Learning and optimization in combinatorial spaces

With a focus on deep learning for vehicle routing

Kool, W.

Publication date
2022

Document Version
Final published version

Learning and Optimization in Combinatorial Spaces

Wouter Kool

Learning and Optimization in Combinatorial Spaces

with a focus on Deep Learning for Vehicle Routing

Wouter Kool

LEARNING AND OPTIMIZATION IN COMBINATORIAL SPACES

WITH A FOCUS ON DEEP LEARNING FOR VEHICLE ROUTING

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties
ingestelde commissie,
in het openbaar te verdedigen in de Aula der Universiteit
op woensdag 23 november 2022, te 11.00 uur

door Wouter Wilhelmus Maria Kool
geboren te Houten

PROMOTIECOMMISSIE

Promotor:

prof. dr. M. Welling	Universiteit van Amsterdam

Copromotores:

dr. H.C. van Hoof	Universiteit van Amsterdam
prof. dr. J.A.S. Gromicho	Universiteit van Amsterdam

Overige leden:

prof. dr. F. Hutter	University of Freiburg
prof. dr. W.J. Cook	University of Waterloo
dr. B. Dilkina	University of Southern California
prof. dr. A.H.G. Rinnooy Kan	Universiteit van Amsterdam
prof. dr. K. Sima'an	Universiteit van Amsterdam
dr. V. Niculae	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The research in this thesis has been conducted at the Amsterdam Machine Learning Lab (AMLab) of the University of Amsterdam. Funding was provided by ORTEC.

SUMMARY

In this thesis, we develop methods for using machine learning to solve combinatorial optimization problems, with a focus on vehicle routing problems. The thesis consists of two parts. In Part i (Chapters 3 and 4), we develop practical methods using machine learning models to solve different variants of vehicle routing problems. As these models represent probability distributions over combinatorial spaces, in Part ii (Chapters 5 and 6), we focus on sampling from such models and optimizing their parameters.

Specifically, in Chapter 3, we use *reinforcement learning* to train the *attention model*, which represents a construction heuristic, to solve different variants of routing problems. In Chapter 4, we present *deep policy dynamic programming*, which uses another learned model to guide a restricted dynamic programming algorithm, for improved performance on routing problems and the ability to handle complex constraints such as time windows.

Given the deterministic nature of combinatorial problems, duplicate samples from the models in Part i are uninformative, so Part ii focuses on sampling without replacement from such models. In Chapter 5, we present *ancestral Gumbel-top-k sampling* as an efficient method for drawing samples without replacement from structured models over combinatorial domains, and we illustrate the general applicability beyond routing problems. In Chapter 6, we derive statistical gradient estimators based on such samples without replacement, which can be used to improve the gradient-based training procedure for the model in Chapter 3.

This thesis is based on the following four papers:

- Wouter Kool, Herke van Hoof and Max Welling. "Attention, Learn to Solve Routing Problems!" In *International Conference on Learning Representations (ICLR), 2019*.
- Wouter Kool, Herke van Hoof and Max Welling. "Ancestral Gumbel-Top-k Sampling for Sampling Without Replacement." In *Journal of Machine Learning Research (JMLR), 2020*.
- Wouter Kool, Herke van Hoof and Max Welling. "Estimating Gradients for Discrete Random Variables by Sampling without Replacement." In *International Conference on Learning Representations (ICLR), 2020*.
- Wouter Kool, Herke van Hoof, Joaquim Gromicho and Max Welling. "Deep Policy Dynamic Programming for Vehicle Routing Problems." In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR), 2021*.

SAMENVATTING – SUMMARY IN DUTCH

In dit proefschrift onwikkelen we methoden om machine learning ('machinaal leren') te gebruiken voor het oplossen van combinatorische optimalisatieproblemen, met een focus op routeringsproblemen. Het proefschrift bestaat uit twee delen. In Deel i (Hoofdstukken 3 en 4) ontwikkelen we praktische methoden die machine learning modellen gebruiken om verschillende varianten van routeringsproblemen op te lossen. Omdat deze modellen kansverdelingen representeren over combinatorische ruimtes, richten we ons in Deel ii op het verrichten van trekkingen uit zulke verdelingen en het optimaliseren van hun parameters.

In meer detail, in Hoofdstuk 3 gebruiken we *reinforcement learning* om het *attention model* te trainen, welke een constructieheuristiek representeert om verschillende varianten van routeringsproblemen op te lossen. In Hoofdstuk 4 presenteren we *deep policy dynamic programming*, dat een ander geleerd model gebruikt voor het sturen van een dynamisch programmeringsalgoritme met een beperkte zoekruimte, met als resultaat een verbeterde prestatie op routeringsproblemen en de mogelijkheid om om te gaan met complexe restricties zoals tijdvensters.

Gegeven de deterministische aard van combinatorische problemen zijn duplicaattrekkingen uit de modellen in Deel i niet informatief, dus Deel ii richt zich op het verrichten van trekkingen zonder terugleggen van zulke modellen. In Hoofdstuk 5 presenteren we *ancestral Gumbel-top-k sampling* als een efficiente methode voor het trekken zonder terugleggen uit gestructureerde modellen over combinatorische domeinen, en we laten de brede toepasbaarheid zien naast routeringsproblemen. In Hoofdstuk 6 leiden we statistische schatters af voor gradienten op basis van trekkingen zonder terugleggen, welke kunnen worden gebruikt om de gradient-gebaseerde trainingsprocedure voor het model in Hoofdstuk 3 te verbeteren.

Dit proefschrift is gebaseerd op de volgende vier artikelen:

- Wouter Kool, Herke van Hoof en Max Welling. "Attention, Learn to Solve Routing Problems!" In *International Conference on Learning Representations (ICLR), 2019*.
- Wouter Kool, Herke van Hoof en Max Welling. "Ancestral Gumbel-Top-*k* Sampling for Sampling Without Replacement." In *Journal of Machine Learning Research (JMLR), 2020*.
- Wouter Kool, Herke van Hoof en Max Welling. "Estimating Gradients for Discrete Random Variables by Sampling without Replacement." In *International Conference on Learning Representations (ICLR), 2020*.
- Wouter Kool, Herke van Hoof, Joaquim Gromicho en Max Welling. "Deep Policy Dynamic Programming for Vehicle Routing Problems." In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR), 2021*.

