than by a state of the art VRP Two-Phase algorithm. Although there are a number of issues that render current deployment pre-mature in two real warehouse environments MCTS-CNN shows high potential because of its strong scalability characteristics.

Handledare: Lijo George
Ämnesgranskare: Michael Ashcroft
Examinator: Mats Daniels
IT 19 053

Contents

1. Abbreviations/Terminology

Batching/order interleaving: General term for generating combinations out of a number of boxes/items/orders/picklists that need to be picked in a warehouse. If good combinations are found picking can be improved in terms of e.g. lower distance or times. It can be synonymous to many of the VRP types (see below) depending on constraint characteristics.

Box-calculation: The calculation of which items should go in which box for shipping. It is used when a customer orders many items that cannot fit into a single box.

DP: Dynamic Programming.

DRL: Deep Reinforcement Learning.

KPI: Key Performance Indicator.

MDP: Markov Decision Process.

NP-hard: A family of problems which are not solvable in polynomial time as problem complexity grows.

TSP: Traveling Salesman Problem. The general TSP case is a VRP with one vehicle. The TSP is a subset of the VRP.

Pick location: Geometric location in the warehouse where an item is picked. In this study the word *node* is also often used to denote pick location.

Pick run: Synonymous to pick route/round/path i.e. the sequence of pick locations passed by a single vehicle to pick a set of items.

UCB/UCT: Upper Confidence Bound/Tree.

Vehicle: Generic term for mobile units, trucks, forklifts, trolleys etc. that are filled with boxes or pick items during the pick runs.

VRP: Vehicle Routing Problem. Laporte et al. (2013): “The vehicle routing problem (VRP) consists of designing least cost delivery routes through a set of geographically scattered customers, subject to a number of side constraints”. “Customers” are often represented as nodes in a graph, where all node-node distances can be obtained by a pre-computed distance matrix. Variants to the VRP: VRP-PD: Pick-up and Delivery - There are multiple pickup and delivery locations. CVRP: Capacitated – The vehicles have a limited carrying capacity. VRP-LIFO: Last In First Out - The last item picked must be the one first delivered. VRPTW: Time Windows - The deliveries must be carried

out within certain time constraints. VRP-MT: Multiple Trips - The vehicles can do many pick-runs. OVRP: Open - Vehicles do not have the same start and end location. MVRP: Multi – There are multiple vehicles picking the items. All of the above versions to the VRP are considered NP-hard. Warehouse VRP's are made more complex by rack obstacles and box-calculation and time – window constraints, covered in section 3. In warehouses the VRP type is usually "batching" (see above).

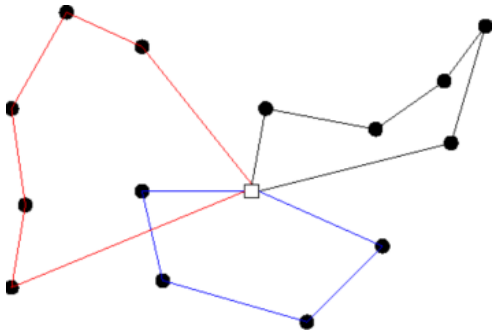


Figure 1: Drawing of a standard VRP with a single depot (square). The red, blue and black lines could represent multiple vehicles in which case it would be a MVRP. If it is a single vehicle it would be a VRP-MT. In the real warehouse datasets there are examples of both of these cases.

If the VRP includes changes during the routing it is dynamic (DVRP) and the below figure covers many of those variants:

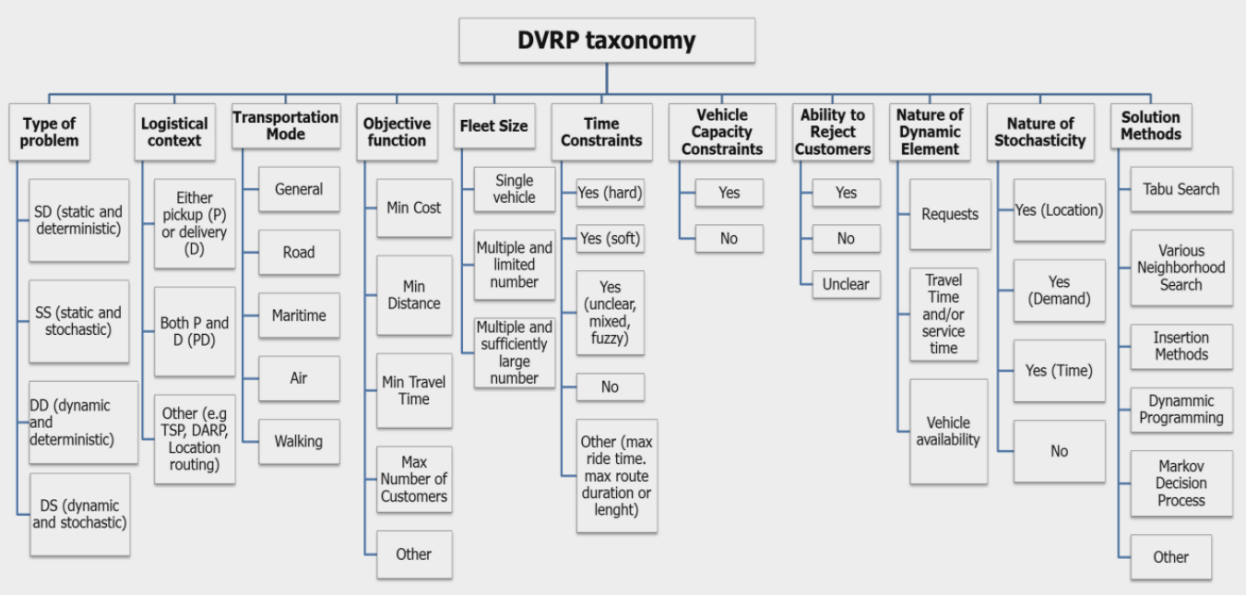


Figure 2: DVRP taxonomy by Psaraftis et al., 2015.

WMS: Warehouse Management System. The system controlling overall warehouse operations.

2. Introduction

The Vehicle Routing Problem (VRP) is an old and well-known algorithmic challenge. When studied in the context of modern warehouse operations the problem can reach tremendous complexity due to a multitude of intertwined processes. Development of delineated problem formulations and new ML (Machine Learning) methodology in this domain are welcomed by research since the VRP belongs to the most difficult family of problems in computer science, and by industry since menial human labor can be decreased while profits increased (Bartholdi and Hackman, 2005). In warehouses picking time constitutes 55% of total work time hours and, in the words of De Koster et al. (2007): "it costs labor hours but does not add value": [it is] "waste".

Traditionally warehouse VRP's are optimized using domain specific *heuristics*, *meta-heuristics, two-phase* algorithms, *dynamic* and *constraint* programming (section 4). These methods are mostly "lazy" which means that they operate without "learning" in an ML sense (there is no or very little pre-training), and it is a challenge to keep computational time low with growing problem complexity. They are also difficult to maintain and scale because of the difficulties in designing and understanding heuristics. In this study the extent to which warehouse VRP environment features can be autonomously learnt by a Deep Convolutional Neural Network (CNN) that is then used in realtime Monte Carlo Tree Search (MCTS) is investigated. The end goal is an algorithm highly scalable and transferrable from one warehouse to another. It is inspired by the Alpha Go Zero (AGZ) algorithm with the same basic MCTS-CNN construction blocks, but with necessary modifications due to the differences between Go and the VRP. A key difference is that the aim in a VRP is not to win an adversarial game, but to optimize distance travelled or some other scalar Key Performance Indicator (KPI).

The algorithm is developed with and tested on simulated data but it is also discussed in context of real warehouse VRP data, on which other VRP algorithms have already been deployed in an industrial setting (a reference algorithm *Two-Phase* is provided by *Sony-TenshiAI* which serves as supporter of this investigation). The reason the DRL algorithm is not directly studied in a real setting is because the domain is deemed too intricate for a direct application. The use of a simulated environment allows for better control over problem complexity and algorithmic performance evaluation.

3. Datasets, Delimitations and Aim

This section introduces the "real" warehouse VRP environment and how a thesis scope investigation can be demarcated. Two warehouse datasets (anonymized here) were available:

Warehouse A: Forklift box and batching pick area with dimensions 100 x 250 meters with 26000 pick runs.

Warehouse B: Trolley box and batch pick area with dimensions 80 x 70 meters with 10000 pick runs.

The pick runs in both datasets are represented in table format row-row with pick run, pick location identifiers and timestamps of when pick locations were visited. There is also in some datasets a time of how long it took to pick an item and this feature is very noisy and there is not sufficient corresponding data on why certain picks take certain time.

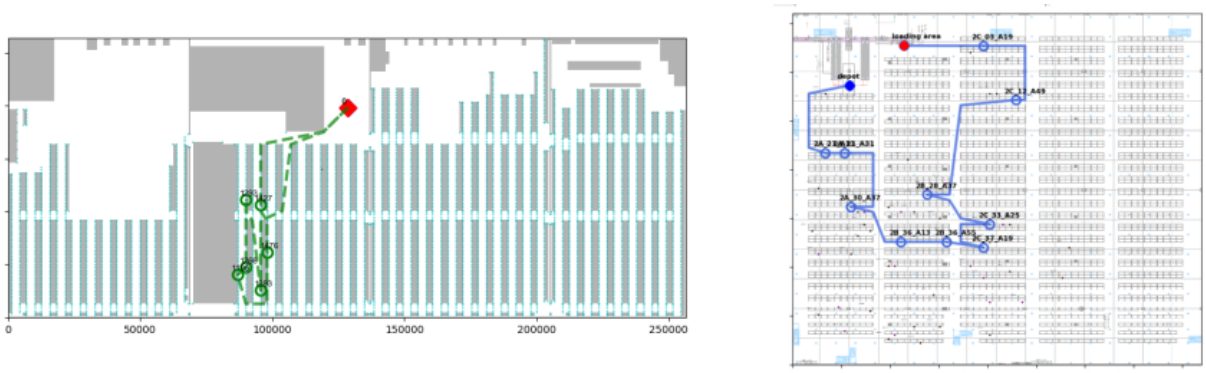


Figure 3: Maps of warehouses A and B (seen from above). The lines and circles are suggested single pick runs i.e. TSP's that navigate around racks.

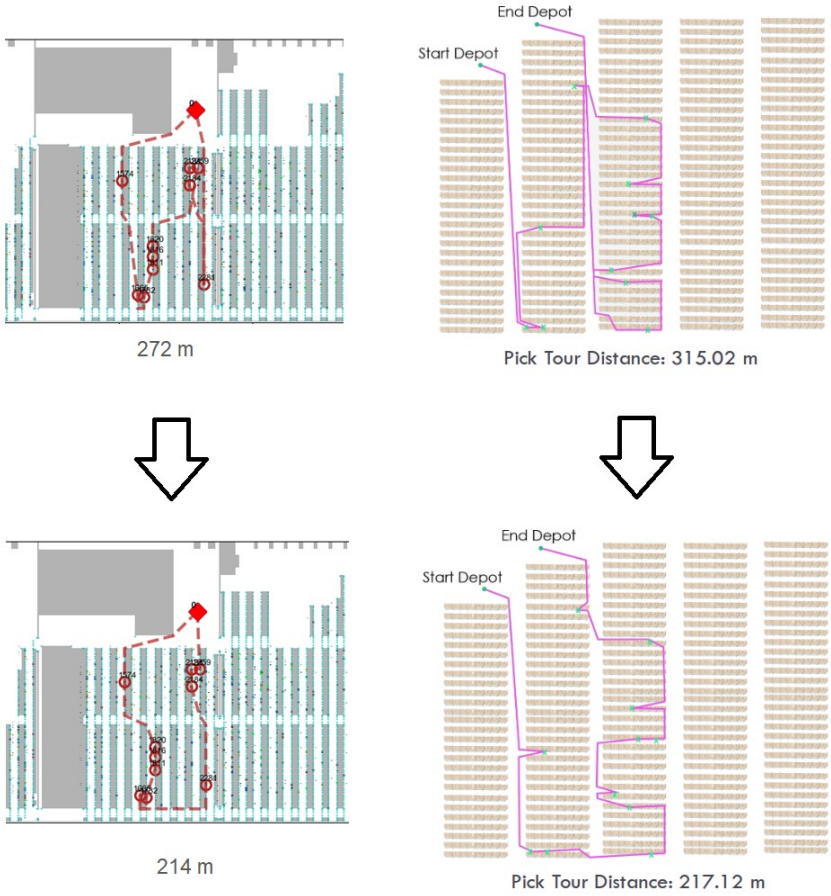


Figure 4: Example of TSP optimization in warehouses A and B.

As can be seen in the maps above both warehouses have several rows of racks with aisles in between. The aisles between racks are denoted *aisles* whereas the aisles between rows of racks are denoted *cross-aisles*. Warehouse A also has obstacles that are represented as gray polygons on the left above map. The majority of research on VRP's in warehouses are on layouts that have just one or a couple of cross aisles and no additional obstacles (Henn et al., 2012, Theys et al. 2010, Janse van Rensburg, 2019). In a preceding project both warehouses were digitized with a graph containing all node-node shortest distances computed through the Warshall – Floyd algorithm (Janse van Rensburg, 2019).

The vehicles in both warehouses A and B carry a set of boxes that are filled with box specific items from the racks. A vehicle route that carries $\mathbb{N}^+$ of these boxes is known as a *batch*, and selecting which box should go into which batch is known as *batching*. It can also be modeled as a so-called *Multi-Compartment* VRP. The box in both these warehouses is considered *unbreakable* i.e. all pick items belonging to a box must be picked in the same pick run. The effect of this hard constraint is significant since it gives less freedom of choice to generate new optimized pick runs. New pick runs can only be

generated by combining boxes rather than combining pick items within the boxes. The difference can be seen in the figure below:

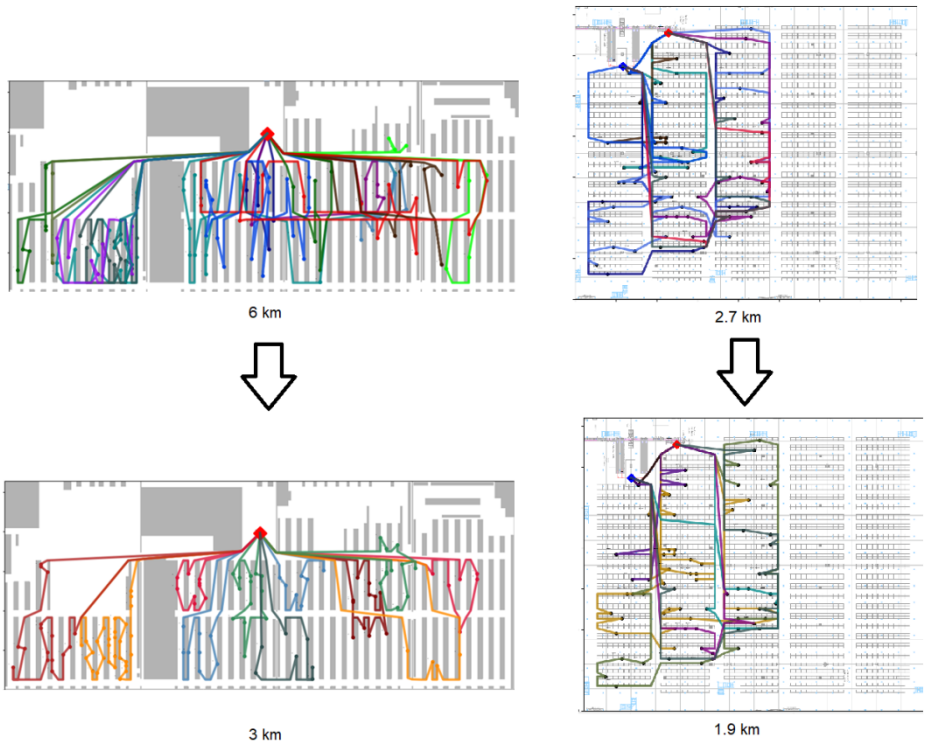


Figure 5: Example of batching optimization in warehouses A and B (a form of VRP). The colored lines represent pick runs, that contain one or several boxes that contain the pick items (which pick item belongs to which box is not shown). In the example on the left (A) new optimized pick runs were created out of the pick items within the original boxes (original boxes were "broken"). On the right (B) optimized pick runs were created out of original boxes (boxes were not "broken"). Clearly there is more optimization potential if original boxes are allowed to be broken.

Neither warehouse A nor B disclosed information about how they conduct batching until after the implementation part of this study was completed. This study was thus restricted to VRP optimization without box-logic (boxes were "broken"). Even after some info on this constraint was made available there was still missing information on why a certain box went into a certain batch. Often pick item deadlines have a large impact on how a batch can be constructed. Warehouse B use a methodology known as *deadline driven wave picking* to pick all items with certain deadlines within certain time-windows (e.g. item x must be picked before shipping time y). In the simulated data produced in this study pick items without any box belongingness were used.