LIST OF PUBLICATIONS

This thesis is based on the following papers:

- Wouter Kool, Herke van Hoof and Max Welling. "Attention, Learn to Solve Routing Problems!" In *International Conference on Learning Representations (ICLR), 2019*.
- Wouter Kool, Herke van Hoof and Max Welling. "Ancestral Gumbel-Top-k Sampling for Sampling Without Replacement." In *Journal of Machine Learning Research (JMLR), 2020*.
- Wouter Kool, Herke van Hoof and Max Welling. "Estimating Gradients for Discrete Random Variables by Sampling without Replacement." In *International Conference on Learning Representations (ICLR), 2020*.
- Wouter Kool, Herke van Hoof, Joaquim Gromicho and Max Welling. "Deep Policy Dynamic Programming for Vehicle Routing Problems." In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR), 2022*.

I was the lead in the projects that resulted in these papers: the main ideas originated from me, I conducted all experiments and did most of the writing. Herke and Max had important advisory roles and provided feedback on the writing. Herke also helped by (re-)writing some paragraphs, when required to meet deadlines (thanks!). Joaquim helped shaping the idea for deep policy dynamic programming.

During the PhD I have also authored the following papers:

- Wouter Kool, Herke van Hoof and Max Welling. "Stochastic Beams and Where to Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement." In *International Conference on Machine Learning (ICML), 2019*. **Best Paper Honorable Mention**
- Wouter Kool, Herke van Hoof and Max Welling. "Buy 4 REINFORCE Samples, Get a Baseline for Free!" In *Deep Reinforcement Learning meets Structured Prediction Workshop at ICLR, 2019*.
- Wouter Kool, Chris J. Maddison and Andriy Mnih. "Unbiased Gradient Estimation with Balanced Assignments for Mixtures of Experts." In *I Can't Believe It's Not Better Workshop (ICBINB) at NeurIPS, 2021*. **Best Poster Award**
- Wouter Kool, Joep Olde Juninck, Ernst Roos, Kamiel Cornelissen, Pim Agterberg, Jelke van Hoorn, Thomas Visser. "Hybrid Genetic Search for the Vehicle Routing Problem with Time Windows: a High-Performance Implementation." In *12th DIMACS Implementation Challenge Workshop*.

The first paper on stochastic beam search was further developed into ancestral Gumbel-top-k sampling. The second paper was an early version which resulted in the paper on gradient estimation with discrete random variables. The last paper is the result of my internship at DeepMind in 2021, where Chris had an important advisory role, and Andriy was my main supervisor, who gave valuable feedback and helped with writing of the paper. The last paper describes our winning submission for the VRPTW track of the 12th DIMACS implementation challenge.

Finally, I enjoyed working with Iris to contribute to the following paper:

- Iris A. M. Huijben, Wouter Kool, Max B. Paulus and Ruud J. G. Sloun. "A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning." To appear in *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 2022*.

CONTENTS

1 INTRODUCTION

1.1 LEARNING TO OPTIMIZE DECISIONS

You make decisions every day. Some decisions are naturally *continuous*: what time do you stop working? Other decisions are *discrete*: will you work from home or from the office today? After all, even though *hybrid* is the new normal, you can only be at one physical place at a time: a natural constraint. In some cases, this can be even more challenging: imagine you're a delivery driver, having to deliver hundreds(!) of packages each day! Which package should you deliver first, second, third, etc.? *Combining* these decisions you arrive at a *combinatorial* decision making or *optimization* problem. Looks familiar? Indeed, it is the famous *travelling salesman problem* (TSP) (Held and Karp, 1971).

As a delivery driver ('21st century salesman'), you'd be more than happy to optimize your decisions and minimize your travels. Such optimization is a good thing: fewer resources (time, energy, human effort) are used to achieve the same effect: getting packages to happy customers. At large scale, the savings can be enormous, and as such, the optimization of (combinatorial) decisions has been extensively studied in the field of *operations research* (OR) (Taha, 2011). Numerous examples exist, such as vehicle routing (Toth and Vigo, 2014), optimization of schedules (Pinedo, 2012), and many more.

But what actually defines an optimization problem? Through the eyes of a mathematician, an optimization problem has an *objective* that should be minimized (or maximized) by selecting the value of certain *decision variables*, which should represent a *feasible* (valid) solution as defined by a set of *constraints*. In general, an optimization *problem* is characterized by its mathematical structure, whereas different *instances* of such a problem are characterized by different input *parameters*, or *data*, with that given structure. For example, the TSP considers the *problem* of finding the shortest route visiting a set of locations (returning at the start location), whereas a specific TSP *instance* specifies the distances between locations (explicitly or implicitly through a set of coordinates for locations).

"Given a set of locations, with distances d_{ij} between locations i and j, what is the shortest roundtrip visiting all locations?" While seemingly a natural formulation of the TSP, this question highlights the 'traditional' view of OR, which concerns the optimization problem, but neglects a very important aspect: the data. In practice, do we actually care about the shortest *distance*? Shouldn't we minimize driving *time* or *cost* (fuel, salary, depreciation) rather than distance? If so, how can we compute, or at least estimate, the driving time or cost for certain trips, taking into account

unknown factors such as traffic, weather and road conditions? Very likely, the answer involves (historical) data, transitioning the problem into the fields of *data science* (DS) (Provost and Fawcett, 2013) and *machine learning* (ML) (Bishop, 2006), often popularly referred to as *artificial intelligence* (AI). If we use machine learning to make predictions about the input for an optimization problem, this is often referred to as the *"predict, then optimize"* paradigm (Elmachtoub and Grigas, 2021). Using machine learning in this way allows us to make sure that we solve *the right problem instance*.