In the below figure a flowchart shows components needed for a real deployed warehouse VRP optimizer that uses box-constraint logic (batching):

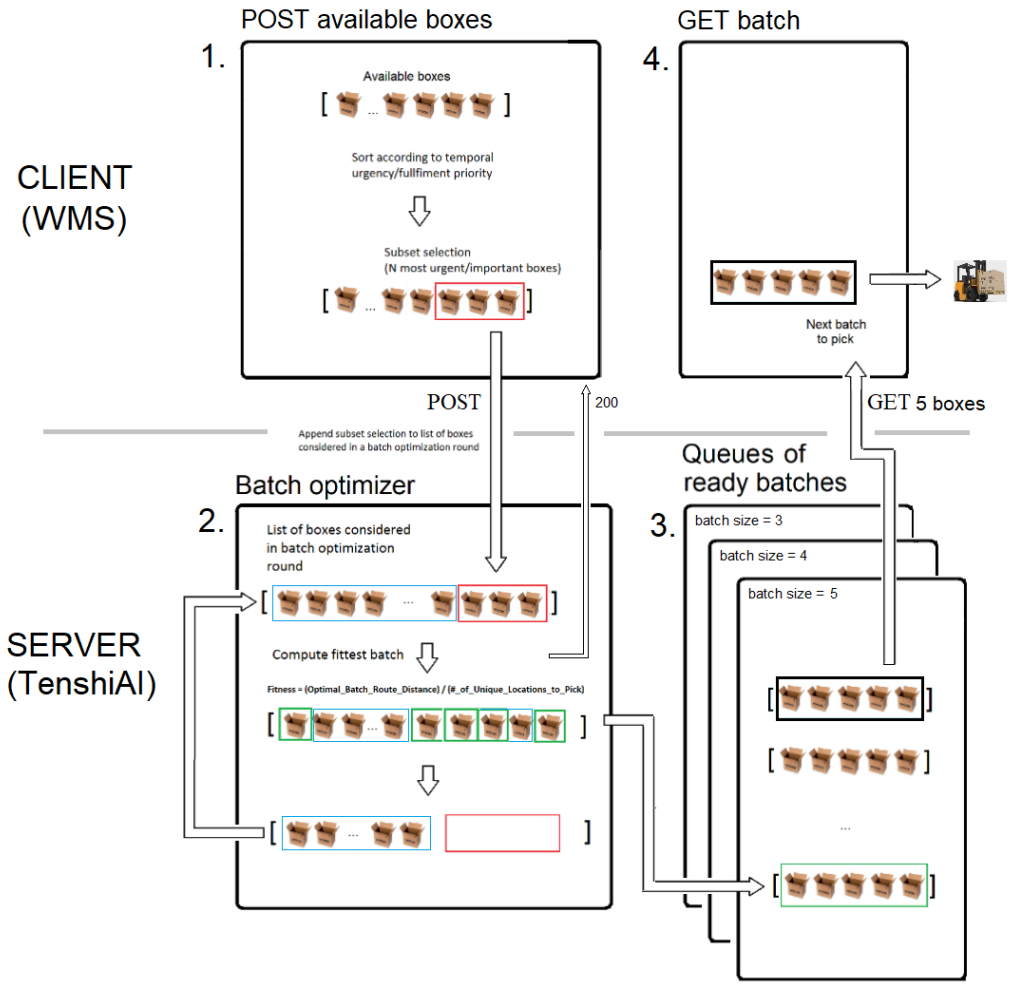


Figure 6: Continuous Batching (courtesy Sony-TenshiAI). The client (WMS) and the batch optimizer run algorithms to select which items are available for picking before any actual VRP optimization ("Compute fittest batch"). The VRP optimization in this whole study is covered completely within the "Compute fittest batch" step (for items within the "broken" boxes).

There are multiple levels of processes that the available datasets only partly reflect and the question is how useful an algorithm is that is only implemented on one of those levels. The objective or Key Performance Indicator (KPI) could be overall warehouse profitability and this is difficult to optimize because of the scope and unavailability of data. Minimization in the total pick time or maximization of *pick rate* (how many items can be picked within a certain time) would be easier to optimize, but this is also plagued by missing data (e.g. pick error, items that are out of stock, worker breaks, congestion and the effect of vehicle turn frequencies within pick runs). Total distance travelled is optimizable with the data but its frame of reference is on the contrary insufficient as it cannot provide all information necessary to dictate change in operations on a higher level.

For general VRP's possible objectives or Key Performance Indicators (KPI) are summarized as follows by Psaraftis et al. (2015):

Objectives/KPI's:

a) To minimize
- Route cost
- Route distance
- Travel time
- Total lateness
- Number of vehicles
- Cost of service plus penalty
- Customer dissatisfaction
- Makespan

b) To maximize
- Quality of service
- Profit

Figure 7: Possible objectives for general VRP's (Psaraftis et al., 2015).

Some of these (such as customer dissatisfaction) are not relevant in warehouse VRP's and the below sketch shows a simplified version with KPI's arranged hierarchically by how much information they need to be optimized:

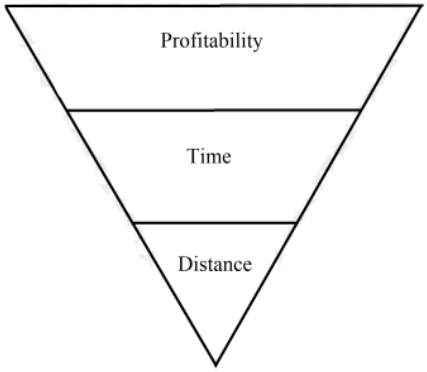


Figure 8: Sketch showing the amount of features needed to optimize different KPI's and subsequently the complexity that the models have to handle. To minimize time in a VRP minimal distance would probably be the most important feature but others, such as pick error, missing items and congestion, would also be needed. All of this information and more still would be needed to approach the profitability KPI.

The warehouse is thus an open environment where complex hierarchical decisions are made in a dynamic fashion, only part of which is logged in data. From time to time there are major changes to the WMS and its KPI's for long term business reasons. One example is introduction of autonomous ground vehicles (AGV's). Presently AGV's are mostly routed on pre-defined tracks in testing environments that contain human pickers and the men and machines may interfere with each other and affect each other's performance. After a trial run filled with unprecedented problems and opportunities the warehouse manager might have one idea on how well the trial went, the WMS provider another and the AGV company a third. Formulating and delineating a general optimization usecase in this complex environment is therefore difficult.

3.1. Thesis questions

This investigation is about developing and testing a DRL algorithm, *MCTS-CNN,* in a simulated warehouse VRP environment where the aim is to minimize distance:

1. *How can MCTS-CNN be used to minimize travel distance in simulated warehouse VRP's?*

2. *How can a CNN be pre-trained on simulated warehouse VRP data?*

3. What is the MCTS-CNN performance on simulated warehouse VRP data on the distance minimization KPI level?

Question 1 is covered in section 5 which provides the theory for using *MCTS-CNN* in the warehouse VRP domain. Questions 2 and 3 are answered in section 7 which provides experimental results from the implementation of *MCTS-CNN*. Question 3 is answered by comparing VRP solution distances and prediction computational times between *MCTS-CNN* and a reference model referred to as *Two-Phase*.

The experiments are carried out on with the following VRP set up:

1. Obstacle inclusive (racks).
2. Multiple vehicles.
3. Varying start and end locations.
4. Single visits to nodes.
5. No time-windows.
6. No boxes or box-calculations.
7. Capacities (in some experiments)
8. Traffic congestion (in some experiments)

One underlying purpose for attempting a DRL algorithm is *scalability* i.e. that it is relatively easy to scale up the algorithm when the problem complexity is grown by e.g. moving from the distance to the time minimization KPI. Current VRP benchmark datasets focus mainly on number of pick locations (often called *nodes* below) and this barely covers distance minimization in a warehouse with obstacles (unless one applies the digitization procedure by e.g. Janse van Rensburg (2019)), and even less time minimization. Whether the DRL algorithm implemented here can be expected to scale well is covered in a discussion section (8).

4.Review of solution techniques

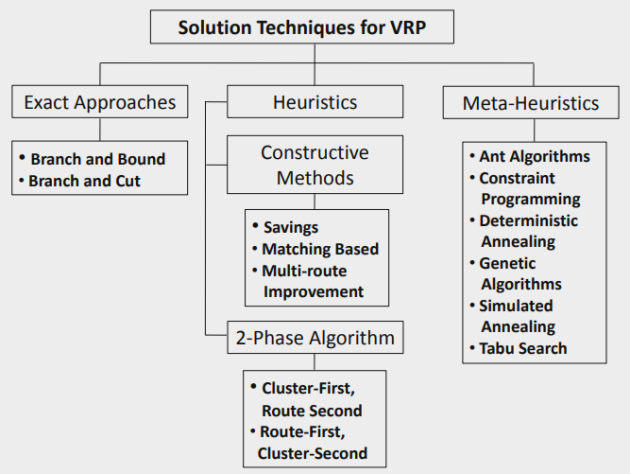


Figure 9: Some common VRP solution techniques (Altman, 2019).

4.1. Heuristics, meta-heuristics and constraint programming

Figure 9 shows some common distance minimization solution techniques on canonical VRP's without any racks or other obstacles. Exact approaches will not be covered here because only small scale VRP's (e.g. TSP) can be solved to exact optimality within a short amount of time and this investigation has more focus on scalability (Csiszar, 2007; Goodson, 2010; Arnold & Florian, 2017). Concerning constructive methods there are e.g. the *Savings* method, the *Matching* based method and *Multi-route improvement*. They all aim at establishing an initial candidate solution and then improving on it through various heuristical procedures such as k-opt swapping and pruning (Toffolo, 2018; Edelkamp, 2012; Cordone, 1999; Arnold & Florian., 2017). Concerning meta-heuristic methods there are e.g. *Simulated Annealing*, *Genetic Algorithm*, *Ant Colony Optimization*, *Particle Swarm Optimization* (Chiang & Russell, 1996; Thangiah, Nygard & Juell, 1991; Potvin & Bengio, 1996, Homberger & Gehring, 1999 respectively). These take inspiration from systems in biology/physics and often result in better search compared to constructive methods by e.g. allowing for temporary reduction of solution quality to avoid local minima. In the domain of constraint programming a LNS (Large Neighborhood Search) method has been proposed which extracts and then reinserts nodes in a constrained tree search (Shaw, 1998). There are also hybrid models that e.g.

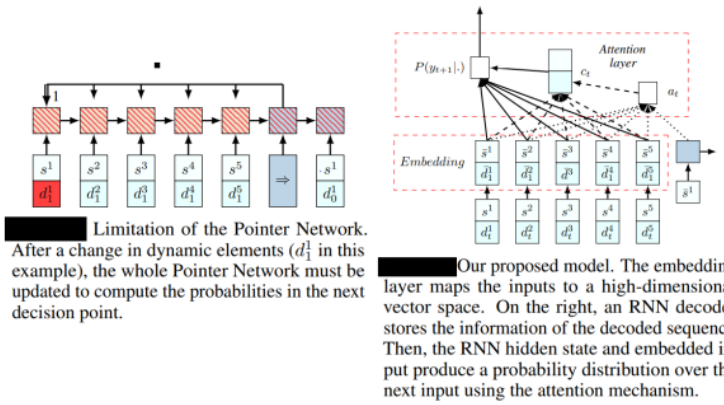
use a Genetic Algorithm and constraint techniques in unison (De Backer & Furnon, 1997).

If it is hard to define meaningful ways to search through the solution space a *Two-Phase* algorithm may be suitable (Csiszar, 2007). The VRP can e.g. first be clustered into TSP's that are then optimized. This is an appealing method for VRP's with the distance KPI because an exact linear programming solver, *Concorde*, can output minimal distance pick-runs for ~100 items in less than 1 second (Applegate et al., 2002, fitted to a warehouse context by Janse van Rensburg, 2019). If the data is first clustered with less than 100 items in each cluster (i.e. max capacity is set to 100) and then solved with *Concorde* it is possible to allocate significant time to unsupervised clustering before settling on a solution candidate.

Parallelization can be applied to many of the above methods. Threads can e.g. be assigned to parameter tuning and sweeps over the path finding or clustering and this can significantly reduce computational time (Crainic, 2007).

4.2. Deep Reinforcement Learning (DRL)

In the field of DRL there are some VRP applications with the Pointer Network (PN) and modifications to it (Vinyals et al. 2017, 2018; Nazari et al, 2018). PN is a type of RNN (Recurrent Neural Network) that can train on and output sequences of variable length; which is indispensable in the VRP case since route lengths vary. Vinyals et al. did not initially manage to produce consistent results due to PN's sensitivity to small changes in the sequences, something Nazari et al. fix by adding another layer of encoder/decoder attention mechanism whose logic is optimizable through reinforcement learning. This led to state of the art performance on CVRP's and (in combination with other techniques) on VRP like subtasks in game environments (Nazari et al., 2018 p. 4 – 5, 7-8; Vinyals et al., 2017, ICLR, 2019).



Limitation of the Pointer Network. After a change in dynamic elements (d_1^1 in this example), the whole Pointer Network must be updated to compute the probabilities in the next decision point.

Our proposed model. The embedding layer maps the inputs to a high-dimensional vector space. On the right, an RNN decoder stores the information of the decoded sequence. Then, the RNN hidden state and embedded input produce a probability distribution over the next input using the attention mechanism.

Figure 10: Nazari et.al.'s proposed improvement to the Pointer Network by adding an embedding layer that can disregard small changes to sequences.

Another approach with DRL is to first cut the sequences into snapshot transition states, train a Convolutional Neural Network (CNN) on these states and then use this in unison with Monte Carlo Tree Search (MCTS) (Hassabis & Silver et al., 2016, 2017; Mnih et.al., 2013; Clark, 2016). In a VRP setting the cutting of sequences into transition states is particularly suitable if the environment is dynamic and vehicles need to be rerouted relatively often (DVRP). Clark built a visual interface for a warehouse DVRP which shows how unstable through time a VRP environment can be when vehicles are affected by congestion:

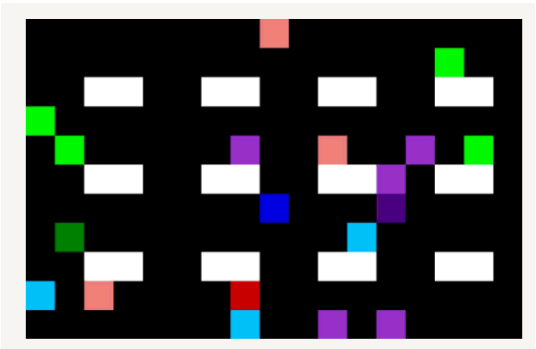


Figure 11: A snapshot or transition of Clarks MVRP simulated environment in a warehouse. The white rectangles are impassable racks, the lighter squares are pick nodes and the darker squares are other pickers that are also impassable. I.e. the dark green square is a picker that has to pick all the light green squares while at the same time avoiding racks and other moving pickers. The simulation makes a good case for the use of real time rather than a-priori solvers.

If a model is tasked with producing routes for the vehicles in the figure above we can see that, regardless of its quality, it may be obsolete in just a few time steps due to traffic congestion problems. Transition states that a CNN can train on could provide a scalable way in which to generalize over such scenarios. When scaling up to DVRP's this type of autonomous feature extraction is appealing and more features can easily be added as additional input layers. The general intuition is to move as much processing as possible

from post-training to pre-training in order to achieve more effective search post-training.

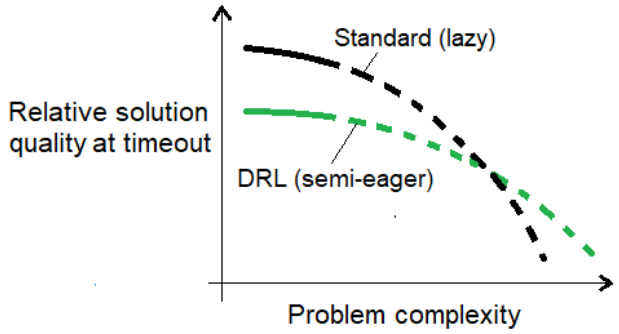


Figure 12: Sketch that shows the potential advantage of DRL over standard 'lazy' models. Lazy means that the model is pre-trained to a very small extent (e.g. by training to get the best hyperparameters) whereas semi-eager is pre-trained as much as possible (e.g. by pre-training a CNN and using it inside MCTS post-training). If DRL is successfully implemented it could be as strong as lazy models for less complex problems but it is sometimes less suitable in those contexts due to more difficult implementation.

5. Theory

The VRP is usually formulated as a vehicle or commodity flow or as a set partitioning problem (Dantzig et al., 1959; Toth et al., 2002; Munari et al., 2017 respectively. The latter can be found in Appendix). It is also possible to formulate it as a Markov Decision Process (MDP) (Thomas & White, 2004, 2007; Goodson et al., 2010, 2019; Psaraftis et al., 2015). Reinforcement Learning provides a toolset with which to learn "what to do" in the MDP (See Sutton & Barto, 2018 for background on what Reinforcement Learning is and how it is related to MDP's). In practice the MDP approach is most intuitively applied in DVRP scenarios. This is however not a necessity and an MDP formulation is equally valid for the general VRP. The MDP formulation in the next section covers a general VRP where $\mathbb{N}^+ = \{1, 2, \dots\}$ vehicles must visit a set of nodes once and where they can start and end at any node.

The heuristic, meta-heuristic and constraint programming approaches covered in the previous section can be applied to optimize the DVRP scenarios as well by dynamically updating the data-structures they take as input. How this can be done will not be covered in this theory section. As is often the case, there are multiple ways in which to provide solutions to a problem. The MDP/RL methodology was chosen because it is believed to allow for more theoretical sturdiness and scalability.

5.1. Markov Decision Process (MDP)

An MDP is a discrete time dynamic system model with the following components:

s_t - the state of the system at time t

a_t - the action taken at time t

r_t - the reward at time t

Let S be the set of possible states, A the set of possible actions and R the set of possible rewards, such that $s_t \in S$, $a_t \in A$, and $r_t \in R$. We assume that A is finite, such that we can choose one of a set of discrete actions at each time point, and that $R = \mathbb{R}$. The reward, obviously, is meant to represent the value to us (the system controller) of a particular state.

The system is assumed to be Markovian, in that the current state of the system depends only on the state of the system at the previous time slice and the action taken at the

previous time slice. Further the reward at a particular time is conditionally independent of all other variables given the state at that time. This means the system is completely specified by the *transition* and *emission* distributions:

T: P($s_{t+1}|s_t, a_t$)

E: P($r_t|s_t$)

The "solution" to an MDP is the discovery of an optimal policy, where a policy is a function π: S->A, such that it gives a particular action to perform in any particular state. The "goodness" of a policy is defined in terms of expected cumulative discounted rewards.

Consider the cumulative rewards of a sequence of states 0, … :

$$G = \sum_{i=0}^{\infty} R_i$$

The rewards in the trajectory can be discounted (such that future rewards are valued less) by parameter $\gamma \in [0,1]$:

$$G = \sum_{i=0}^{\infty} \gamma^i R_i$$

No discounting occurs when γ=1.

Let v_π: S -> $\mathbb{R}$, be a function that gives the expectation of G for sequences of rewards generated by acting on a policy $\pi \in \Pi$ from a given state $s \in S$:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G|S_0 = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_k |\, S_0 = s\right]$$

The distributions governing this expectation being, of course, E and T, under the assumption that the policy π governs actions. The optimal policy, π^* , is such that $v_{\pi^*}(s) \geq v_\pi(s)$ for all $s \in S$ and $\pi \in \Pi$. We term these *state-values*, and they have the Bellman equation:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G|S_0 = s] = \sum_{s'} P(s'|s, \pi(s))\,[R(s') + \gamma v_\pi(s')]$$

Finally, let q_π: $(S, A) \to \mathbb{R}$ be a function that gives the expectation of G for sequences of rewards generated by performing an action, a, in a given state $s \in S$ and *there-after* acting on a policy $\pi \in \Pi$:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G|S_0 = s, A_0 = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_k |\, S_0 = s, A_0 = a\right]$$

Again, the distributions governing this expectation being E and T, under the assumption that the policy π governs all actions *except* A_0. We term these *action-values* and they have the Bellman equation:

$$q_*(s,a) = \mathbb{E}_\pi[G|S_0 = s, A_0 = a] = \sum_{s'} P(s'|s,a)\left[R(s') + \gamma \max_a q_\pi(s',a)\right]$$

5.2. The warehouse VRP as an MDP

5.2.1. State, Actions, Rewards

As a vehicle travels through a VRP environment, pick locations/nodes go from being *unvisited* to *visited*. Each node visit can be defined as a point where a decision is made on which *unvisited* node a vehicle should be routed to next. As long as the environment is sequentially updated between each decision this can be made to hold true for $\mathbb{N}^+$ vehicles. For example, vehicle A is first routed to an *unvisited* node, the node is set to *visited*, then vehicle B is routed to an *unvisited* node, the node is set to *visited* etc. Each vehicle has a specific *end node* and when there are no more *unvisited* nodes left it is routed there.

State (S): A representation of the *warehouse environment* with *visited* and *unvisited* nodes, the *current visited* nodes for all vehicles and their corresponding *end nodes*, and the *current visited* node of a given vehicle. Let S_t denote S at a discrete time-step when the given vehicle on a *current visited* node requests a next *unvisited* node to visit.

The below figure shows an example of a state S_t where racks are included in the *warehouse environment*:

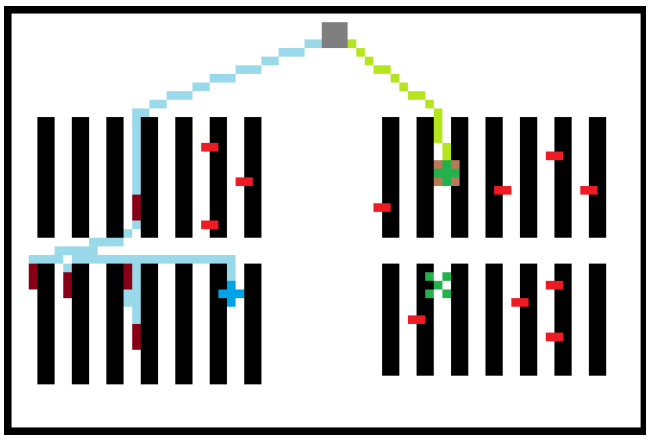


Figure 13: This is a 2D representation of a warehouse VRP environment at a certain time step S_t (some features will be covered by others). The grey square is the start and end location of all vehicles. The black rectangles are racks. The red 2x1 pixels are unvisited nodes. The brown 1x3 pixels are visited nodes. The blue and green crosses are the current locations of all vehicles (standing on "current visited" nodes). The green cross with a brown square background is the given vehicle that requests an unvisited node from S_t. The lower green cross is an example of an unvisited node that is provided for the given vehicle.

The above figure resembles a figure by Goodson et al. (2019) that shows the same basic components but for one vehicle, a square graph map without racks and with different terminology:

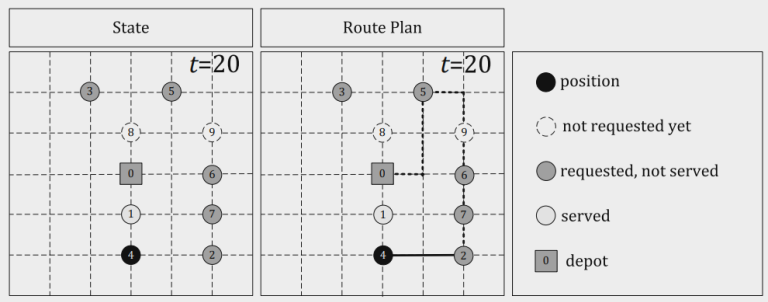


Figure 14: The basic components needed to model a VRP as an MDP are shown in the right box Goodson et al. (2019).

The above two figures show the VRP environment as raster 2D images and if 2D raster images are to be used as input to an algorithm they are usually transformed into binary layers, like so:

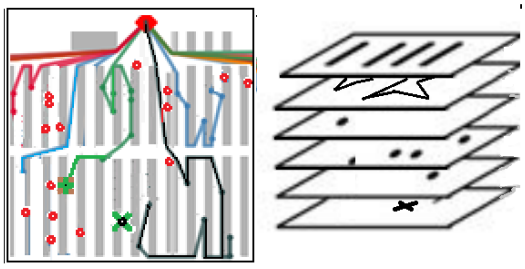


Figure 15: Example of how a VRP state can be represented using binary layers.

Action (A): A finite set of actions of selecting an *unvisited* node or the *end node* from S for a given vehicle, where A_t is the action selected for S_t and where the *end node* is selected only when there are no *unvisited* nodes left. After A_t the *unvisited* node becomes a *visited* node and it also becomes the *current visited* node of the given vehicle.

Reward (R): A finite set of rewards for A, where it zero for all states except for the terminal state T where it is a function of the total distance travelled D:

$$R_t = \begin{cases} 0, & 0 < t < T \\ f^{-1}(D), & t = T \end{cases}$$

5.2.2. Motivation for the action and reward formulations

The reward formulation may seem counterintuitive because rewards could be defined as node-node distances i.e. with a non-zero value for each t, and then summed together in the return G. However, a VRP cannot be solved by constantly selecting the nearest node for a vehicle in the MDP. It is, for example, clear that a TSP algorithm that does this performs worse than more elaborate algorithms (e.g. *Greedy* performs worse than *Simulated Annealing*, Janse van Rensburg, 2019). Removing intermediate rewards removes the issue of estimating how and to what extent they can be useful. Generating good such estimates for a complex VRP-type requires heuristics especially if distance is changed to some more multifaceted VRP KPI such as time (see section 3 on why). Even without intermediate rewards heuristics are still needed in many places in any strong performing VRP algorithm. The question is where they should best be placed (see sections 5.4 and 5.5 for further discussion on choice and placement of heuristics for the algorithm developed).

A perhaps simpler explanation for the removal of intermediate rewards is that they are not used by the *MCTS-CNN* implementation that is used as reference for this study (following the same reasoning as in Hassabis et al., 2016; 2017). They are replaced by values and/or probabilities that fulfill a similar functionality in a different manner.

The formulation for action could also be debated, especially since the experimentation part of this study ended up using a different formulation (section 7). The current action formulation requires all unvisited nodes to have a corresponding action-value and this fits well with the above VRP MDP formulation and allows for more theoretical rigor, but comes with some practical issues (explained further in section 5.3.3).

5.2.3. A general warehouse VRP Markov-model

The dependence of R_t and S_t on S_{t-1} and A_{t-1} (the Markov property) can be achieved if the VRP is forced to follow this time-step sequence:

$$[S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots, S_{T-1}, A_{T-1}, R_T]$$

where T denotes the total number of nodes (pick locations), where S_{T-1} is the state when there are no *unvisited* nodes left, where A_{T-1} is the selection of an *end node* for the last time and where R_T is the reward received for A_{T-1} (function of total distance travelled). R_t is always 0 using the above formulation, A_t only concerns a given vehicle and S_t concerns a whole VRP environment but with a feature that specifies the starting location for a given vehicle.

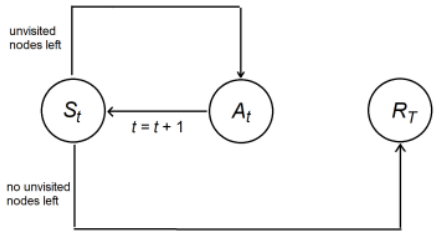


Figure 16: State diagram showing the relationship between S, A and R after the removal of intermediate rewards.

The same *S, A* and *R* definitions can be used to formulate a TSP since TSP is a subset of VRP. In a VRP S_t includes visited and current nodes of $\mathbb{N}^+$ vehicles and in a TSP S_t includes visited nodes and the current node for one vehicle.

The link between VRP's and TSP's that the above MDP formulation provides is important because it allows working in a TSP environment before scaling up to a VRP environment. For TSP's it is easy to obtain optimal solutions that can be used to evaluate the performance of a candidate algorithm.

5.3. Learning policies

5.3.1. Dynamic Programming

Iterative policy evaluation is an algorithm that evaluates how good a given policy is by going through all states iteratively and updating expected returns for state-values with the following pseudo-code (Sutton & Barto, 2018):

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

The inner loop iterates through all states and updates the expected returns, Δ is a measure of how much expected returns are changing and θ is an accuracy parameter that terminates the outer loop on convergence when Δ is small enough. $p(s', r \mid s, a)$ is the probability of transitioning to state s' and receiving reward r given state s and action a ($\pi(s)$).

Policy improvement is an algorithm that compares outputs from and old policy π and a new one π' and deterministically picks the one with higher expected returns over action-values:

$$\pi'(s) \doteq \arg\max_a q_\pi(s, a)$$

This is also called a *greedy* policy and can be applied to all $s \in S$ using the following pseudo-code (Sutton & Barto, 2018):

Policy Improvement
For each $s \in \mathcal{S}$:
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

where $\arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')] = \arg\max_a q_\pi(s, a)$.

Iterative Policy evaluation and *Policy improvement* can be combined in a single algorithm called *Policy Iteration* (Sutton & Barto, 2018):

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

After 2 is completed all state-values have been updated according to an old policy and then 3 probes the state-space for an action that was different following the old policy compared to the greedy policy. If this is the case policy-stable is false and the algorithm returns to 2.

Policy iteration can also be expressed with the following sequence:

$$\pi_0 \overset{E}{\rightarrow} q_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} q_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} \dots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} q_*$$

where *E* denotes policy Evaluation and *I* denotes policy Improvement. It is an exhaustive breadth first algorithm as all possible actions from all states are visited multiple times. While this allows for convergence on the optimal action-value function through the Bellman Optimality equation, it requires sweeps through the full state space. The number of computational operations to implement this is often prohibitive in practice and in such settings heuristic policy iteration can either be maintained while constrained and sub-optimal, or replaced by other methods. Techniques such as *Policy Iteration* and *Policy Improvement* that demand exhaustive visits to an MDP state space (in theory) fall within the *Dynamic Programming* (DP) paradigm.

5.3.2. Exploitation and Exploration

Instead of *computing every single* action-value in a state space, giving the *true expectation*, it is also possible to *estimate* them using *Monte Carlo* sampling methods, giving an *empirical mean* (section 5.3.3). This allows the number of computational operations to be decreased but it also requires decisions concerning which parts of the

state-space that deserve sampling. If the samples are finite optimality cannot be guaranteed unless there are ways to prove that the samples themselves include the optimal state/action-values. If this is infeasible it becomes necessary to balance between increasing sample size - to increase the probability of strong action-values being included, or decreasing it – to reduce number of operations. This is the exploration/exploitation tradeoff/dilemma: Either a smaller section of the state-space can be exploited, or a larger section of the state-space can be explored. The former has better chances of converging before a pre-set number of operations, whereas the latter has lower chances of converging but has a higher chance of finding a better policy.

One traditional way to resolve the exploration/exploitation tradeoff is by adding a noise parameter, ε, to the greedy policy, giving the so called *ε-greedy* algorithm. Actions are there mostly selected greedily but with probability ε uniform random actions are selected instead. Setting a high ε parameter will thus lead to more exploration of the state-space and more variance whereas a lower one leads to more exploitation and bias. Extra parameters can be added to *ε-greedy* such as a *decay* schedule that gives high exploration in the beginning of the algorithm and high exploitation at the end of it. The intuition is that initial exploration is needed to increase the chances of finding strong action-values that can then be exploited to converge on a good policy.

Another way to resolve the exploration/exploitation tradeoff is by *Upper Confidence Bounds* (UCB). UCB is linked to the concept of *optimism in the face of uncertainty* which focuses search on states where the action-value is uncertain (Salakhutdinov, 2019). The uncertainty can be estimated as an upper confidence $U(s, a)$ over state visited counts $N(s, a)$ with the following intuition:

$$\text{Small } N(s,a) \Rightarrow \text{Large } U(s,a)$$

$$\text{Large } N(s,a) \Rightarrow \text{Small } U(s,a)$$

i.e. when action-values have been sampled and updated many times e.g. through random samples from the state-space, they are more likely to be closer to the true action-values and do not need further exploration. The upper confidence term can thus be included in the selection of action-values so that less visited, more uncertain action-values are selected within the upper confidence bound:

$$q(s,a) \leq Q(s,a) + U(s,a)$$

where q denotes the action-value selected and Q the Monte-Carlo empirical mean over previously sampled action-values (section 5.3.3). It is now necessary to define U so that it shrinks when states become more visited leading to increased exploitation around the distribution mean and less around its tails. In the example figure below this is represented by the width of probability distributions built up by 3 node Q value means

on the x-axis and the probability of selecting that node on the y-axis. A distribution with more width is more uncertain and hence has an upper confidence further to the right:

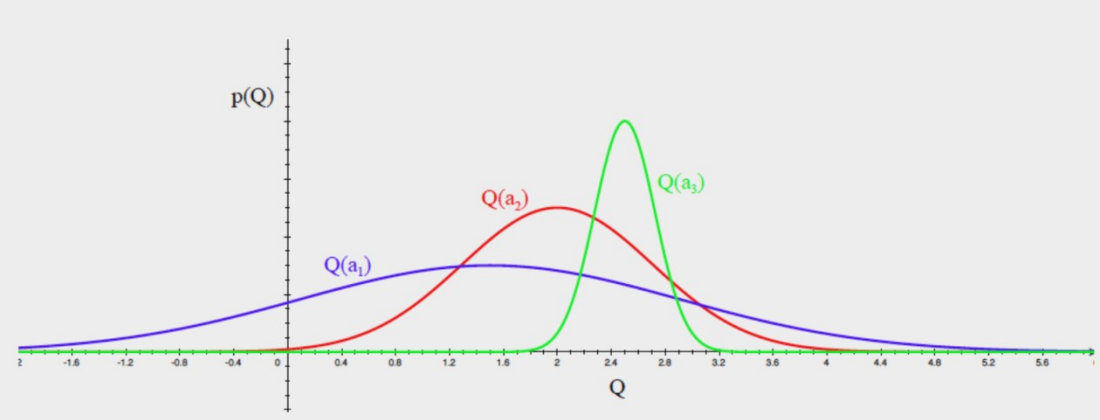


Figure 17: There are three possible actions to explore further and the one that is most "uncertain" is the one with the most area under its upper tail, i.e. the blue one (Salakhutdinov, 2019). UCB theory deems this to be the state most worthwhile to explore further.

For non-MDP problems such as multi-arm bandits U can be exactly computed using *Hoeffding's inequality theorem* (Sutton & Barto, 2018):

$$P[q(a) > Q_t(a) + U_t(a)] \leq e^{-2N(a)U_t(a)^2}$$

which gives the probability of being outside the upper confidence bound for any distribution and where $N(a)$ denotes the number of times an action has been sampled. Solving for $U_t(a)$ in $P(a) = e^{-2N(a)U_t(a)^2}$ gives:

$$U_t(a) = \sqrt{\frac{-\log P(a)}{2N_t(a)}}$$

Hoeffding's inequality has also been successfully applied to MDP's heuristically e.g. with the following version called *Upper Confidence Tree* (UCT) (Hassabis et al., 2017; builds on Chaslot et al., 2008):

$$U(s,a) = P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

where $P(s, a)$ is an MDP prior policy probability (making exploration *model-based*, more on this in section 5.5) and where b are all actions that are not a. If $P(s, a)$ outputs uniform random probabilities so that all transition-states are visited equally often we can see that the numerator will grow slower than the denominator as N grows, leading to a decaying U.

5.3.3. Learning Policies II (Monte Carlo sampling with Upper Confidence)

The upper confidence U described in the previous section estimates how worthwhile it is to explore states and $P(s, a)$ in the $U(s, a)$ equation can be seen as an *exploration policy* probability. In the previous section it was stated that the action selection in upper confidence based exploration should be within the sum of the mean action-value and U: $q(s,a) \leq Q(s,a) + U(s,a)$ and this section will produce and motivate a Monte-Carlo sample based action-selection equation.

MC methods can be used to formulate the same basic sequence as the one for actions-values with DP policy iteration to move toward q_* (section 5.3.1):

$$\pi_0 \overset{E}{\rightarrow} q_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} q_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} \dots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} q_*$$

but by sampling some states instead of all states to evaluate (E) and improve (I) policies such that $Q(s,a) \rightarrow q_*(s,a)$. This still converges on an optimal policy as in DP if states are assumed to be evaluated and improved an infinite amount of times by the *greedy in the limit with infinite exploration* theorem (GLIE) which requires infinite state visits N:

$$\lim_{t \to \infty} N(s,a) = \infty$$

and infinite action samples k:

$$\lim_{k \to \infty} \pi_k(a|s) = 1(a = \arg\max_{a' \in A} Q_k(s,a'))$$

where k signifies the number of times a state is sampled, also known as number of *rollouts*, and where 1 indicates a completely greedy policy probability after infinite samples i.e. $\lim_{k \to \infty} \pi_k(a|s) = 1 \; when \; a = \arg\max_{a' \in A} Q_k(s,a')$. The GLIE theory is based on central limit proofs on MDP's (Singh et al., 2000; Sasikumar, 2017; Chopin, 2004). It can be used to improve policies greedily: $\pi'(s) \doteq \arg\max_a q_\pi(s,a)$. In a practical implementation where complete GLIE is not possible k can heuristically be set as a tradeoff between exploration/exploitation. Low k increases exploitation and lowers computational operations whereas high k increases computational operations and increases chances of finding a stronger policy.