If we know the right problem instance to solve, we remain with an equally important question: how can we actually *solve* it? Can we use machine learning here as well? This brings us at the scope of this thesis: a second paradigm that fuses machine learning and optimization, which we refer to as *"learning to optimize"* (Li and Malik, 2016). Consider a setting where we *repeatedly* have to solve different, but related, instances of a specific combinatorial optimization problem, for example, the daily routing problem for a local delivery service. There may be patterns in the data of the problem instances, and as a result, good or even optimal solutions for those instances may exhibit certain patterns as well. Can these patterns be learned? And can they be used to solve new problem instances faster, or find better quality solutions given a limited amount of computation time? In other words, can we use machine learning, to *learn to optimize*?

Whereas combinatorial optimization problems have been studied for decades, the addition of machine learning to the combinatorial optimization toolbox has emerged only in recent years, primarily as a result of the success of *deep learning* (DL) and *reinforcement learning* (RL) to perform complex tasks. DL enables the training of large scale (deep) *neural network* (DNN) models, by gradually improving their parameters using samples from a dataset, a technique known as *stochastic gradient descent* (SGD). RL enables models to learn without a dataset, by gaining experience from interacting with an environment. This thesis investigates the application of deep learning on a special set of combinatorial optimization problems: *vehicle routing problems* (VRPs). In particular, we transform the problem of optimization into a statistical problem of *modelling* the conditional distribution of good (or even optimal) solutions given a problem instance. As such, the DNN models used in this thesis represent probability distributions over a combinatorial domain of solutions, which can be considered more generally as discrete structures. Therefore, this thesis also presents methods to sample from such models and optimize their parameters.

The research in this thesis is motivated by our belief that machine learning, especially deep learning, has a large and underexplored potential to improve the way we solve (combinatorial) optimization problems. But what makes us believe in this potential? The answer is *AlphaGo* (Silver et al., 2016), especially *AlphaGo Zero* (Silver et al., 2017), DeepMind's neural network based algorithm that has beaten the worlds best human player (and the best human-designed algorithm) in the popular boardgame *Go*. While Go, being a two player game, is different from combinatorial optimization, both are computational problems that can, in principle, be solved by exhaustive search. However, in practice, we can only consider an extremely small part of the enormous search space, and AlphaGo has been very successful by using

deep neural networks to identify the most promising moves and board positions to consider in the search. By learning from experience, obtained by playing Go 'against itself', AlphaGo's neural network proved to be much more powerful than human designed heuristics for selecting moves and scoring board positions. Since combinatorial optimization algorithms typically contain many human designed heuristics as well, we believe that there is similar potential in replacing parts of their internal decision strategies by deep neural networks. Even a marginal improvement of the quality of the optimization result would lead to tremendous savings of money and resources in practice, given the vast amount of optimization problems that are solved on a daily basis in the world. This thesis can be seen as a *Go!* towards unleashing this potential.

1.2 SCOPE & RESEARCH QUESTIONS

This thesis is structured in two parts. In Part i, we introduce the idea of machine learning applied to combinatorial optimization problems, and we develop models that can generate solutions to vehicle routing problems. To train such models using gradient descent, we need to draw samples from the model and estimate the gradient of the objective (known as the *loss function*) with respect to the model parameters. Therefore, in Part ii, we develop principled methods for sampling (without replacement) from models over combinatorial spaces and optimizing their parameters. These methods are applicable (but not limited) to the routing models in Part i. The chapters in both parts of this thesis are structured around a set of research questions.

1.2.1 Part i: routing & reinforcement learning

Research questions 1 and 2 are the focus of Part i of this thesis, which provides two examples of applying learning for routing problems. The first is based on reinforcement learning (learning from experience), while the second combines supervised learning (learning from examples) with an OR technique known as *dynamic programming* (Bertsekas, 2017).

RESEARCH QUESTION 1: *How can we use reinforcement learning for learning to solve vehicle routing problems?*

In Chapter 3, adapted from our publication Kool et al. (2019a), we address this question and develop the *attention model*, based on the popular Transformer architecture (Vaswani et al., 2017), which we train using a special variant of REINFORCE (Williams, 1992), to solve five different variants of routing problems.

RESEARCH QUESTION 2: *How can we integrate machine learning models and dynamic programming for vehicle routing problems?*

In Chapter 4, adapted from our publication Kool et al. (2022a), we address this question and develop *deep policy dynamic programming* (DPDP) as a method to combine a learned graph neural network model (Joshi et al., 2019a) for vehicle routing problems with the principle of dynamic programming, for three different routing problems.

1.2.2 Part ii: sampling & statistical estimation

Most of the problems considered in Part i are *deterministic*, such that duplicate samples are uninformative when training models to solve these problems. Therefore, Part ii of this thesis provides technical contributions that enable efficient *sampling without replacement* from models over combinatorial domains, and using these samples for gradient based optimization of those models.

RESEARCH QUESTION 3: *How can we obtain unique samples from models over combinatorial domains?*

To answer research question 3, we present *ancestral Gumbel-top-k sampling* (Kool et al., 2020a), a generalization of *stochastic beam search* (Kool et al., 2019c) in Chapter 5. This algorithm, an extension of ancestral sampling, can efficiently draw multiple samples without replacement from a distribution defined over a combinatorial domain.

RESEARCH QUESTION 4: *How can we improve gradient based training of neural network models using unique samples?*

In Chapter 6, adapted from our publication Kool et al. (2020b), we answer research question 4 by deriving the *Unordered Set Policy Gradient Estimator*, which can be used to train models with SGD using samples without replacement, obtained using ancestral Gumbel-top-k sampling (or any other algorithm).

2 BACKGROUND

2.1 COMBINATORIAL OPTIMIZATION

Combinatorial optimization (Papadimitriou and Steiglitz, 1998; Wolsey and Nemhauser, 1999; Schrijver, 2003) considers the problem of finding of an optimal solution from a finite set of discrete elements. This set is defined as a combination of certain decision variables with a set of constraints, and is too large to apply exhaustive search, i.e. consider all elements in the set to find the optimum.