MC methods evaluate and improve policies by sampling the MDP from a root state s_t until it terminates $\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, \dots, S_T^k\}_{k=1}^K$ where the superscript shows k rollout and the subscript shows t time-step. When S_T^k is reached returns G are "backed up" or "backpropagated" to s_t. This is known as *deep sample backups* as compared to *shallow full backups* in DP (similar to *depth-first* versus *breadth-first* theory). There are several

variants of MC backpropagation and the following is a version that updates stored state-values with incremental means (Sutton & Barto, 2018):

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

where $V(S_t)$ denotes state-value and G_t the return extracted at a leaf node ($G = \sum_{i=0}^{\infty} \gamma^i R_i$). The same equation can be formulated for action-values:

$$Q(S_t, A_t) = Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$$

It is now possible to set up a sampling policy that uses this empirical mean and the UCT equation from the previous section. The mean of the incrementally updated action-values can be summarized as:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u \xrightarrow{P} q_\pi(s, a)$$

where 1 and $\xrightarrow{P}$ indicate probabilistic convergence by GLIE (if $N(s, a)$ tends to infinity for a node its action-value will tend to 1, assuming 1 is the maximum action-value). Returns G are aggregated in the inner sum the same way as defined in section 5.1 (sum of rewards without discounting). The aggregation is conducted K number of times in the outer sum. K can also be replaced by a countdown timer that allows $\mathbb{N}^+$ rollouts until timeout (where it would be infinite time for GLIE). At each state visit in this process the action-value is incrementally updated as the $1/N$ mean.

The above equation is then combined with upper confidence. At each time-step the state with the maximum sum between action-value Q and upper confidence is selected (Sutton & Barto, 2018):

$$\pi(S_t) = A_t = \arg\max_{a \in A}(Q(S_t, a) + cU(S_t, a))$$

For U the MDP UCT equation is inserted (Hassabis et al., 2017):

$$A_t = \arg\max_{a \in A}\left(Q(S_t, a) + cP(S_t, a)\frac{\sqrt{\sum_b N(S_t, b)}}{1 + N(S_t, a)}\right)$$

c is a scaling parameter that determines the level of exploration. $Q(S_t, a)$ denotes *exploitation* as an MCTS estimate of an action-value and if all states are sampled equally often it should converge on the true optimal action-value according to GLIE (heuristically for MDP's). $U(S_t, a)$ denotes *exploration* and should be relatively high in the beginning of sampling and decay through time. The ratio difference between the

visited counts make sure this happens (by pushing the bars left in Figure 17) so the $P(S_t, a)$ prior is not strictly necessary for this and can replaced by e.g. $P(S_t)$. A function approximator can also be used instead of $P(S_t, a)$ or $P(S_t)$ to speed up exploration at the cost of more bias (section 5.5).

5.4. Monte Carlo Tree Search (MCTS)

The previous section shows how action-values in an MDP can be sampled and updated while maintaining strong convergence properties through GLIE exploitation and UCT exploration. In this section the components are combined and built on to provide a full algorithm that can be used to evaluate and improve policies similar to policy iteration (Sutton & Barto, 2018; Coulom, 2006; Chaslot et al., 2008; Gelly et al., 2006, Hassabis et al., 2016, 2017).

MCTS includes several steps and there are several possible implementations that have evolved over the years. Algorithm 1 shows an older version by Gelly et al. (2006) applied for a two-player game:

```
 1:  function playOneSequence(rootNode);
 2:    node[0] := rootNode; i = 0;
 3:    while(node[i] is not leaf) do
 4:      node[i+1] := descendByUCB1(node[i]);
 5:      i := i + 1;
 6:    end while ;
 7:    updateValue(node, -node[i].value);
 8:  end function;

 9:  function descendByUCB1(node)
10:    nb := 0;
11:    for i := 0 to node.childNode.size() - 1 do
12:      nb := nb + node.childNode[i].nb;
13:    end for;
14:    for i := 0 to node.childNode.size() - 1 do
15:      if node.childNode[i].nb = 0
           do v[i] := ∞;
16:      else v[i] := -node.childNode[i].value
         /node.childNode[i].nb
         +sqrt(2*log(nb)/(node.childNode[i].nb)
17:      end if;
18:    end for;
19:    index := argmax(v[j]);
20:    return node.childNode[index];
21:  end function;

22:  function updateValue(node,value)
23:    for i := node.size()-2 to 0 do
24:      node[i].value := node[i].value + value;
25:      node[i].nb := node[i].nb + 1;
26:      value := -value;
27:    end for;
28:  end function;
```

Algorithm 1: MCTS version that uses a UCB formula for exploration and sets infinite values for any unvisited child nodes. Pseudocode by Gelly et al., (2006).

Algorithm 2 below shows a newer version by Fernandes (2016), which in turned is based on Chaslot et al., (2008). This version serves as the MCTS reference in this study but modifications to it were made since both Chaslot et al.'s and Fernandes versions are for two player games. Unfortunately pseudo-code written in this study currently falls under a non-disclosure agreement and cannot be shown. Another simpler version is also covered but it is weaker and is only shown to motivate why some of the more elaborate components of *Version 1* are needed. The four key ingredients in Chaslot et al.'s MCTS version are *Selection, Expansion, Simulation* and *Backpropagation* and can be used after *Initialization* in the following manner:

5.4.1. Version 1: MCTS inspired by Chaslot et al., (2008).

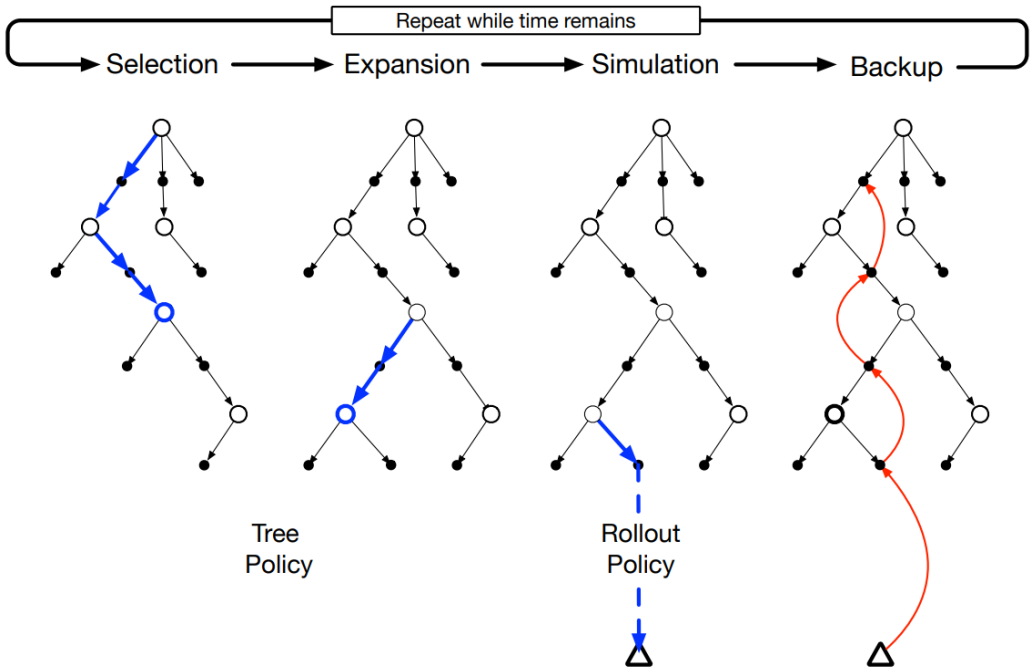


Figure 18: Monte Carlo Tree Search (from Chaslot et al., 2008).

Walkthrough of MCTS using an example and pseudocode:

1. *Initialization*: A tree structure is initialized. A node structure is initialized that contains visited counts and action-values (both as zeros) and a pointer to a state. Note that *node* is here used in the context of a *tree structure* and is hence different to a *VRP node*/pick location. The tree is populated with a *root node* $S_{0,0}$, where the first subscript indicates depth and the second a unique child node index. A *current node pointer* is initialized to point at $S_{0,0}$. A *Tree Policy* is initialized which uses the action-selection equation from the previous section ($A_t = \arg\max_{a \in A}(Q(S_t, a) + cU(S_t, a))$ to select child nodes. A *Simulation Policy* is initialized, which selects a possible next action by uniform random $\pi(s|\cdot)$. This produces random uniform $P(S_t, a)$ priors (inside $U(S_t, a)$). A *countdown timer* is initialized with a pre-set value of how much time is allowed for search before a step must be taken in the MDP (i.e. used instead of pre-set number of K rollouts).

2. *Selection*: The *current node* $S_{0,0}$ is queried for a child node. There is no child node so the algorithm skips *Selection* and proceeds to *Expansion* in 3.

3. *Expansion*: A child node to the *current node* $S_{0,0}$ is created, $S_{1,0}$. It is created using the *Tree Policy* over possible child nodes/next states. In this very first iteration it will be random uniform (solely determined by $P(S_{0,0}, a)$).

4. *Simulation*: A rollout is launched from $S_{1,0}$. It selects next nodes using the *Rollout Policy* until it reaches a terminal state.

5. *Backpropagation*: A return $G_{1,0}$ is extracted from the terminal state. This return is then used to update the action-values in $S_{1,0}$ and $S_{0,0}$ by incremental means ($Q(S_t, A_t) = Q(S_t, A_t) + (1/N(S_t, A_t))\ (G_t - Q(S_t, A_t))$. The visit counts in $S_{1,0}$ and $S_{0,0}$ are also incremented by 1. The *current node* is reset to $S_{0,0}$ (in this case it was already there).

6. *Selection*: The *current node* $S_{0,0}$ is queried for a child node. There is now 1 child node, $S_{1,0}$. This node can now be selected, but it can also not be selected. The probability that it is selected is governed by the just updated action-value $Q(S_{1,0}, A_{1,0})$, but also by the exploratory upper confidence $U(S_{1,0}, A_{1,0})$ that includes the random prior $P(S_{1,0}, A_{1,0})$ and visited counts $N(S_{1,0}, A_{1,0})$. If it is selected the *current node* is set as $S_{1,0}$ and the algorithm tries another *Selection* for $S_{1,0}$ -> S_2. If it is not selected the algorithm goes back to *Expansion* in 3. Assume it is selected.

7. *Selection*: The *current node* $S_{1,0}$ is queried for a child node. There is no child node so the algorithm proceeds to *Expansion* in 8.

8. *Expansion*: A child node to $S_{1,0}$ is created, $S_{2,0}$. In this second iteration it will also be random uniform as in 3. since there are no other sibling nodes at this depth.

9. *Simulation*: See 4. but with $S_{2,0}$ instead of $S_{1,0}$.

10. *Backpropagation*: See 5. but with $S_{2,0}$ added to the updates. The *current node* is reset to $S_{0,0}$.

11. *Selection*: See 6. Assume it is not selected this time.

12. *Expansion*: A new child node to $S_{0,0}$, $S_{1,1}$, is created.

13. *Simulation*: See 4. but with $S_{1,1}$.

14. *Backpropagation*: See 5. but with $S_{1,1}$ instead of $S_{1,0}$.

etc. The algorithm continues until the countdown timer runs out when a step must be taken. When that happens the *root node* S_0 is changed to its child with the highest action-value, which also becomes the *current node*. The timer is reset.

Selection, *Expansion*, *Simulation* and *Backpropagation* can thus be summarized by:

i. *Selection*: Selection of an existing child node in the search tree.

ii. *Expansion*: Creation of a new child node.

iii. *Simulation*: Rolling out a trajectory from the node in ii to a terminal node.

iv. *Backpropagation*: Extraction of returns at leafs that are used to update action-values in the existing nodes in the tree.

Note that there is no explicit demand for the terminal state T to ever be reached by *Selection* and *Expansion* using the action-selection summary equation (from section 5.3.3):

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u \overset{P}{\rightarrow} q_\pi(s,a)$$

This may seem counterintuitive because T denotes "terminal state", but *Selection* and *Expansion* in MCTS never themselves explicitly have to reach the terminal state but instead use returns computed by *Simulation*. T in regard to this action-selection equation can better be thought of as the scope from which returns *can* be computed rather than *must* be computed.

```
1: function ISMCTS(IS_0, n)
2:     create a single-node tree with root v_0 corresponding to IS_0
3:     for n iterations do
4:         d_0 ← randomly choose determinization from IS_0
5:         (v, d) ← SELECT(v_0, d_0)
6:         if u(v, d) ≠ ∅ then
7:             (v, d) ← EXPAND(v, d)
8:         Δ ← SIMULATE(d)
9:         BACKPROPAGATE(r, v)
10:    return a_c where c ∈ arg max_{c∈c(v_0)} N(c)
11:
12: function SELECT(v, d)
13:    while d is nonterminal and u(v, d) = ∅ do
14:        for all v' ∈ c(v, d) do
15:            N'(v') ← N'(v') + 1
16:        v ← BESTCHILD(v, d, c)
17:        d ← f(d, a_v)
18:    return (v, d)
19:
20: function BESTCHILD(v, d, c)
21:    return arg max_{v'∈c(v,d)} ( Q(v')/N(v') + c sqrt( 2 ln N'(v') / N(v') ) )
22:
23: function EXPAND(v, d)
24:    choose a ∈ u(v, d) uniformly at random
25:    add a child v' to v
26:    a_{v'} ← a
27:    d ← f(d, a)
28:    return (v', d)
29:
30: function SIMULATE(d)
31:    while d is nonterminal do
32:        choose a ∈ A(d) uniformly at random
33:        d ← f(d, a)
34:    return reward for state d
35:
36: function BACKPROPAGATE(v, Δ)
37:    while v is not null do
38:        N(v) ← N(v) + 1
39:        Q(v) ← Q(v) + Δ(v)
40:        v ← parent of v
```

Algorithm 2: An MCTS version called ISMCTS with Selection, Expansion, Simulation and Backpropagation. IS stands for Information Set and basically means that the algorithm is set to handle scenarios with hidden information. Determinization means that there is a mapping from some stochastic/estimated value (from the hidden information) to a heuristically defined one. Pseudocode by Fernandes (2016).

5.4.2. Version 2: Simpler but weaker (on a VRP example)

A different version can be implemented using the same four MCTS basic components but with a simpler *Expansion* step. This version is exemplified on 2 vehicle routing

problems, starting with a TSP with 3 pick location nodes and 1 depot node. This version is only used here to motivate why the more elaborate *Expansion* step in *Version 1* is needed.

The root node in *Version 2* also starts as S_0 and here all 3 child nodes are automatically expanded instead of just one of them. The action-values are denoted V. A simulation is then launched from each of these child nodes and in this version all the simulations are also added to the tree:

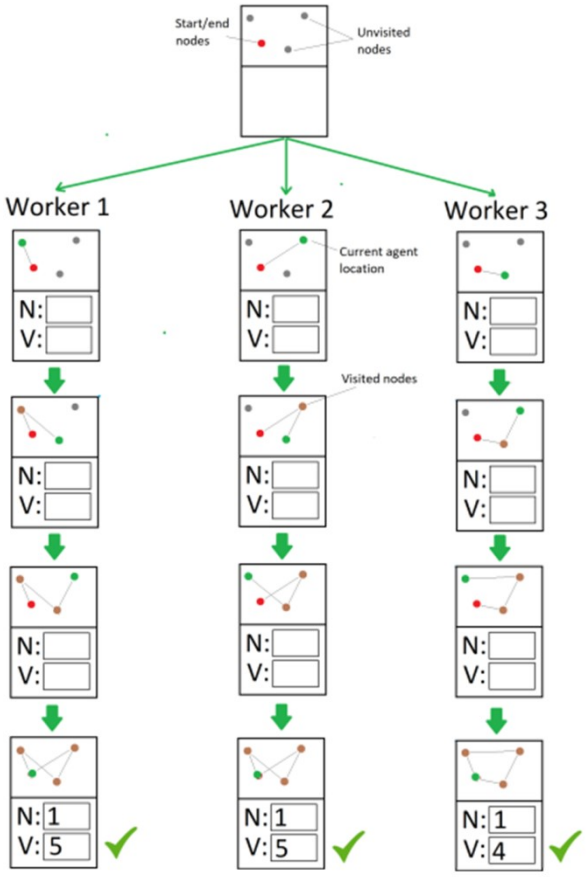


Figure 19: Sketch of how a 4 node TSP would look after a first volley of Version 2 MCTS asynchronous rollouts (this is why the term Worker is used). The red circle is the start/end node, the green circle is the current location of the picker, the brown circles are visited nodes and gray circles are unvisited nodes.

As the pick run is completed at the bottom the visited counter (N) is incremented and the return is calculated as total distance travelled. N and V are then sent upstream to update all the nodes until the root node is reached:

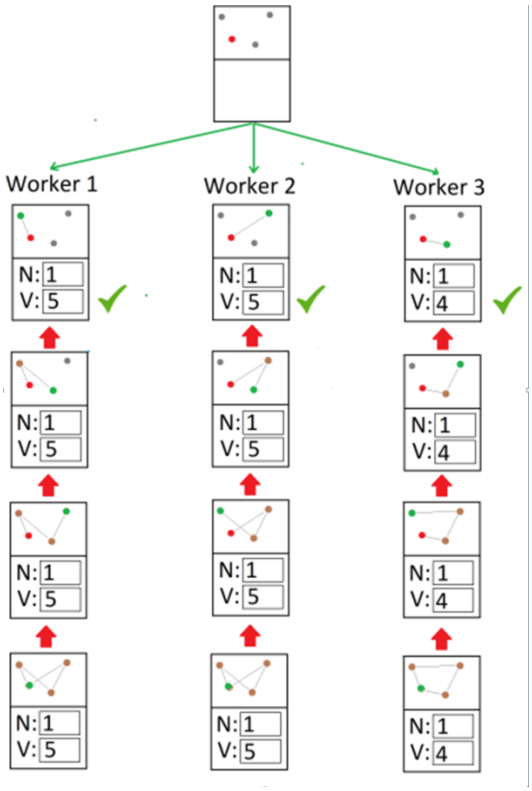


Figure 20: The first backpropagation of values upstream to the top states.