2.1.1 P vs. NP

Combinatorial optimization problems can be classified in terms of their *complexity* (Papadimitriou and Steiglitz, 1998). The class P contains problems which can be solved efficiently, which means that the time to solve them is bounded by a *polynomial* function of the problem size. Each problem in P is also in class NP, which means *non-determinstic polynomial* and is the class of all problems for which a solution can be *verified* in polynomial time, but for which it is an open problem whether a solution can also be found in polynomial time. As an example, consider the *decision version* of the TSP: does a tour of a given maximum length exist? Given such a tour, we can efficiently verify its length, so the problem is in NP, but it is unknown if an efficient algorithm exists to find such a tour (assuming it exists), i.e. whether the problem is in P as well. If we are interested in finding the *shortest possible tour*, it is not even clear how we can verify that a given tour is actually the shortest possible tour. Therefore, the *optimization version* of the TSP is not in NP (as its solution cannot be verified efficiently), but is said to be *NP-hard*, i.e. at least as hard as the problems in NP.

The P vs. NP question (Cook, 2006) asks whether any problem that can be verified efficiently (i.e. is in NP) can also be solved efficiently (i.e. is in P), which would imply that P = NP (as P $\subset$ NP). While proofs that P = NP (or P $\neq$ NP) are written with high frequency, they are proven incorrect with the same frequency (Woeginger, 2016) and the million dollar prize for solving the problem still stands[1]. The common belief is that P $\neq$ NP (Gasarch, 2012), which would imply that for many problems of practical interest, simply no algorithm exists that can find the optimal solution to any problem instance in polynomial time. Whereas this may seem a disappointing conclusion, actually, from a practical perspective, we may not strictly require optimal solutions for all possible problem instances. Instead, we may be interested

1 The P vs NP problem is one of the 7 millenium prize problems (Carlson et al., 2006).

in finding solutions of reasonably good quality, only for practically relevant cases. Therefore, a lot of research is going into the development of *heuristics* (Michalewicz and Fogel, 2013; Hromkovič, 2013; Burke et al., 2014; Marti et al., 2018), which do not have guarantees but can find good solutions quickly. As such, the methods proposed in this thesis, can be seen as *learned heuristics* for combinatorial optimization problems.

2.1.2 Routing problems

The TSP (Held and Karp, 1971) is one of the most famous problems in computer science and the prototypical example of a *routing problem* that considers the problem of finding efficient routes to visit a set of locations. Many variants of routing problems exist, mostly generalisations of TSP, such as the *capacitated vehicle routing problem* (CVRP) (Toth and Vigo, 2014), which considers the problem of finding multiple routes to visit a set of customers, where the total *demand* of customers assigned to each route is limited by the vehicle capacity. Other variants include additional practical constraints such as the *VRP with time windows* (VRPTW) (Bräysy and Gendreau, 2005), which is a generalization of the CVRP with the additional constraint that each customer must be visited in a specific time window. Generally, vehicle routing problems are NP-hard (Lenstra and Kan, 1981).

Routing problems are relevant as they occur in practice, where a reduction of total route distance by a few percent can imply savings of millions of euros for a large company (Kant, 2019). Therefore, many variants have been studied over decades of research. Nonetheless, it remains challenging to solve to optimality instances of size beyond a few hundred customers (Pessoa et al., 2020). Also, different problem variants have different types of constraints and their solutions may be vastly different, such that specialized algorithms are required for different problem variants. As it is non-trivial to develop such algorithms, it is promising to use machine learning to learn specialized algorithms (or algorithm components) that work especially well for specific problem variants on a specific distributions of instances.

Generally, algorithms for solving routing problems can be divided into *exact algorithms*, which come with guarantees on their performance, and *heuristics*, which lack such guarantees but are designed to work well in practice. Exact algorithms are often based on branch & bound (Lawler and Wood, 1966), including variants such as branch & price (Barnhart et al., 1998), where VRPSolver (Pessoa et al., 2020) is a prominent open-source example that is flexible and can be used to solve many problem variants. Dynamic programming (see Section 2.1.3) (Bellman, 1952) is often used as an exact method to solve subproblems in branch & price algorithms. Heuristic methods come in a wide range of varieties, and most rely on some form of search (Schrimpf et al., 2000; Ropke and Pisinger, 2006; Helsgaun, 2017; Vidal et al., 2012; Vidal, 2022; Accorsi and Vigo, 2021). These methods often use efficient operators to make local changes to solutions or combine different solutions into new solutions. They need non-trivial adaptions to be used for different problem variants, as the operators may not always be compatible with different constraints.

While machine learning based approaches still need to be adapted to support different constraints, the learning aspect may help with adapting the solving strategy to the specific problem variants.

2.1.3 Dynamic programming

Dynamic programming (DP) is a paradigm for solving optimization problems by breaking them down into smaller subproblems. Famous examples are Dijkstra's algorithm (Dijkstra, 1959) for finding the shortest path between two locations, and the Held-Karp algorithm for the TSP (Held and Karp, 1962; Bellman, 1962). The Held-Karp DP algorithm for the TSP is expressed by the following recursive formula:

$$C(S, i) = \min_{j \in S \setminus \{i\}} C(S \setminus \{i\}, j) + c_{ji} \quad \forall S, i \in S. \tag{1}$$

In this formula, $C(S, i)$ is the total cost or distance of the optimal (lowest cost/distance) route starting at a start node 0 and ending at node $i \in S$, while visiting all nodes in S. c_{ji} is the cost/distance to move from node j to i. By the principle of optimality, any part of an optimal route must also be optimal, since otherwise the route can be improved, so the optimal route visiting nodes in S while ending at i must necessarily also visit the nodes in $S \setminus \{i\}$ in optimal order, while ending at the node j which is visited before i. Since we do not know which node j should be visited before i in the optimal route, the recursion in equation 1 considers all candidates to determine the one with minimum total cost, taking into account the cost c_{ji} to move from j to i. By recursively computing equation 1 for all S and $i \in S$, we can find the optimal solution to the TSP problem.

Equation 1 represents the natural *backward view* of the dynamic program, where the cost of a solution is expressed in terms of the cost of solutions to smaller subproblems. However, to compute $C(S, i)$ in equation 1, we must compute $C(S \setminus \{i\}, j)$ for smaller subproblems first, which again requires smaller subproblems to be computed first. A naïve recursive implementation of equation 1 would compute the same subproblems over and over again. As an alternative, we can take a *forward* view of the problem, in which smaller solutions get extended and suboptimal partial solutions get removed to apply the DP principle. Specifically, for TSP, we can start with an empty solution, and iteratively extend it into all possible solutions with one node, then into all possible solutions with two nodes, etc. In each stage, we can gather all solutions that have the same set of visited nodes and current node, i.e. all solutions that correspond to the same *DP state*, and we only need to keep the best one for each DP state (implicitly computing equation 1). As such, it can be seen as a brute force search, that removes partial solutions that are somehow *dominated* by better partial solutions. The overall complexity of this algorithm for TSP is $O(n^2 2^n)$, which, in terms of complexity, makes DP the best-known algorithm that guarantees optimal results. An additional benefit of the forward view of DP is that we do not need to keep all solutions at any stage, but we can keep a limited set with the most promising (according to some criterium, such as the cost) solutions, which

is known as *restricted dynamic programming* (Malandraki and Dial, 1996; Gromicho et al., 2012a), and analogous to a *beam search*, which can be seen as a restricted brute force search. By restricting the size of the search space, restricted DP can be seen as a heuristic variant of DP, where the quality of the heuristic depends on how the DP search space is restricted. In Chapter 4 we propose to use a learned neural policy to restrict the state space.

2.2 DEEP (REINFORCEMENT) LEARNING

Deep learning (DL) (Goodfellow et al., 2016) is the name given to the training of large scale (deep) neural network (DNN) models with many layers. Deep learning has seen an enormous rise of popularity in recent years, due to many successes achieved in different fields (LeCun et al., 2015). Whereas deep learning originated in domains such as computer vision (Krizhevsky et al., 2012), speech recognition (Graves et al., 2013), and machine translation (Bahdanau et al., 2015), in recent years, many more applications have been added, which often involve learning with *structured data* such as graphs (Kipf and Welling, 2017). Whereas *supervised learning* (SL) based on (input, output) pairs of examples (such as images with labels) is the most widely used technique, *reinforcement learning* (RL) (Sutton and Barto, 2018) has been hugely successful as an alternative paradigm that can learn from experience by interacting with an environment.

2.2.1 Neural networks and gradient based optimization

A neural network $f_{\boldsymbol{\theta}}(\mathbf{x})$ is a mathematical function that has as input a vector $\mathbf{x}$ and depends on a set of *parameters* $\boldsymbol{\theta}$. It is designed to be flexible such that given the right set of parameters, it can represent a desired function. A *deep* neural network (DNN) is composed of *many layers*, where each layer consists of multiple *neurons*, each of which computes a (different) linear transformation of its inputs, followed by a nonlinear activation function. The *rectified linear unit* $\text{ReLU}(x) = \max(x, 0)$ is the simplest and most popular used nonlinear function, sometimes called an *activation* function. The set of output values, sometimes called *activations*, of the neurons in a layer is considered a *representation* of the input data. The neural network can thus be seen as sequentially updating an internal (hidden) representation of the input data, which is used to make the final prediction using the output layer of the neural network. Figure 1 gives an example of a neural network with an input layer, one hidden layer and an output layer. Increasing the number of hidden layers increases the flexibility or *representational capacity* of the neural network, which explains why *deep* neural networks typically have many layers.

The goal of deep learning is to find parameters $\boldsymbol{\theta}$ such that the DNN $f_{\boldsymbol{\theta}}(\mathbf{x})$ represents a useful function, where the usefulness is specified through a *loss function* $\mathcal{L}$ that quantifies the performance for some task: a lower loss corresponds to better

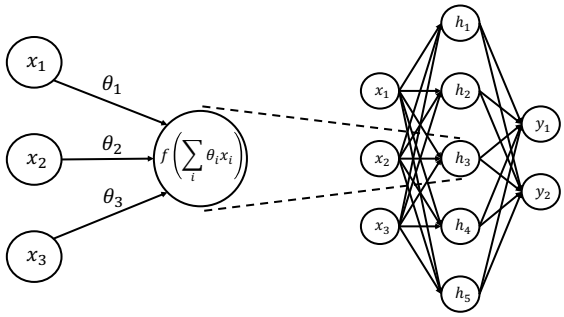


Figure 1: Example of a neural network with 3 inputs, 5 hidden neurons and 2 outputs. Each neuron computes a linear function of its inputs, by multiplying each input by a parameter. The result is then passed through an activation function.

performance. For example, we may have a dataset with N images $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_N$ and corresponding labels $\mathbf{Y} = y_1, ..., y_N$ and the goal may be to predict the label y_i as a function of the image $\mathbf{x}_i$. The loss for the complete dataset can then be computed as the average over the loss $\mathcal{L}(\boldsymbol{\theta}, \mathbf{x}_i, y_i)$ for N individual datapoints in the dataset:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\boldsymbol{\theta}, \mathbf{x}_i, y_i). \tag{2}$$

As an example, the neural network may output a vector of probabilities $\boldsymbol{p}$ over C different classes, and the loss function may be the *cross-entropy loss*

$$\mathcal{L}(\boldsymbol{\theta}, \mathbf{x}_i, y_i) = -\sum_{j=1}^{C} \mathbb{1}_{\{y_i=j\}} \log p_{ij}|_{\mathbf{p}_i = f_{\boldsymbol{\theta}}(\mathbf{x}_i)}. \tag{3}$$