A critical disadvantage of *Version 2* compared to *1* is that many of the states visited by simulations are not going to be useful and are not worth spending time on updating and storing in memory. *Version 1* separates between these two kinds of states by having different procedures for *in-tree*, where action-values and visited counts are stored, and *out-of-tree*, where they are not.

To showcase this problem we can see that it may be computationally expensive not just to update all nodes but to even expand all possible child nodes from the root node. A VRP with 15 pick nodes and 1 depot, 3 racks and 3 pickers would e.g. look like the following:

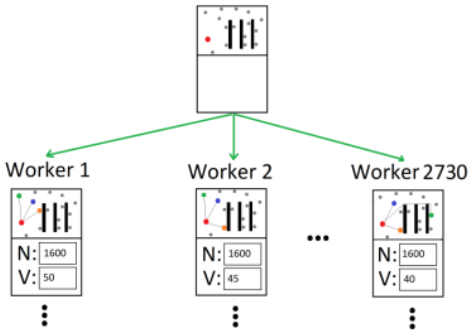


Figure 21: The root its child nodes of a VRP with 15 nodes, racks, and 3 pickers (green, blue, orange). To get values on all first moves 2730 workers would be required.

If the addition of racks is disregarded 3 vehicles and 15 pick locations requires 15x14x13 = 2730 first possible actions. This not only exemplifies the need for strong subsampling mechanisms as provided in *Version 1*, but also how quickly the VRP space explodes. *Version 1* is from now on the MCTS version referred to.

5.4.3. MCTS using different equations for *Selection* and *Expansion*

MCTS where the action-selection formula in section 5.3.3 is used for both *Selection* and *Expansion* guarantees slow convergence on optimal action-values if the $P(S_t, a)$ prior inside $U(S_t, a)$ is random uniform. In practice it may be better to use a simpler formula for *Selection* in case faster sub-optimal convergence is sought and $U(S_t, a)$ cannot be guaranteed to have decayed enough before a pre-set number of backpropagations. One example that only uses node visited counts (Hassabis et al., 2017):

$$A_t = \arg\max_{a \in A} \frac{\sqrt[\tau]{N(S_t, a)}}{\sum_b \sqrt[\tau]{N(S_t, b)}}$$

where a and b are the same as in the UCT component and where τ is a temperature parameter that dictates level of exploration with higher value leading to more exploration and vice versa. Convergence on a sub-optimal policy hence gives the opportunity to have *Selection* itself be exploratory (to the degree of τ). This makes more sense in specific domains where it is convenient to be able to compare capability of search policies quickly (which is made possible by the faster *Selection* convergence). One example is when the problem environment is an adversarial game where the return is defined as winning or losing against another policy. An in depth discussion of what this means and how non-game environments like the VRP differs from this can be found in appendix (section 9.2).

5.4.4. MCTS: A *decision-time* algorithm with *accelerated learning*

MCTS is known as a *decision-time* algorithm since the *Selection-Expansion-Simulation-Backpropagation* cycle (SESB) is carried out many times for each consecutive time-step in the MDP. The alternative would be to solve the whole MDP at once more akin to policy iteration. In MCTS the SESB cycle is rerun many times at each time-step. This makes sense since action-value estimates are better the closer they are to the root state because this is where states are visited/updated more often (this is true both in *Version 1* and *2* and other Monte-Carlo methods applied to Markov chains (MCMC)).

One difference between MCTS and more standard Monte Carlo rollout algorithms is that it builds a search tree in memory throughout the operational horizon. This is why it is called *tree-search* as opposed to just MC-*search* (MCS). At the first time step MCTS is very similar to MCS but as the MCTS search and the MDP operational horizon progresses more and more information becomes available for *Selection* and *Expansion*. If GLIE is assumed to hold (only heuristically true for UCT and MDP usage), the *Simulation* and *Expansion* policies will therefore improve. The learning can therefore be said to *accelerate* (Wheeler, 2017).

Improved search policies do not, however, mean that action-values lower in the tree are necessarily closer to optimal ones because the amount of information needed to estimate optimal action-values cannot be assumed to be constant: Low in the tree the stronger policies may face more difficult problems. If Chess and Go are used to exemplify this intuition, humans generally require more thinking-time/search to make strong moves late in the game compared to the first few moves even though they accumulate information as the game progresses. In a VRP similarly it could be easier to make strong moves early in the sequence. One intuitive strategy would e.g. be to first route vehicles to different areas in the warehouse after which more precise movements are generated using more fine grained information. Although no attempt to prove this intuition is made here, MCTS provides a means to allow for it if it were true.

5.5. Improving the MCTS exploration policy

So far the $P(S_t, a)$ probability inside the $U(S_t, a)$ formula has been assumed to be a random uniform number (e.g. 0 – 1). It can, however, be an estimate from a function approximation model output. This makes exploration and subsequently backpropagation more biased/tied toward the model, but often more useful in practice. There are some prerequisite factors that strengthen such an intuition:

1. The state space is large and the problem is NP-hard but possible to formulate as an MDP.

2. It is possible to build heuristic state-transitions and state or action-values from a strong reference model that has provable performance on the problem.

3. Using a function approximation model it is possible to successfully *estimate* the state-transitions and state or action-values.

4. There are pre-set timeouts/number of operations before decisions must be made to step in the MDP. These timeouts are significantly longer than the time it takes for the function approximation model to output state-transitions and state or action-value estimates.

Estimates are important here because they eliminate the need for full trajectories in the *Simulation* step, which can speed it up by the same factor as provided by the number of state-visits needed to sample real returns from terminal states.

5.5.1. An exploration model

Model M can be defined very broadly but here it primarily refers to a parametrization of an MDP. Let states *S*, actions *A*, policy probabilities *P* and rewards *R* be parametrized by η where $P_\eta \approx P$ and $R_\eta \approx R$ and $M = \langle P_\eta, R_\eta \rangle$. The states and rewards can then be expressed as follows with the model parameters assuming a whole *S* and *A* are known (Sasikumar, 2017):

Real experience state, state-transition, action and reward:

$$S' \sim P^a_{s,s'}$$

$$R = R^a_s$$

where $P^a_{s,s'} = P[s_{t+1} = s' | s_t = s, a_t = a]$ are transition probabilities.

Simulated experience state, actions and rewards:

$$S_{t+1} \sim P_\eta(S_{t+1} | S_t, A_t)$$

$$R_{t+1} = R_\eta(R_{t+1} | S_t, A_t)$$

The equals sign is used in the lower equation since the model's R_{t+1} is a scalar extracted from a reference R_{t+1}, which in its turn reflects a true S_t and A_t. Tilde is used in the upper equation since S_{t+1} is only an approximation of a true S_{t+1} based on a true S_t and A_t. The aim of *M* is to estimate true rewards $s, a \rightarrow r$ as a *regression problem* and state-transitions $s, a \rightarrow s'$ as a *density estimation problem*.

M's parameters can be uniform random or pre-learnt to rollout paths to the leafs from a true root s_t: $\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, \dots, S_T^k\}_{k=1}^{K} \sim M_v, \pi$ where K denotes number of rollouts and M_v and π denote the pre-learnt model and its rollout policy.

M_v can now be used to produce a replacement for the $P(S_t, a)$ prior probabilities in the action selection formula in 5.3.3 and the below sections will go through how a Convolutional Neural Network can be used as M_v.

5.5.2. Convolutional Neural Network (CNN)

CNN's are currently dominant at detection and classification tasks in computer vision (LeCun et al., 2015). CNN's are a type of *feed forward* neural networks (FFNN) where an input is first *forward propagated* through layers of neurons and synapses, leading to an output that is compared to a target, and the difference is then *backpropagated* through the network while computing change derivatives for the synapses. One difference between CNN's and their predecessor *fully connected* FFNN's is the sharing of synapse weights between neurons. This, in turn, is only possible by the usage of small filters or feature maps to have the neurons focus on different nodes in the previous layer. The number of computational operations needed to update weights is lower in CNN's than FFNN's, leading to more effective function approximation in many cases. FFNN's can in theory solve the same problems as CNN's, but in practice this is often unachievable due to an infeasible amount of computational operations required before synapse weights converge. Currently the state of the art in CNN technology is such that they can generalize geometric patterns on par with the human eye.

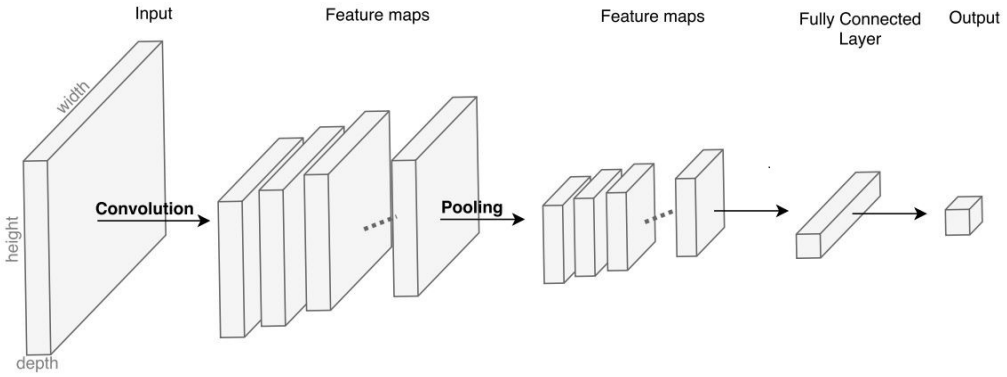


Figure 22: General CNN architecture intuition (Lundeqvist & Svensson, 2017).

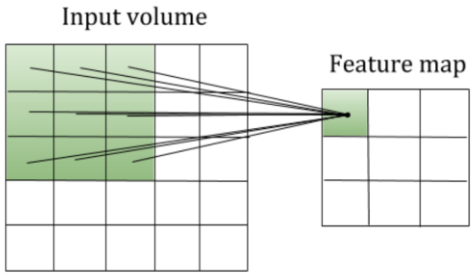


Figure 23: Feature mapping intuition in CNN's (Lundeqvist & Svensson, 2017).

FFNN's are trained using variants of stochastic gradient descent (section 5.5.3) and pattern, batch or mini-batch learning. Pattern learning means weights are updated after one input data is fed and one corresponding cycle (called *epoch*) of forward/back-propagation. Batch learning means that weights are updated only after all inputs in a dataset have been fed and cycled. Mini-batch learning is in between with a manually set number of inputs that are fed and cycled. Each weight has a corresponding *delta-weight* that is used to store the value with which the weight should change when they are updated. When batch or mini-batch is used the delta-weigh values are updated after each cycle using an incremental mean or more advanced methods.

Normally CNN's are trained using mini-batch learning since this strikes a balance between pattern learning – more computational time to update weights and batch learning – which updates weights less often but is slower to learn. The latter is slower to learn because weights are only updated after all data samples have been cycled through the CNN.

If a CNN is forked to have multiple outputs it is said to be *multi-head*.

If a dataset is used for CNN training that is not general enough to adequately represent all possible states in a problem the CNN will *overfit*. *Overfitting* is similar to *bias* (*overfitting* can be seen as the *noun* form of *biased*).

5.5.3. Estimating state-values using gradient descent

The CNN weights are updated using variants of stochastic gradient-descent (SGD) and this section shows how this can be used to move approximation of state-values closer to target state-values. Approximation of action-values is also possible but it is not shown here because the *MCTS-CNN* combination implemented here uses state-values.

Let $\hat{v}$ denote an approximation of a target state-value v so that $\hat{v}(S, w) \approx v_\pi(S)$ where w represents parameter *weights*. Let $J(w)$ be a differentiable function that uses a vector of weights to parametrize a state. Weights can then be updated by gradient descent:

$$\Delta w = -\frac{1}{2}\alpha \nabla_w J(w)$$

where Δw denotes *delta-weights* i.e. how weights should change, where $\nabla_w J(w)$ denotes the gradient with which weights should be changed and α denotes a temperature parameter *learning rate*. The negative sign is there because the idea is to have weights decay and ½ is there because of various statistical reasons (e.g. it is a side-effect of derivation that naturally leads to a L_2 regularized learning rate decay). Since $J(w)$ is differentiable it can be expressed as mean squared error (MSE) between a true state-value and an approximation:

$$J(w) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,w))^2]$$

The delta-weights can be expressed by substituting in $J(w)$:

$$\Delta w = -\frac{1}{2}\alpha \nabla_w \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,w))^2]$$

$$= \alpha \mathbb{E}_\pi[v_\pi(S) - \hat{v}(S,w)] \nabla_w \hat{v}(S,w)$$

If a model is used to give sub-optimal (sampled) reference state -values instead of true ones the full expectation is removed, giving:

$$\Delta w = \alpha\big(v_\pi(S) - \hat{v}(S,w)\big) \nabla_w \hat{v}(S,w)$$

The general SGD formula for sampled state-values is given by the following (Sutton & Barto, 2018):

$$\Delta w_t = \alpha[G_t^\pi - \hat{v}(S_t, w_t)] \nabla_w \hat{v}(S_t, w_t)$$

The only difference here is that $v_\pi(S)$ is replaced by a sampled return from the model G_t^π. If this return is unbiased and α decays the SGD formula is guaranteed to converge on a local optimum (Sutton & Barto, 2018). The formula for updating the weights:

$$w_{t+1} = w_t + \Delta w_t$$

$$= w_t + \alpha[G_t^\pi - \hat{v}(S_t, w_t)] \nabla_w \hat{v}(S_t, w_t)$$

The final state-value approximation is the result of several epochs of *S, A* and *G* samples and weight updates, giving the following equation when solved for state-value approximation (if pattern learning is used):

$$v_\pi(s) \approx \hat{v}(s,w) = \frac{1}{H} \sum_{h=1}^{H} \left[G_h^\pi - \frac{w_{h+1} - w_h}{\alpha_h \nabla_w \hat{v}(s_h, w_h)} \right]$$

where H denotes $\mathbb{N}^+$ number of epochs and α_h a decaying learning rate. Note that this equation only specifies state-values *after* training and does not show how the matrix operations between $\nabla_w \hat{v},\ w$ and G are supposed to work (it relies on the choice of one of the multitude of neural network architectures, of which CNN is one (itself containing a multitude of choices)). The idea is that the fraction term output can jump both above or below the target *G's* and the aim is to have it move closer to them in smaller steps through the decaying α_h. Approximate state-values are however normally just shown with a simple matrix dot-product $\hat{v}(s,w) = w^{\mathrm{T}} x(s)$ (Sutton & Barto, 2018).

For batch-learning delta-weights are stored using incremental means:

$$\Delta w_{\text{Б}+1} = \frac{1}{B}\sum_{\text{б}=1}^{B}\left[\Delta w_{\text{б}} + \frac{1}{\text{б}+1}[\Delta w_h - \Delta w_{\text{б}}]\right]$$

where б denotes an item in a batch, B denotes $\mathbb{N}^+$ number of items in the batches and Б denotes a unique batch. In full batch learning $B = H$ and in mini-batch learning $B < H$. The weight updates are given by the following:

$$w_{\text{Б}+1} = w_{\text{Б}} + \Delta w_{\text{Б}+1}$$

giving the batch weight update equation:

$$w_{\text{Б}+1} = w_{\text{Б}} + \frac{1}{B}\sum_{\text{б}=1}^{B}\left[\Delta w_{\text{б}} + \frac{1}{\text{б}+1}[\alpha_{\text{б}}[G_{\text{б}}^{\pi} - \hat{v}(s_{\text{б}}, w_{\text{б}})]\nabla_w \hat{v}(s_{\text{б}}, w_{\text{б}}) - \Delta w_{\text{б}}]\right]$$

Batch normalization is often used to increase neural network learning stability and reduce overfitting (Doukkali, 2017). It makes sure that layer outputs are normalized within batch statistics. This is essentially the same procedure as is normally applied to data in pre-processing when features are normalized. This type of procedure is also beneficial to carry out between neural network layers so that some neurons or layers do not become overly dominant. A common way to implement this is by having each neuron output be subtracted with the batch mean and divided with the batch standard deviation.

Batch normalization can be used with or without a neural network loss function that penalizes weights by l_2 regularization (also known as *ridge regression*). l_2 regularization is a method with which to penalize weights that grow too large and this overlaps with batch normalization in various ways, some of which are not desirable (Laarhoven, 2017; Doukkali, 2019). It is outside the scope of this study to join the discussion on these undesirable consequences and various combinations of batch normalization and l_2 regularization are used in the experimentation part of this study. There are a multitude

of choices that need to be made when deciding on a neural network architecture (it is a scientific field known as *neural architecture search*).

5.5.4. Producing training data for the CNN

Until now MCTS has only been described in the context of decision time policy iteration for the purpose of solving an MDP VRP. The previous section shows how the MCTS exploration component $P(S_t, a)$ can be replaced by the output of a CNN state-value estimate. However, using a CNN instead of uniform random for $P(S_t, a)$ limits the scope of states that can be visited as dictated by the CNN synapse weights. A uniform random $P(S_t, a)$, which leads to MCTS that slowly converges on optimality if GLIE holds, is hence sacrificed for less variance but more bias and faster convergence on sub-optimal action-value estimates.

For successful use of the CNN the training data needs to be as representable as possible of optimal state-values. Theoretically a strong way to generate the training data is to use MCTS as described initially using a random uniform $P(S_t, a)$ that can then be replaced by the CNN output. Two parameters need to be set and tuned for this to work:

N: visited counts because GLIE infinite visit counts are unavailable.

K: number of rollouts (or timeout scalar) because GLIE infinite rollouts/infinite time are unavailable.