We may write $y_{ij} = \mathbb{1}_{\{y_i=j\}}$, such that $\mathbf{y}_i = (y_{i0}, ..., y_{iC})$ is a vector which we call the *one-hot* encoding of the label, and equation 3 is the cross-entropy between the ground-truth label vector $\mathbf{y}_i$ and the prediction $\mathbf{p}_i$. The cross-entropy is minimized (and equal to zero) if $\mathbf{p}_i = \mathbf{y}_i$, so minimizing the cross-entropy loss results in maximizing the 'similarity' between the predictions and ground-truth labels (the data). We can minimize the loss using *gradient descent* (see Figure 2), where a small step in the direction opposite to the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y})$ is taken to decrease the loss, using step size or *learning rate* η:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}). \tag{4}$$

The gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L} = (\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, ...)$ is the vector of all partial derivatives with respect to each of the parameters of the vector $\boldsymbol{\theta}$. As the neural network is a composition of many simple functions, the gradient for the current set of parameters $\boldsymbol{\theta}$ can be computed efficiently using the *chain rule*, which is facilitated using deep learning or *automatic differentiation* frameworks such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016) or JAX (Bradbury et al., 2018). Often, the dataset is too large to compute the gradient for the complete dataset, and we instead use a randomly sampled subset of the data, also called a *minibatch*. The algorithm is then referred to as *stochastic gradient descent* (SGD), as the minibatch gradient is a stochastic estimate of the true gradient. For more detailed background on deep learning, we refer to the deep learning book (Goodfellow et al., 2016).

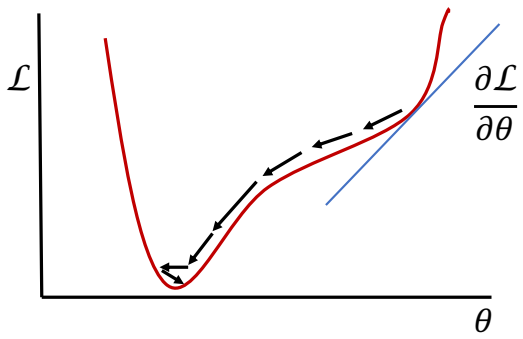


Figure 2: Gradient descent minimizes the loss function by taking small steps in the direction opposite of the gradient (which is positive as indicated by the blue line's slope).

2.2.2 Learning with graph-structured data

Whereas deep neural networks were originally developed for *grid-structured* data, such as images (which are grids of *pixels*), or *sequentially structured* data, such as natural language sentences, modern deep learning applications involve other structures of data, especially represented in the form of *graphs*. A graph consists of a set of *nodes* and a set of *edges* connecting those nodes. Graphs can represent a variety of objects, such as social networks, molecular data, or road networks. As such, recently a lot of research has been devoted to the development of efficient *graph neural networks* (GNNs), that can handle graph-structured data in a principled manner (Bronstein et al., 2021).

Like normal neural networks, GNNs consist of multiple layers, where each layer updates the hidden representation of the data. Similar to how *convolutional* networks have different representations for different spatial locations (Goodfellow et al., 2016), graph neural networks track different representations for different nodes, and in each layer, the representation for each node is obtained by propagating information from other nodes in the local graph neighborhood. This process is sometimes referred to as *neural message passing* (see Figure 3). As the information that is propagated is represented by a neural network function that can be learned, the overall model can learn to propagate relevant information between nodes, to make predictions for individual nodes based on their graph context. When a global prediction is required for the complete graph, the individual node representations can be aggregated (e.g. by taking the average), and the result can be used to make a global prediction. Based on the node representations, we may also make predictions for individual nodes (e.g. predict which node should be the next to visit when constructing a TSP tour, see Chapter 3), or edges (e.g. whether an edge should be in the TSP tour, see Chapter 4). When making predictions for edges, we may represent an edge by the concatenation of the representations of its two adjacent nodes, or we may compute explicit internal representations for edges in each layer.

A simple graph neural network may compute the information it receives from its neighbors simply by averaging over the different neighbors. This limits the expressivity of the model as the same amount of information is propagated from all neighbors, which is inefficient if some neighbors are more relevant than others.

To address this problem, we may use *attention* (Bahdanau et al., 2015), which allows a node to gather different amounts of information from different neighbors. The *graph attention network* (GAT) (Velickovic et al., 2018) has become a popular neural network model for learning on graphs using attention. It is closely related to the *Transformer* model (Vaswani et al., 2017), which can be seen as a variant of the GAT on a fully connected graph. The Transformer model has become the basis of many state-of-the-art deep neural network model architectures, and as such has enabled a boost in performance on a wide variety of tasks, including language modelling, specifically machine translation (Vaswani et al., 2017), image recognition (Dosovitskiy et al., 2021), and many more. It is also the basis for the *attention model* introduced in Chapter 3. For an overview of machine learning with graph-structured data, we refer to the *geometric deep learning* book (Bronstein et al., 2021).

2.2.3 Reinforcement learning

Whereas supervised learning, i.e. learning from labeled data, is the primary paradigm used in deep learning, there is an additional growing interest in *unsupervised learning* (learning without labels) and *reinforcement learning* (RL) (Sutton and Barto, 2018), which is learning from feedback by acting in an environment. In this section, we focus on RL, which, in principle, allows an *agent* to learn to solve optimization problems, solely from feedback on the quality (i.e. the objective value) of solutions proposed by the agent. Often, an RL problem is formulated as a *Markov decision process* (MDP), where an agent needs to define a *policy*, specifying how to select an *action* given that the agent is in a certain *state*. The MDP, which can be seen as the *environment*, then determines how the state transitions into a new state, where the agent should select the next action, etc. At any time, an agent may receive a *reward* following the selection of an action in a certain state, and the goal is to maximize the cumulative reward over time, which is called the *return*. In some cases, the agent does not observe the full state of the environment, but only gets a particular observation, resulting in a partially observable MDP (POMDP). As a famous example, an agent learned to play Atari games using the raw screen pixels as observation and the game score as reward (Mnih et al., 2015).