Training data can then be generated in the following way:

1. A VRP is initialized with the MDP formulation in section 5.2. A database is initialized.
2. MCTS is initialized the same way as in section 5.4.1 with a random uniform *Simulation* policy and with the countdown timer version. MCTS is run to produce state-values for the VRP.
3. All nodes in the MCTS tree that are visited > N times are sent to the database. Lower N will lead to a larger database with more variance, higher N will lead to a smaller database with more bias. Each node in the database will hence contain a pointer to a state and a target action-value as produced by MCTS.

Using MCTS for state-value training data generation this way allows for a CNN that is outputting strong GLIE state-values estimates if it is successfully trained (assuming GLIE is true in the training data generation).

This approach is only presented here in *Theory*. In practice, many heuristic procedures where the GLIE connection is severely weakened or completely broken are often necessary to realize a working implementation (similarly to how CNN choices need to be

made explained in section 5.5.3). The main problem is that MCTS using a random uniform *Simulation* policy with full rollouts will require an extreme amount of time to generate sufficient data in large state spaces such as the VRP. A more practical approach is to maintain a MCTS version that generates weaker GLIE outputs quickly. Deepmind uses such an approach in the Alpha Go Zero (AGZ) agent by using a two head CNN with heuristically defined "value" and "policy" that combined constitute something similar to an state-value that is generated without rollouts (Hassabis et al., 2017). This has a weaker GLIE support but is helped by *self-play* (discussed further in section 9.2).

Another approach is to use a completely different model to generate the training data. In the implementation in this study a non-MDP model, *Two-Phase*, is used for this purpose (section 6.1) (this model is not described here in Theory since it is un-disclosable property of *Sony Tenshi-AI* and is more in line with the real complex warehouse VRP environment introduced in section 3 i.e. outside the scope of this study).

The CNN can be pre-trained on a training database using e.g. mini-batch learning. One important prerequisite in MDP function approximation is that the mini-batches consist of randomly picked state-transitions from the database. This is because state-transitions are going to be correlated to a large degree within the MDP sequences they were generated in, potentially causing overfitting for a mini-batch epoch. By randomly selecting nodes from the database this correlation is broken. This procedure is known as *replay memory* or *experience replay* (Sutton & Barto, 2018).

6.Model building

Three main blocks in the DRL VRP model were developed: 1. VRP training data generator (*Two-Phase*). 2. CNN. 3. MCTS. The blocks were developed with the following work-flow:

```
Build VRP data generator
Build CNN
Build MCTS
for episode = 1, M do
    Generate VRP data D
    Cut D into transition states S with V's and P's
    while CNN Loss < T do
        Train/evaluate CNN on S V's and P's
        Modify CNN to minimize Loss
    end while
    Evaluate CNN in MCTS on D
    Tune generator difficulty
end for
```

Algorithm 3: The MCTS-CNN algorithm on VRP's. V's and P's signify heuristic values and policy probabilities respectively. T signifies target loss.

The VRP data generator is hence *Two-Phase* instead of MCTS as described in the *Theory* section. The decision to move away from MCTS data generation was made during early experimentation as the expected problem of computational time became evident (section 5.5.4).

6.1. VRP data generator

The data generation has 5 components:

1. VRP map and pick location generator: Initially this was a TSP with a 19x19 pixel grid with a set depot and 4 pick locations (similar to the example described in section 5.4.2) and it was gradually made more complex. Features were for the most part represented as binary layers. The most difficult problem attempted was a 75 x 49 pixel warehouse map with 50 pick nodes, racks, 3 vehicles, 1 vehicle capacity feature and one congestion feature (the latter two were only briefly tested and are not shown in the result section). A feature such as vehicle capacity requires special treatment since it cannot be directly implemented as a single binary layer. Capacity is a scalar that represents how much load a vehicle can take and could be directly represented as a separate layer. However,

datatypes cannot be mixed in standard deep-learning programs so all binary features therefore need to be converted to the same scalar datatype in that case.

2. *Two-phase model* that outputs approximate solutions to the VRP. The workings of this model will here be briefly covered (subject to non-disclosure): a) Before the model is used the warehouse is digitized such that each pick location is a node in a graph. b) The shortest distances between all nodes are computed using Warshall-Floyd and populated in a distance matrix (Janse van Rensburg, 2019). c) The nodes are clustered using graph-clustering and constraints when relevant. d) Each cluster becomes a TSP and it is solved using Concorde (Applagate et al., 2002; Janse van Rensburg, 2019).

3. *Two-phase to MDP data transformation*. Each Two-Phase VRP solution is transformed into an MDP sequence using the formulation in section 5.2.3. The first *current vehicle* in the VRP is picked randomly. The routes for the vehicles in the Two-Phase model are set to be anti-clockwise in experiments where the start and end locations are the same (a type of travel convention).

4. *Sequence cutting*: Each resultant MDP sequence is cut into transition-states where each state is assigned the same target "value" or return i.e. the total distance travelled for the completed MDP sequence as produced by *Two-Phase*. Each state is also assigned a target "policy" i.e. given a state and a current vehicle, what is the next node it should visit. A delimitation made here is that the sequence cutting is made so that one vehicle is routed at a time: Instead of e.g. first routing vehicle A to its first two nodes before sending out vehicle B to its first node, vehicle A is routed to its first node followed by vehicle B being routed to its first node, followed by vehicle A being routed to its second node etc. When a vehicle has no nodes left it is routed to the depot. This puts a significant limitation to the training database and for a real implementation more exhaustive scenarios would have to be explored.

5. *Augmentation/Noise injection*: There is now a set of transition-states with "value" and "policy" that is 100% biased to the output of the *Two-Phase* model. The CNN also needs to be exposed to transition-states that are not direct outputs of the *Two-Phase* model. This is because the CNN, when used at decision-time will see states that have not been in the training-database. Such states have to be generated to reduce bias/ overfitting. The technique used for this is to take direct outputs of the *Two-Phase* model and *augment/noise-inject* them. First a subset of *Two-Phase* is changed so that the current vehicle's node is changed and/or visited/unvisited nodes are shuffled in various ways. Then a VRP solution is recomputed by *Two-Phase* from the current vehicle position for this new state. The amount of augmentation versus the original solution was hardcoded as 66% for most of the experiments.

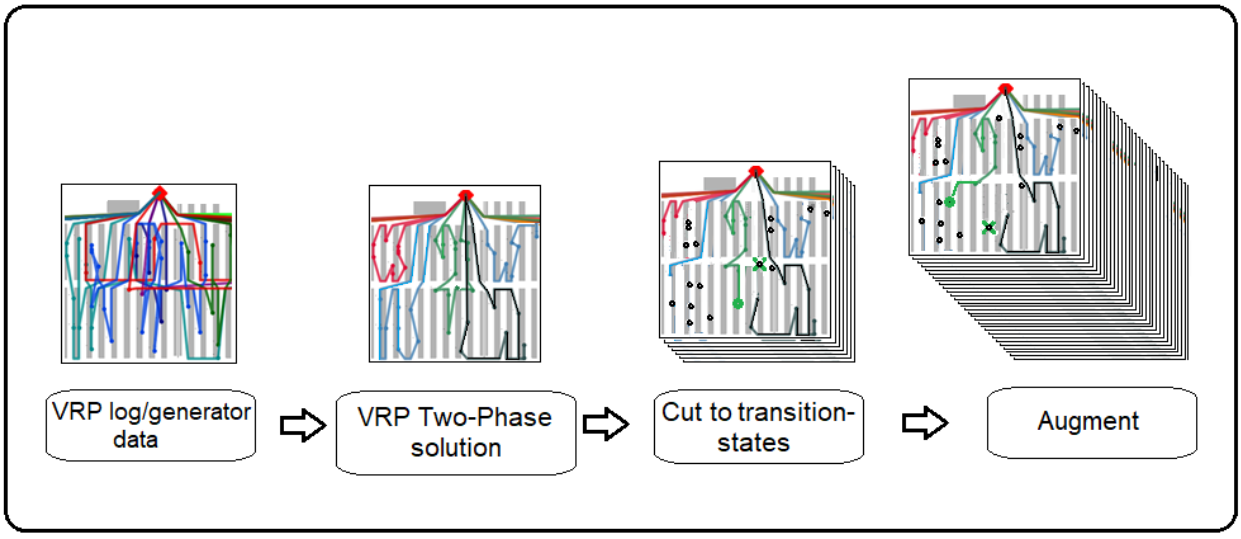


Figure 24: The 4 steps in the VRP data generation. The crosses signify the target next moves given by Two-Phase. The growing stacks signify the increase in amount of data produced.

If transfer learning over various warehouses is to be implemented the visual data will have to be transformed into pre-fixed formats. This was successfully carried out by Deepmind on various Atari games (Mnih, 2013). In the warehouse case the minimum format must be large enough to represent the racks, around 3 pixels for each aligned rack. The largest map used in the VRP generation was 75 x 49 pixels and this hence corresponds to around 25 aligned racks and this is similar to the number of racks in the real datasets available.

6.2. CNN pre-training

The link between the data generator and the CNN is achieved by transforming the map-states into binary image layers that can be read as input layers. There are many questions how this is done in the best manner e.g. how many features/binary layers are needed? Is it better to use a single 8 bit layer? To what extent should pre-processing and feature engineering be used? Answers to these questions will be provided in the section on experimental results (7).

The figure below shows how a state map is fed to CNN with several binary layers:

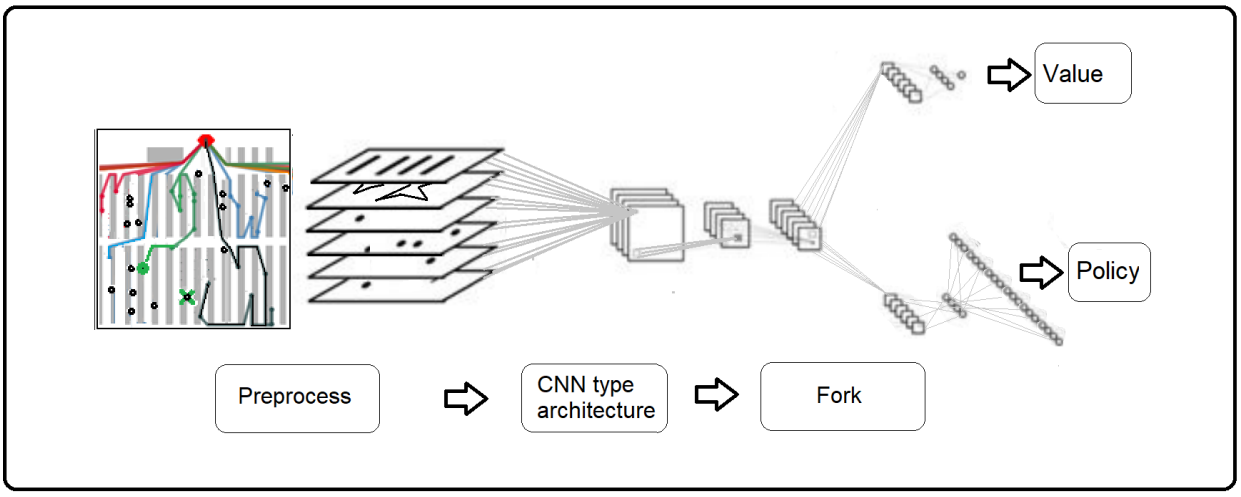


Figure 25: The CNN for VRP. In this example the image is split up into 6 binary layers where the first one is the racks, the second is the paths travelled, the third one is the unvisited nodes, the fourth the visited nodes, the fifth the current pickers locations and the sixth the target next locations. CNN approximates the total distance (V) and the next moves (P).

Figuring out the specificities of the CNN in terms of input and hidden layers, normalization, hyperparameters, outputs, gradient descent function, is an important part of experimentation. One aim, as specified in the thesis questions, is to investigate how this can be done.

The CNN is mainly trained using the following loss function (computed after each forward propagation):

$$L(w) = c_1(V - Z)^2 + c_2(x - \hat{x})^2 + c_3(y - \hat{y})^2 + \lambda\|w\|_2^2$$

where L signifies *loss* for the CNN forward-propagation through weights w; c denotes loss weighting hyperparameters; V is "value" or total distance for the whole VRP as solved by Two-Phase; Z is the corresponding "value" target, $\hat{x}$ and $\hat{y}$ are coordinate "policy" predictions; x and y are corresponding targets; $\lambda\|w\|_2^2$ denotes regularization. This latter term is not used in all experiments but is displayed here to show that either l_2 regularization *or* batch normalization is used. Why the policy predictions are not state or action-value estimates is covered in the section on experimental results since it is a solution to problems that emerged during experimentation.

For the delta-weight updates the ADAM (adaptive moment estimation) optimizer is used (Kingma & La, 2014). ADAM is a recently popular member of a family of enhancement versions to the standard SGD delta-weight formula that make use of old delta-weight updates (sometimes called "moments" or "momentum") at the cost of some increased memory usage.

As a principle during experimentation the CNN is kept as small and simple as possible. After evaluating it on a test set (standard 0.7 0.15 0.15 data splitting is applied) some of its policy predictions are visualized.

6.3. Post-training: MCTS

After the CNN converges on the generated states it is used in *MCTS-CNN* to produce solutions on a batch of uncut VRP data, loaded randomly from a database of all VRP's sequences generated up until that point. The total distance of the solutions is compared to the total distances achieved by the *Two-Phase* model. The MCTS equation used for *Selection* and *Expansion* (section 5.3.3):

$$A_t = \arg\max_{a \in A} \left(Q(S_t, a) + c\hat{v}(S_t, w) \frac{\sqrt{\sum_b N(S_t, b)}}{1 + N(S_t, a)} \right)$$

$P(s, a)$ in the MCTS equation is replaced with the CNN state-value output ($\hat{v}(s, w)$). The "policy" output is thus only used for CNN pre-training. The reason for this is that $P(s, a)$ in MCTS is supposed to be as close as possible to the true state-value but *Two-Phase* does not generate state-values but instead the heuristic surrogates "value" and "policy" (section 5.5.4). "Policy" in the Two-Phase implementation developed here is a coordinate tuple so the transition from this to a usable term within the MCTS action-selection equation is not entirely trivial (see further notes on this in the results section).

As noted previously the action-selection equation does not necessarily mean that *Selection* and *Expansion* ever reach the terminal state. The below figure shows how the MCTS prediction stage may look as it steps in the MDP VRP:

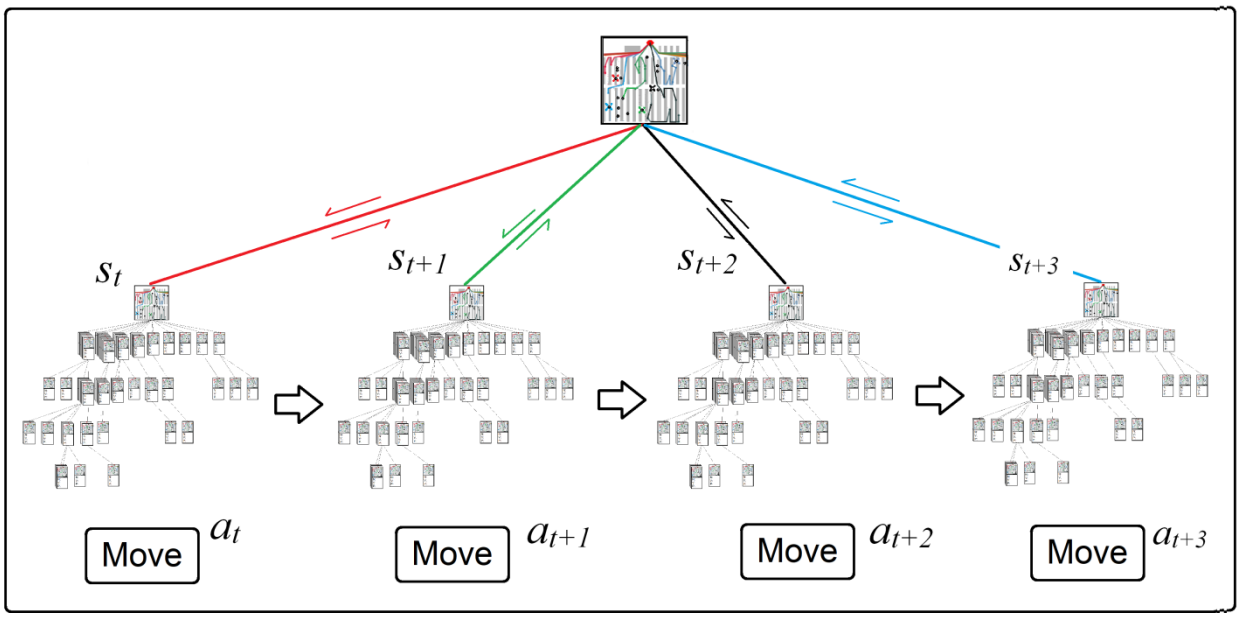


Figure 26: High level aggregated sketch of the decision time MCTS.

Hard coded parameters used in action-selection equation for the experiments were c, which was always set to 1, and a general search timeout before a move had to be made, which was always set to 3 seconds. In a real setting the effect of this latter search timeout could be minimized by concurrency and computing actions for vehicles not when vehicles are standing on a node and waiting, but ahead of time. In the experiments the vehicles were routed one node forward at a time sequentially as shown in the figure above. There are no random variables used at this MCTS stage but recreation of results could never be guaranteed since the 3 second timeout approach was used since the processing units on the computer also handle other tasks.

6.4. Tools

Python (with many libraries), Keras, Tensorflow-GPU, MySQL.

7. Results

This section answers thesis questions 2 and 3 (from section 3.1).

How can MCTS-CNN be used to minimize travel distance in simulated warehouse VRP's?

CNN pre-training on simulated data is possible to a large extent. The features used in the experiments were depot locations, rack locations, available next pick locations, picked item locations, paths travelled so far, and in some experiments vehicle capacities and congestion, and these were successfully used by the CNN to output good approximations of the generated VRP *Two-Phase* reference solutions.