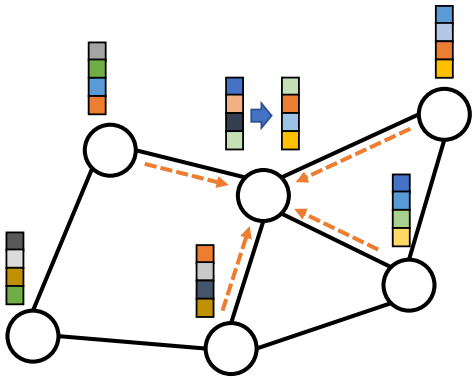


Figure 3: Message passing in a graph neural network: the representation of a node gets updated using information from the other nodes.

Generally, RL methods can be divided into *value* based and *policy* based methods. Value based methods focus on estimating (learning) the value of taking a certain action in a certain state, where the value is the expected *return*: the expected total future reward. Knowing these *action-values*, the optimal policy is to select the action with the highest value for each state. While conceptually simple, the disadvantage of value-based methods is that it may be difficult to learn or estimate the value of taking a certain action in a certain state, and it is not possible to learn a *stochastic policy* which would take different actions with certain probabilities.

As an alternative, *policy* based methods directly learn a policy. A popular policy based algorithm is REINFORCE (Williams, 1992), which, in simple terms, can be explained as an algorithm that samples actions, and increases the probability of taking them when a positive return is observed, and vice versa. REINFORCE is a very generally applicable method, as it makes few assumptions about the environment it is learning from. In general, we consider *discrete* random variables, or actions (in RL terminology), but REINFORCE is applicable with continuous random variables as well. In the case of a discrete random variable, we can easily derive the REINFORCE gradient estimator that can be used to optimize a policy using SGD. Let $p_{\boldsymbol{\theta}}(a|s)$ be the stochastic policy, parameterized by θ, that specifies the probability for an action a given a state s, and let $f(s,a)$ be the objective function that we are optimizing, e.g. the return obtained from the RL environment. Then:

$$\begin{aligned}
\nabla \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[f(s,a)] &= \nabla \sum_a p_{\boldsymbol{\theta}}(a|s) f(s,a) \\
&= \sum_a \nabla p_{\boldsymbol{\theta}}(a|s) f(s,a) \\
&= \sum_a p_{\boldsymbol{\theta}}(a|s) \frac{\nabla p_{\boldsymbol{\theta}}(a|s)}{p_{\boldsymbol{\theta}}(a|s)} f(s,a) \\
&= \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)} \left[\frac{\nabla p_{\boldsymbol{\theta}}(a|s)}{p_{\boldsymbol{\theta}}(a|s)} f(s,a) \right] \\
&= \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)} \left[\nabla \log p_{\boldsymbol{\theta}}(a|s) f(s,a) \right]. \qquad (5)
\end{aligned}$$

For continuous variables, some more mathematical care is necessary to move the derivative (gradient) inside the summation (which becomes an integral), but the derivation is similar. The REINFORCE estimator uses a sample $a \sim p_{\boldsymbol{\theta}}(a|s)$ to estimate the gradient, which is expressed in equation 5 as an expectation, as

$$\nabla \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[f(s,a)] \approx \nabla \log p_{\boldsymbol{\theta}}(a|s) f(s,a). \qquad (6)$$

While REINFORCE is quite general, e.g. f can be a black-box function that does not even need to be differentiable, it is a high variance estimator which makes learning (optimizing the parameters θ) slow. This variance can be reduced by including a *baseline* to which the value $f(s,a)$ (the return) is compared. Note that for any constant b or state-dependent function $b(s)$ that does not depend on a, we have $\nabla \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[b(s)] = \nabla b(s) = 0$, such that we can subtract this baseline from $f(s,a)$:

$$\nabla \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[f(s,a)] = \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)} \left[\nabla \log p_{\boldsymbol{\theta}}(a|s) (f(s,a) - b(s)) \right]. \qquad (7)$$

A well-chosen baseline reduces the variance of the estimator, and therefore produces more stable gradient estimates, which can be combined with a higher learning rate to improve the speed of learning/optimizing parameters using gradient descent. In this thesis, we consider different candidates for a baseline, e.g. the *greedy baseline* in Chapter 3 and the multi-sample estimator with *leave-one-out* baseline (with and without replacement) in Chapter 6.

REINFORCE is sometimes also called the *score function* estimator, as it uses the score function $\nabla \log p_{\boldsymbol{\theta}}(a|s)$ to estimate the effect of changing the parameters θ of the distribution $p_{\boldsymbol{\theta}}(a|s)$ on the expectation $\mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[f(s,a)]$. For continuous distributions, we may use the *reparameterization trick* (Kingma and Welling, 2014; Rezende et al., 2014) as an alternative, if the function $f(s,a)$ is differentiable, and the action a can be written as a differentiable function $a_{\boldsymbol{\theta}}(s,\epsilon)$ of noise ϵ from a parameterless distribution (e.g. standard uniform) and the state:

$$\nabla \mathbb{E}_{a\sim p_{\boldsymbol{\theta}}(a|s)}[f(s,a)] = \nabla \mathbb{E}_{\epsilon\sim p(\epsilon)}[f(s,a_{\boldsymbol{\theta}}(s,\epsilon))] = \mathbb{E}_{\epsilon\sim p(\epsilon)}[\nabla f(s,a_{\boldsymbol{\theta}}(s,\epsilon))]. \quad (8)$$

We can take a sample $\epsilon \sim p(\epsilon)$ to estimate this expectation, where the gradient $\nabla_{\boldsymbol{\theta}} f(s,a_{\boldsymbol{\theta}}(s,\epsilon))$ can then simply be computed with the chain rule.

Discrete random variables can also be reparameterized, but this does not result in a usable gradient estimator. For example, consider a Bernoulli variable with success probability p, which we can reparameterize in terms of standard noise $\epsilon \sim \text{Uniform}(0,1)$ as $\mathbb{1}\{\epsilon \leq p\}$. The problem is that the indicator function $\mathbb{1}\{\epsilon \leq p\}$ has gradient 0 almost everywhere and is not continuous at the point $\epsilon = p$, which makes the second step (swapping gradient and expectation) in equation 8 invalid. Informally, we say that such a reparameterization function is not differentiable. As a solution, it has been proposed to use relaxations of discrete variables, based on the *Gumbel-max trick* (Maddison et al., 2016; Jang et al., 2016).