Input layers: It was found that many binary layers, as in AGZ, perform better than a single integer layer. The problem with the integer layer is that overlap between features at the same pixel coordinates causes problems. Either information must be destroyed or new features representing combinations must be expressed on those coordinates and this does not scale well. Multiple binary/scalar layers on the other hand require more preprocessing, take up relatively more memory the more features are used and the CNN needs more weights in the first hidden layer (not necessarily a problem). Experiments on simple VRP's showed that these potential issues are far less severe than the integer alternative.

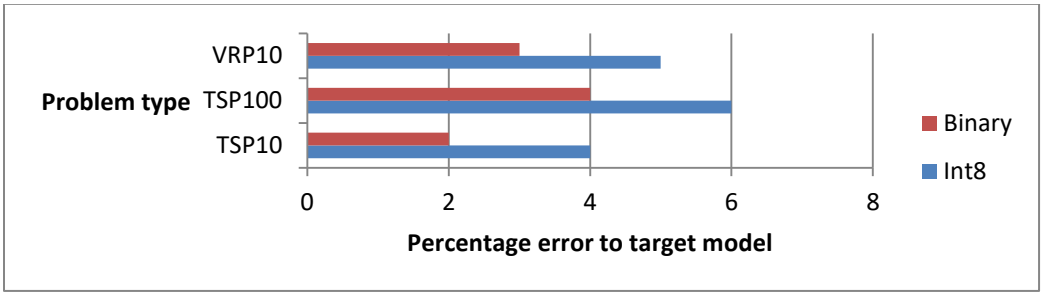


Figure 27: Difference in percentages on simple VRP's due to data type of input layers used.

Choice of features: The main result obtained here was that showing CNN the *traveled paths so far* helped learning. Initially these paths were not shown and it was feared that they would be too noisy or complicated to help, but the CNN made good use of them. Lines should help when lower values get associated with bad line patterns to a large enough degree and vice versa and this was achieved but how this scales is an open question. AGZ showed that complex geometric shapes could be associated with value, but whether this applies to the line geometry of VRP's is not certain.

The selection of the other features was more common sense and analysis of importance was deprioritized against other experiments.

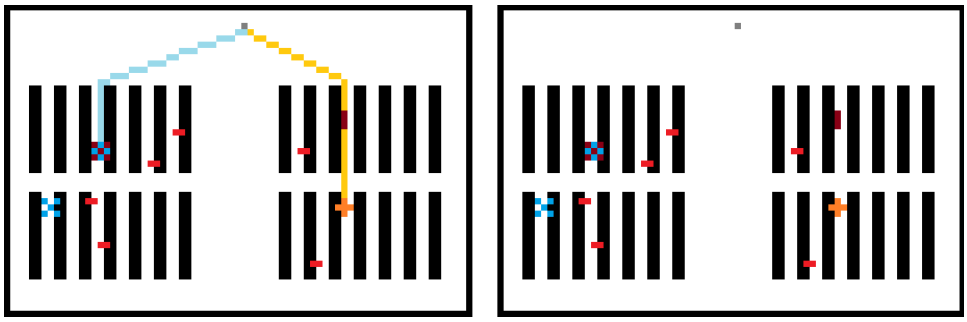


Figure 28: Showing the CNN traveled paths so far (left) helps learning. The right image is the same state without this feature.

Shape of input features: It was found that representing features using various shapes also helped learning. Initially each feature was represented with single pixels and better results were obtained when various 3x3 pixel shapes were used instead. It is already evident in research why this would improve performance: Without shapes different combinations of features can result in signal overlap to a much larger extent in the first convolutional layers in the CNN. The shapes prevent this from happening by setting a stronger footprint in these initial layers. 3x3 pixels were used as the area in which shapes were designed because 3x3 was the CNN kernel size used.

Policy prediction: A modification to the softmax policy prediction in AGZ was implemented. Early experiments using softmax (as in AGZ) seemed to work well but it was nonetheless abandoned due to the following: In AGZ there are 19x19 pixels and each pixel has a neuron in the final policy hidden layer which is responsible for estimating an action-value. In a VRP, however, 19x19 pixels is not enough to represent a real environment and therefore the number of neurons would also have to be larger, leading to more training time. Also, in Go most pixels represent possible moves at a given time, but in a VRP only the unvisited pick locations are possible moves, rendering most of the softmax output redundant. The output was consequently changed from softmax to a single x, y coordinate prediction:

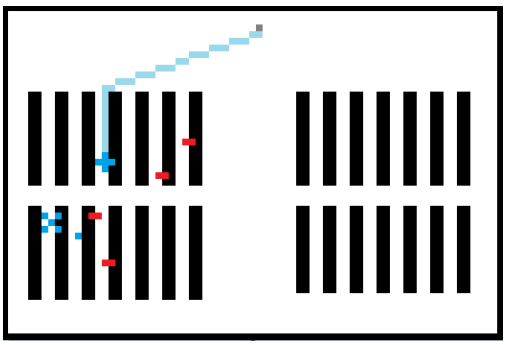


Figure 29: Output from the CNN with the policy prediction modification on a 7 node TSP. The lower blue cross is the target next pick location as produced by the Two-Phase model and the single blue point is the CNN prediction and part of the loss is computed as the distance between this point and the target.

In the above TSP the blue dot is the policy output of the CNN and policy loss is its distance to the lower blue target cross (normalized to between 0-1). For these experiments a rack-modified Manhattan distance was used but in a real setting it is possible to modify it to be exact by querying a Warshall-Floyd distance matrix over the warehouse nodes (section 3).

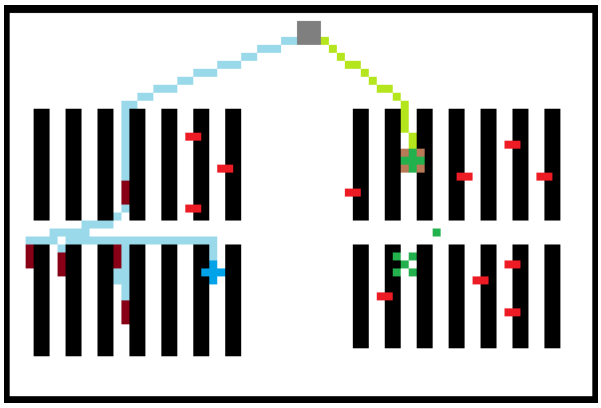


Figure 30: CNN versus Two-Phase rational on a 20 node VRP with 2 vehicles. The brown vertical shapes are visited pick locations. The green cross with brown background is the "current" vehicle (standing on a visited node). The lower green cross and dot are the target and coordinate prediction for where this vehicle should go to next.

Value prediction: Target value was defined as the total distance of the *Two-Phase* reference solution. The CNN value head was set to output scalars that come as close as possible to this target and value loss was computed using Mean Squared Error (MSE).

CNN loss function and its parameters: It was found that the CNN learnt best when the loss "value" component was about twice as large as the "policy" component (controlled by the CNN loss function weighting parameter c). This suggests that it was easier for the CNN to predict target total distances rather than the target next moves for vehicles. This could be because of a variety of reasons such as the use of a single x, y coordinate policy output rather than softmax. One of the disadvantages of DL is that it is a black-box and

therefore hard to extract information on how the input tensors should be updated to improve learning.

CNN layers: For simple VRP's with between 5 and 20 nodes and 1-2 pickers a network consisting of two conv2D layers (with kernel size 3x3, stride 2), one batch normalization layer followed by the fork between policy and value into two heads with fully connected layers leading to the respective value, policy parts, with two additional fully connected layers leading to the x and y's, was deemed sufficient to achieve loss between 0 – 0.1. For more complex VRP's the loss was generally higher with this set up and using three blocks of two or three conv2D layers (3x3 kernel size, stride 2) with skip connections and batch normalization sometimes worked better (neural network search experiments were not carried out). The usage of such "blocks" are sometimes called "residual towers" and "skip connections" is a way to avoid the *vanishing gradient problem* (for details see Fung, 2017). Larger architectures were generally avoided due to the extra training time required.

Augmentation: For all the experiments 1/3 of the states were un-augmented as produced by Two-Phase, whereas 2/3 were augmented (as described in section 6.1). Changing these settings could affect results. More augmentation means more exploration potential for the algorithm, less means more exploitation.

What is the MCTS- CNN performance on simulated warehouse VRP data on the distance minimization KPI level?

It was found that when the CNN loss was 0 – 0.1 the prediction stage *MCTS- CNN* quickly converged on a solution that was as good or sometimes better than *Two-Phase*. When it was 0.1 – 0.2 the CNN prediction visualizations often showed an incorrect rational but the MCTS search could still beat *Two-Phase* after a few seconds of search. When it was above 0.2 *MCTS- CNN* generally converged on a weaker solution for an unknown reason (see discussion below). Experimental results are displayed below:

Nodes	Mobile units	CNN Loss	MCTS vs Two-phase distance (+/- %)	MCTS vs Two-phase pred time (+/- %)
5	1	0	0	1250%
10	1	0.01	0	1300%
5	2	0	0	-22%
10	2	0.05	0	-6%
20	1	0.06	0	1150%
100	1	0.09	-3%	510%
20	2	0.15	-2.20%	-10%
50	3	0.22	4.10%	-62%
80	4	0.26	-0.35%	-35%

Table 1: Results on various VRP types. The experiments with capacities and congestion are omitted here. MCTS denotes MCTS-CNN after the CNN is trained.

The rows that only have 1 vehicle (called *mobile unit* above) resulted in a much larger prediction time for *MCTS- CNN* since the CNN used (Keras with Tensorflow-GPU) always requires ~0.5 seconds to start up before making a prediction. The *Concorde* TSP solver (used exclusively by *Two-Phase* when there is only a single vehicle) on the other hand often predicts TSP's with fewer than 100 nodes in 0.05-0.1 seconds and does not need any "start-up" time (this should be resolvable by pre-loading the weights into RAM but this failed in the above experiments).

8. Discussion

It was found that the *MCTS- CNN* algorithm shows high capability on the simulated data but that implementation in real WMS contexts like the two presented in section 3 is pre-mature. There is not enough data to work with the time-minimization or profitability KPI's, while the distance-minimization KPI is not sufficient to motivate changes in real warehouse operations (section 3). One key missing ingredient is the "box" and the logic of how boxes are filled and why and when they are selected for picking (the so called "box-calculation"). Even if a warehouse is rebuilt to minimize the effect of this and other complicated constraints by introducing repacking areas or moving racks (e.g. the direction Amazon-Kiwa robotics have taken), the pervasiveness and uniqueness of constraints in real-world warehouses has surfaced as a serious threat to generalizable solution models.

One questionmark is also why *Two-Phase* should be the choice of reference model on which the CNN learns. When the VRP difficulty is low (e.g. TSP) it is easy to motivate using it because its proper functioning can be empirically verified, but this quickly becomes impossible with increasing VRP complexity. There are no optimal solutions available for more complex VRP's and the development of a strong reference model should precede the development of a model which depends on that same model for pre-training.

More experiments are needed to deduce why *MCTS- CNN* converged on bad solutions when the CNN loss was above 0.2. One simple explanation was that search timeout was always set to 3 seconds, which may be less than what is needed given the weaker CNN accuracy. For the 50 node 3 vehicle case it was 62% faster at generating a weaker prediction compared to *Two-Phase* but if it were to search more this would likely change. Improving the CNN in pre-training is at least as important though because its loss clearly had an impact on results. Conducting neural architecture search on hyperparameter configurations and how data can be pre-processed for it in the VRP domain would be a suitable next study.

The c parameter in the action-selection formula was in these experiments always set to 1 and changing it could also have a significant impact on results by modifying the exploration/exploitation tradeoff.

For the real domain better warehouse VRP data collection could be set up since this caused limitations for this study. Ways in which to thoroughly log and then analyze picking processes could be devised. For a VRP time minimization KPI this could include pick errors, worker specific performances, missing items, vehicle break downs, congestion, vehicle turn frequencies etc. An increasing number of companies are also

setting up in house sensors to collect such data with high precision. For research this type of data can then be used to build benchmark problems that can be made publicly available for peer collaboration. Currently it is hard to find benchmark datasets on the time minimizing warehouse VRP KPI level, especially ones that are box, rack or AGV inclusive. Companies who have them can be protective of their solutions but they are also generally appreciative of the complexity of warehouse operations and the need for further research.

Concerning scalability a DRL algorithm such as *MCTS- CNN* is appealing since the pre-training part could theoretically bear a large brunt of growing problem complexity: When the VRP complexity grows the CNN can be enlarged and pre-trained for a longer period of time, without necessarily demanding more MCTS search at prediction time. A following question is then whether more complex VRP features can be fed to a CNN. In this study the CNN input was a tensor built out of features such as racks, current vehicle location or unvisited nodes, that all owned one 2D matrix each in the tensor. There is no theoretical reason why features demanding more than 2D (if such are sought) could not also be added to the tensor. One blocking feature is datatype and if a layer requires scalar representation the datatype of all other layers also need to be changed to scalar (binary is preferred due to a lower memory requirement). This is not, however, a significant issue and can be conducted as a pre-processing step as mini-batches are streamed from the training-database to the CNN. There is also no theoretical reason why the box-inclusive VRP cannot be optimized by *MCTS- CNN*. If there are hard constraints regarding boxes the training data can be updated to make certain nodes invisible or undesirable for vehicles. Manual incisions in the training data (e.g. by making nodes invisible) is probably preferred here compared to having a system learn itself how to avoid breaking constraints (e.g. by linking constraint breaking to rewards). Some mixture of a heuristic and a DRL algorithm is thus likely how this subject can be pursued in future real warehouse implementations. The size of the training database needed and how it should best be scaled to tackle specific VRP's is an unresolved question.

9.Appendix

9.1. VRP Set Partition Formulation:

VRP general set partitioning formulation with the same start and end depot (Munari et al., 2017): Let c_r denote the cost of route r, $r \in R$ where R is all possible routes. For a CVRP these routes are only the ones where nodes are visited once, where they start and finish at the depot and where capacity is lower than a set amount. In a VRPTW the routes would also have to be within time-windows. Further cases can also be modeled using the set partition formulation. The total cost of a route that sequentially visits nodes $i_0, i_1, \dots, i_p$, $p > 0$ is given by:

$$c_r = \sum_{j=0}^{p-1} c_{i_j\, i_{j+1}}$$

Let λ_r denote a binary decision variable that is equal to 1 if and only if a certain route r is selected $\lambda_r \in \{0, 1\}$. Let a_r denote a binary vector where $a_r = (a_{r1}, \dots, a_{rn})^T$ and where $a_{ri} = 1$ if and only if route r visits node i. Let K denote the total number of vehicles at the depot. Let C denote the number of nodes. The set partitioning VRP objective function is:

$$\min \sum_{r \in R} c_r \lambda_r$$

$$\text{s.t.} \quad \sum_{r \in R} a_{ri} \lambda_r = 1, i \in C,$$

$$\sum_{r \in R} \lambda_r \leq K,$$

The λ_r and a_r constraints ensure exactly one visit at each node. When R is large a column generation technique is used to create a restricted master problem over small subsets $\bar{R} \subset R$ with $r \in \bar{R}$:

$$\min \sum_{r \in \bar{R}} c_r \lambda_r$$

$$\text{s.t.} \quad \sum_{r \in \bar{R}} a_{ri} \lambda_r = 1, i \in C,$$

$$\sum_{r \in \bar{R}} \lambda_r \leq K,$$

Let $u = (u_1, \dots, u_n) \in \mathbb{R}^n$ and $\sigma \in \mathbb{R}$. These are dual variables linked to the λ_r and α_r constraints and a dual solution $(\bar{u}, \bar{\sigma})$ is obtained for all possible columns. The general solution can be expressed as a resource constrained elementary shortest path problem:

$$\min_{r \in R}(\bar{u}, \bar{\sigma}) = \sum_{i \in N} \sum_{j \in N} \left(c_{ij} - \bar{u_i}\right) x_{rij} - \bar{\sigma}$$

where $x_r = \{x_{rij}\}_{i,\, j \in N}$, $\overline{u_0} = \overline{u_{n+1}} = 0$ and $x_{rij} = 1$ if and only if $r \in R$ first visits i and then goes directly to j.

9.2. Self-play

The use of the MCTS, a CNN and self-play combination was developed by Deepmind to reach state of the art performance in the game of Go with 10^{172} possible state combinations (Hassabis et al., 2017). One key difference between Go and VRP's is that Go is a 2-player turn based game and this allows a methodology known as *self-play* which eliminates the problem of how to tune variance/bias in the training data. "The core idea of the … algorithm is that the predictions of the neural network can be improved, and the play generated by MCTS can be used to provide the training data" (Wheeler, 2017). Deepmind first experimented with semi-supervised learning on historic human games and "expert policies" to help the agent, but then reached the pinnacle of self-play in Alpha Go Zero (AGZ), an agent that generated all its own training data and is completely self-taught.

Understanding self-play and the amount of benefit it can bring to a DRL agent is relevant for this study, since it cannot be used in VRP's because they are not adversarial. First an example analogy: Let's say an agent tries to solve a set of problems by trying to combine experience gained from some already solved problems (low "loss"). If the problems are too easy the agent will not learn because the combinations to solve the problems are already set in stone; if they are too hard the agent will not learn either because the search through combinations will not yield enough positive feedback. The golden zone is somewhere in between where just enough feedback is received to maintain search in the right direction.

In a complex non-self play problem it is first necessary to simplify it to make sure the agent learns properly and then gradually scale it up until the full problem can be attempted. Knowing how to simplify and complicate the problem is hard and in two-player turn based game environments this issue can be significantly alleviated. Since agent capability in that domain is judged solely on whether they win or not, an agent can

be set to play against itself without prior knowledge. The moves in the first game may look like random MC search and after it all states and actions are stored in a database that the losing agent is sent to train on, attempting new ways in which to solve the problem of beating the best agent. After this a rematch is played and if this results in a win the roles are switched and the other agent now has to go back to train on the database, which is now updated with more training data. The intriguing fact is that this continually updated database embodies the exact data needed to beat the best agent i.e. it is the golden zone. Problem complexity can be increased automatically and precisely and through Markov decision process theory it represents convergence on optimal play (assuming enough games are played and stored, good parametrization etc.) (Fu, 2017; Wheeler, 2017; Mnih et.al., 2013).

Self-play can be used with a variety of search methods but it is most intuitive when agents start out without prior knowledge, such as in AGZ. The MCTS part in AGZ is alone capable enough to reach top 5% professional human level in Go (e.g. Crazy Stone, 2014) and the CNN can be seen as an add-on that speeds up search through function approximation.

For CNN loss the Mean Squared Error of value, softmax policy and batch regularization is used:

$$L(\theta) = (W - Z)^2 + \boldsymbol{\pi}^\top \ln \mathbf{p} + \lambda \|\boldsymbol{\theta}\|_2^2$$

Equation 1: The loss function in AGZ. The first expression is the value loss (Z is the result of a self-play game), the second expression is the policy loss (π is the target probabilities and p is the posterior probabilities) and the third expression is regularization of network parameters λ ≥ 0.

9.3. Training database issues

In AGZ a very large training database of 10's of millions of states is stored and this limits overfitting and upholds the optimality convergence, but in order to reach satisfactory levels in other domains techniques that intelligently prunes and appends to the database may be needed. Asynchronous Actor Critic models such as A3C address this problem by enforcing variability between agents and this can allow the throwing away of older data (Mnih et al., 2016). This has led to the fastest training achieved on Atari games and is also the technique used in Deepmind's Starcraft II agent AlphaStar (AS) (Vinyals et al., 2019). Although the exact combination of methods used is not yet published AS uses CNN's and LSTM's (Long Short Term Memory) to read states into a relational graph and output multi-head action sequences for units to cooperate toward a central value baseline (Foerster et al., 2018; ICLR2019; Vinyals et al., 2017, 2018, Mnih et al., 2016, et al.). AS has already shown impressive capability but Starcraft II is a fast realtime environment and a realtime architecture may not necessarily be more applicable to

VRP's than a non-realtime one. AS outputs actions more often than AGZ but AGZ spends more time to search for optimal actions. It is currently also clearly the case that the AS agents are overfitting through their tournament like evolution, and pure self-play has not yet been achieved. Hence there are still many unanswered questions on when and where to apply semi-supervised versus unsupervised learning and how to design the training database, even for self-play environments.

Whether MC or TD(λ) (Temporal Difference learning) works best is another relatable debate. If the state-action space is small enough it is possible to rollout enough paths to build parametric distributions that are more robust than the non-parametric and faster learning but biased TD(λ) alternative and the question is what small enough means. In Go the state-action space is discrete-finite and AGZ shows that in that case it is small enough, which is noteworthy considering 10^{172} possible states. The reason TD(λ) performs worse in that case is because of a phenomenon known as *the deadly triad* i.e. when function approximation is used in conjunction with off-policy training and bootstrapping (Sutton & Barto, 2018). The agent basically gets lost when it bootstraps on freely explored paths. If a VRP environment is preprocessed to meet the same criteria as in Go i.e. that state-actions are discrete-finite it is certainly possible to defend using function approximation with off-policy MC on it, but whether this scales better than devising expert policies and bootstrapping on those is not directly answerable. First a scalable VRP environment needs to be defined and this is yet to be done.

The problems with how to maintain a good training database is hence not resolved in new challenging self-play domains, but it is even less resolved in non-self play ones. In the field of General Adversarial Networks (GANs) this topic is particularly researched and parallels to trying to set up DRL without self-play have already been noticed (Vinyals, 2016). The aim in GAN problems is to minimize the divergence between a generator prior distribution and a discriminator posterior distribution:

$$\min_{\theta} D(p_0, q_\theta)$$

Equation 2: Find parameters θ of network q such that the divergence between the posterior of q and prior p is minimized.

$$D(p,q) = \max_{f \in \mathcal{F}} \mathrm{E}_{x\sim q}\left[g_1(f(x))\right] - \mathrm{E}_{x\sim p}\left[g_2(f(x))\right] \qquad \mathcal{F} \subseteq \mathcal{X} \to \mathbb{R} \text{ and } g_1, g_2 : \mathbb{R} \to \mathbb{R}.$$

Equation 3: The divergence can be expressed as the largest distance between the expected values of convex functions g1 and g2. It can also be expressed as a Kullback-Leibler divergence.

Convergence in that case is often defined as a Nash equilibrium where a cost is assigned to individualistically changing network parameters and it is notoriously difficult to build a practical implementation where this works as intended (Huszar, 2017). Having multiple networks learn by simultaneous gradient descent renders the search space non-

conservative and this upsets stability. Attempts at resolving these issues have been partly successful by changing the objective function but more research is needed (Mascheder et al., 2017).

Up until recently GAN's have mainly been applied to classification whereas the VRP solution performance sought is measured in distance, time or some other scalar which are more akin to regression. Olmschenk, Zhu and Tang propose a way to adopt GAN's to regression by feeding information on discriminator gradient losses directly to the generator (Olmschenk et al., 2018):

$$L = L_{labeled} + L_{unlabeled} + L_{fake} + \lambda \mathbb{E}_{\boldsymbol{x} \sim p_{unlabeled}, \hat{\boldsymbol{x}} \sim p_{interpolate}} \left[(\|\nabla_{\hat{\boldsymbol{x}}}(f(\boldsymbol{x}) - f(\hat{\boldsymbol{x}}))\|_2 - 1)^2 \right]$$

Equation 4: The GAN loss function proposed by Olmschenk, Zhu and Tang. It is a sum of the loss on labeled (train/validation), unlabeled (test) and fake (noise) data and the regularized size of the gradient changes (Olmschenk et.al., 2018).

Without this adjustment the generator may learn how to mimic the unlabeled data leading to feature mixing in the first convolutional layers and overfitting on local minima. Although the same problem concerns classification in practice training is more likely to fail for regression because of the larger variety of possible feature associations. Olmschenk et al.'s description of their adjustment:

> We have the discriminator backpropogate gradients to the generator which tell the generator how it should change the examples its producing. Punishing the discriminator for the size of the norm of these gradients has the effect of restricting the rate of change in how the discriminator views the fake examples. (Olmschenk et al., 2018, p. 8).

It is an attempt to stabilize learning by producing more detailed information for the generator. For the *MCTS-CNN* model a similar principle could be used post-training. The posterior distribution of values at each node in the search tree might indicate how difficult the VRP is: If there are many equally promising child nodes in the tree the VRP is harder; if there are fewer it is easier. This information could be useful for the generator to tune VRP difficulty. Just as for the regression GAN's this does not resolve the implicit issues relating to the presence of multiple learning processes, but practically it could lead to better results.

In this study the above technique is not attempted and instead manual engineering is used inside each iteration of the outer loop of the post-training algorithm. Difficulty is increased by adding more nodes or vehicles to the VRP. The CNN evaluation is done by comparing its results to the Two-phase model and the level of noise injection is hardcoded by augmenting a number of Two-phase solutions. It is a tradeoff between variance and bias dictated by the level of augmentation: More augmentation makes the data more varied, less makes it more biased.

The architecture is set to output moves in the time range of seconds or tens of seconds and this is in line with the time requirements of the lazy models commonly used in warehouse VRP's. The main algorithmic focus in this study is on the extent to which a VRP environment can be fed and learnt by a CNN and the newer methodologies explored in e.g. AS are left for future research.

One such methodology is the unit cooperation towards a centralized value baseline (Foerster et al., 2018). The reason this helps in Starcraft is because there it is often necessary to commit obviously disadvantageous actions on a unit specific level to promote the greater good. In a warehouse VRP setting this could e.g. be applied on the time-minimization level by routing vehicles to assist other vehicles that encounter congestion or missing items.

10. References

Altman, T.: Computational complexity and Algorithms: Presentation on Capacitated VRP's: http://cse.ucdenver.edu/~cscialtman/complexity/, 2019, collected 8-Jan-2019.

Alvarez, A.; Munari, P.: Metaheuristic approaches for the vehicle routing problem with time windows and multiple deliverymen: Gest. Prod., São Carlos, v. 23, n. 2, p. 279-293, 2016.

Applegate, D.; Cook, W.; Dash, S.; Rohe, A.: Solution of a min-max vehicle routing problem, INFORMS, 2002.

Arlotto, A.; Steele, M.: A central limit theorem for temporally nonhomogenous Markov chains with applications to dynamic programming, INFORMS, 2016.

Arnold, F.; Gendreau, M.; Sorensen, K.: Efficiently solving very large scale routing problems, Cirrelt, 2017.

Bartholdi, J. J.; Hackman, S. T.: Warehouse & distribution science, Georgia Institute of Technology, 2014.

Chaslot, G.M.J.B., Bakkes, Sander; Spronch, Pieter: Monte carlo tree search: A new framework for game AI., 2008.

Chaslot, G.M.J.B., et al.: Progressive strategies for monte carlo tree search, NMANC, 2008.

Chiang, W. C.; Russell, R. Simulated Annealing Meta-Heuristics for the Vehicle Routing Problem with Time Windows, Annals of Operations Research, Vol. 93, No. 1, 1996, pp. 3-27.

Chopin, N.: Central limit theorem for sequential Monte Carlo methods and its application to Bayesian inference, The annals of statistics, 2004.

Clark, Jesse: Skynet salesman, Multithreaded, 2016, https://multithreaded.stitchfix.com/blog/2016/07/21/skynet-salesman/, collected 15-Jan-2019.

Cordone, R.; Wolfler C.R.: A heuristic for the Vehicle Routing Problem with Time Windows, Kluwer, Boston, 1999.

Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo Tree Search, 5th International Conference on Computer and Games, 2006.

Crainic, T. G.: Parallel solution methods for vehicle routing problems, Cirrelt, 2007.

Csiszar, Sandor: Two-phase heuristic for the vehicle routing problem with time windows, IMT Budapest, 2007.

Dantzig, G. B; Ramser, J. H.: The Truck Dispatching Problem, Management Science, 1959, p.80-91.

De Backer, B.; Furnon, V.: Meta-heuristics in Constraint Programming Experiments with Tabu Search on the Vehicle Routing Problem, Proceedings of the Second International Conference on Meta-heuristics (MIC'97), Sophia Antipolis, 1997.

De Koster, R., Le-Duc, T., and Roodbergen, K.J.: Design and control of warehouse order picking: a literature review. European Journal of Operational Research 182(2), 481-501, 2007.

Doukkali, F.: Batch normalization in neural networks, Towards data science, https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c, collected 8-Jan-2019.

Edelkamp, S.: Memory-Restricted Search, Heuristic Search, 2012.

Fernandes, P.: Framework for Monte Carlo Tree Search-related strategies in Competitive Card Based Games, Integrated MS report, University of Porto, 2016.

Foerster, J. N.; Farquhar, G.; Afouras, T.; Nadelli, N.; Whiteson, S.: Counterfactual multi-agent policy gradients, Association for the Advancement of Artificial Intelligence, 2018.

Fu, C. M.: AlphaGo and Monte Carlo tree search: The simulation optimization perspective, Winter Simulation Conference, 2016.

Fung, V.: An overviwe of resnet and its variants, Towards Data Science, https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035, collected on 27-Jan-2019.

Gelly, S.; Wang Y.; Munos, R; Teytaud, Y.: Modification of UCT with Patterns in Monte-Carlo Go, RR-6062, INRIA. 2006. ffinria-00117266v3f, 2006.

Goodson, J. C.: Solution methodologies for vehicle routing problems with stochastic demand, dissertation, Iowa Research Online, 2010.

Goodson, J. C.; Ulmer, Marlin W.; Mattfeld, D. C.; Thomas, B. W.: Modeling dynamic vehicle routing problems: A literature review and framework, SLU, 2019.

Hassabis, D.; Silver, D., et al.: Asynchronous methods for deep reinforcement learning. In International Conference on Machine Learning, pages 1928–1937, 2016.

Hassabis, D.; Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search, Nature, 2016,

Hassabis, D.; Silver, D., et.al: Mastering the game of Go without human knowledge, Deepmind, London, 2017.

Henn, S.; Koch, S.; Wascher, G.: Order batching in order picking warehouses a survey of solution approaches, Springer, 2012.

Homberger J.; Gehring H.: Two Evolutionary Meta-Heuristics for the Vehicle Routing Problem with Time Windows: Meta-Heuristics for Location and Routing Problems, Information Systems and Operational Research (Special issue), Vol. 37, 1999, pp. 297-318.

Huszar, F.: GANs are broken in more than one way: The numerics of GANs, https://www.inference.vc/my-notes-on-the-numerics-of-gans/, 2017, collected 8-Jan-2019.

ICLR2019 (Anonynous authors): Deep reinforcement learning with relational inductive biases, under double blind review, 2019.

Janse van Rensburg, L.: Artificial intelligence for warehouse picking optimization - an NP-hard problem, MSc thesis, Uppsala University, 2019.

Kingma, D.; Ba, J.L.: ADAM: A method for stochastic optimization, ICLR, arXiv, 2014.

Laarhoven, T.: L2 regularization versus batch and weight normalization, arXiv, 2017.

Laporte, G.; Toth, P.; Vigo, D.: Vehicle Routing: Historical Perspective and Recent Contributions, EURO, 2013, Journal on Transportation and Logistics 2.1-2

LeCun, Y.; Bengio, Y.; Hinton, G.: Deep learning, Nature p. 436-444, 2015.

Lundeqvist, E.; Svensson, M.: Author profiling: A machine learning approach towards detecting gender, age and native language of users in social media, UPTEC IT thesis, Uppsala University, 2017.

Mandl, P.: Estimation and control in Markov chains, Advances in applied probability, 1974.

Mascheder, L.; Nowozin, S.; Geiger, A.: The Numerics of GAN's, Conference on Neural Information Processing Systems, 2017.

Mnih, V. et al.: Asynchronous methods for deep reinforcement learning, Deepmind, arxiv, 2016.

Mnih, V. et.al.: Playing Atari with Deep Reinforcement Learning, Deepmind, arxiv, 2013.

Munari, P.; Dollevoet, T.; Spliet, R.: A generalized formulation for vehicle routing problems, arXiv, USC working paper, 2017.

Nazari, M.; Oroojloy, A.; Takac, M.; Snyder, L. V.: Reinforcement Learning for Solving the Vehicle Routing Problem, arXiv.org Artificial Intelligence, 2018.

Olmschenk, Greg; Zhu, Zhigang; Tang, Hao: Generalizing semi-supervised generative adversarial networks to regression, arXiv:1811.11269v1, 2018.

Potvin, J.-Y.; Bengio S.: The Vehicle Routing Problem with Time Windows Part II: Genetic Search," INFORMS Journal on Computing, Vol. 8, No. 2, 1996, pp. 165-172.

Psaraftis, H. N.; Wen, Min; Kontovas, C. A.: Dynamic vehicle routing problems: Three decades and counting, Department of transport, University of Denmark, 2015.

Roodbergen K. J.; Koster R.: Routing methods for warehouses with multiple cross aisles, International Journal of Production Research, vol. 39, no. 9, pp. 1865–1883, 2001.

Roodbergen, K.J. and De Koster, R.: "Routing order pickers in a warehouse with a middle aisle". European Journal of Operational Research 133(1), 32-43, 2001.

Ropke, S.: Heuristic and exact algorithms for vehicle routing problems, DTU, 2006.

Salakhutdinov, R.: Deep reinforcement learning and control, MLD, http://www.cs.cmu.edu/~rsalakhu/10703/Lecture_Exploration.pdf, collected 8-Jan-2019.

Sasikumar, S.N.: Exploration in feature space for reinforcement learning, Master thesis, ANU, 2017.

Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems, Principles and Practice of Constraint Programming—CP98, Springer-Verlag, New York, 1998, pp. 417-431.

Singh, S; Jaakkola, T; Szepesvari, C; Littman, M.: Convergence results for single-step on-policy reinforcement learning algorithms, Machine Learning, 39, 2000.

Sutton, R. S.; Barto, A. G.: Reinforcement Learning: An introduction, MIT, complete draft, 2018.

Thangiah, S. R.; Nygard, K. E. ; Juell, P. L.: GIDEON: A Genetic Algorithm System for Vehicle Routing with Time Windows, Proceedings of the Seventh IEEE Conference on Artificial Intelligence Applications, Miami Beach, 24-28 February 1991, pp. 322-328.

Theys, C.; Braisy, O., Dullaert, W.: Using a TSP heuristic for routing order pickers in warehouses, European Journal of Operational Research, vol. 200, no. 3, pp. 755–763, 2010

Thomas, B.W.: Waiting strategies for anticipating service requests from known customer locations, Transportation Science, 2007, p. 319-331.

Thomas, B.W.; White, Chelcea C.I.: Anticipatory route selection, Transportation Science, 2004, p.473-487.

Toffolo, T. A. M.; Vidal, T; Wauters, T.: Heuristics for vehicle routing problems: Sequence or set optimization, arXiv: 1803.06062, 2018.

Toth, P.; Vigo, D. (editors): The Vehicle Routing Problem, Monographs on Discrete Mathematics and Applications, 2002.

Vinyals, O. et al.: AlphaStar team results presentation, https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii, 2019, collected on 27-Jan-2019.

Vinyals, O. et al.: StarCraft II: A new challenge for reinforcement learning, Deepmind, arxiv, 2017.

Vinyals, O.; Bengio S.; Kudlur, M.: Order matters: Sequence to sequence for sets, Google Brain, 2016.

Vinyals, O.; Pfau, D.: Connecting generative adversarial networks and actor-critic methods, Deepmind, arxiv, 2016.

Wheeler, Tim: AlphaGo Zero - How and why it works, http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/, 2017, collected 15-Aug-2018.

Zajac, Zygmunt: Introduction to pointer networks, FastML community, 2017. http://fastml.com/introduction-to-pointer-networks/, collected 8-Jan-2019.

11. Acknowledgements

Lijo George, Sony and the TenshiAI team, Michael Ashcroft, Louis Janse van Rensburg, Suraj Murali, Sangxia Huang, Volker Krüger.