2.2.4 The Gumbel-max trick

The Gumbel-max trick is a reparameterization trick (i.e. a way to sample as a deterministic function of standard random noise) for categorical variables, which works as follows. Let D be a set of categories, and let $\phi_i, i \in D$ be *unnormalized log-probabilities* for categories $i \in D$. Let $g_i \sim \text{Gumbel}(0), i \in D$ be i.i.d. standard *Gumbel* variables. Then for any subset $B \subseteq D$ it holds that (Maddison et al., 2014):

$$\max_{i\in B} G_{\phi_i} \sim \text{Gumbel}\left(\log \sum_{i\in B} \exp \phi_i\right), \quad (9)$$

$$\arg\max_{i\in B} G_{\phi_i} \sim \text{Categorical}\left(\frac{\exp \phi_i}{\sum_{i'\in B} \exp \phi_{i'}}, i \in B\right). \quad (10)$$

As a result of equation 10, we can draw a sample from a categorical distribution by taking the logarithm of the probabilities, perturbing them independently by adding Gumbel noise, and returning the category corresponding to the largest *perturbed*

log-probability. In the relaxation proposed by Maddison et al. (2016) and Jang et al. (2016), known as the *Gumbel-Softmax* or *Concrete* distribution, the non-differentiable arg max in equation 10 is replaced by a differentiable *softmax*, as an approximation which is differentiable. This distribution, and the resulting estimator when combining with the reparameterization trick (equation 8), has quickly become popular in the machine learning community, despite the bias resulting from the softmax relaxation (Huijben et al., 2022). In this thesis, we do not use such biased estimators, but we do build on the Gumbel-max trick extensively, especially in Chapter 5 where we develop *ancestral Gumbel-top-k sampling*.

2.3 MACHINE LEARNING & OPTIMIZATION

As we discussed before, there are two paradigms that combine machine learning and optimization: *predict, then optimize* and *learning to optimize*.

PREDICT, THEN OPTIMIZE The *predict, then optimize* framework (Elmachtoub and Grigas, 2021) uses machine learning models to make predictions about uncertain quantities and uses these in optimization algorithms. We can therefore also say this is about learning *what* to optimize. An example is the prediction of driving times for use in a routing algorithm, which may vary during the day. If we predict a single value, which is then considered as 'truth' by the optimization algorithm, this is called a *point* prediction. Clearly, such a point prediction can be wrong and an alternative is to predict a *distribution* or a *confidence interval*, which allows the machine learning model to express uncertainty in the prediction. The optimization algorithm can then take this into account if desired, e.g. through *robust optimization* (Ben-Tal and Nemirovski, 2002; Bertsimas et al., 2011), which aims to find solutions which are robust with respect to the uncertainty, or *stochastic optimization* (Heyman and Sobel, 2004), which aims to optimize the objective in expectation.

LEARNING TO OPTIMIZE *Learning to optimize* is the paradigm considered in this thesis, where machine learning is used to learn or improve an optimization algorithm, i.e. to learn *how* to optimize. Within this paradigm, we can put approaches on a scale that varies between *end-to-end* machine learning approaches, where a machine learning model is trained to directly produce a solution to an optimization problem, and *hybrid* approaches, where a machine learning model is used alongside or within an optimization algorithm (Bengio et al., 2021). Chapter 3 presents an example of an end-to-end approach, whereas Chapter 4 illustrates an example of a hybrid algorithm. Recently presented end-to-end approaches are often *constructive* approaches which are conceptually similar to the attention model presented in Chapter 3. On the other hand, hybrid approaches can be roughly divided into using machine learning to improve heuristics within approximate *search methods*, which we call the *learning to search* paradigm (Chen and Tian, 2019; Hottung and Tierney, 2020), and using machine learning to improve *exact algorithms*, where *learning to*

branch (Khalil et al., 2016; Gasse et al., 2019; Nair et al., 2020) is the most prominent and general example. When using machine learning to improve exact algorithms, internal decisions of the solver are guided by a machine learning model, and such approaches can then be turned into heuristics, for example by limiting the runtime of the algorithm. Chapter 4 can also be seen as an example of using machine learning to improve an exact algorithm, in our case dynamic programming.

2.3.1 Challenges in machine learning for combinatorial optimization

Using machine learning, especially deep learning, in the learning to optimize paradigm to learn algorithms for combinatorial optimization presents a number of challenges.

GRADIENT BASED TRAINING WITH DISCRETE (NON-DIFFERENTIABLE) OUTPUTS First of all, the training of deep learning models using gradient descent requires the model, ultimately a mathematical function, to be differentiable with respect to its parameters. This is not the case for a model with discrete outputs, which has a gradient of zero almost everywhere. The solution is to consider a *stochastic* model, which defines a distribution over discrete outputs, and to optimize performance on the task *in expectation*, for example using REINFORCE (see Section 2.2.3). While this makes the gradient well-defined, it is still intractable to compute if the expectation is over a combinatorial domain that grows exponentially with the size of the input. The common solution is to *estimate* the gradient using samples, adding a second source of stochasticity (besides sampling the minibatch) in the *stochastic gradient descent* (SGD) algorithm introduced in Section 2.2.1. As the performance of the SGD algorithm depends on the variance of the gradient estimates, Part ii of this thesis is devoted to sampling techniques and statistical estimators that reduce this variance when sampling from combinatorial spaces.

DEFINING THE RIGHT MODEL ARCHITECTURE Machine learning models for combinatorial optimization should take as input *structured data* (such as graphs), and predict structured outputs (solutions) as well. This requires the right (graph) neural network architecture, for example to exploit symmetries (see Section 2.2.2). There are different options to facilitate output structure. For example, when predicting a tour, we may predict one step at a time (see Chapter 3), or we may directly predict which edges should be in the tour all at once, and use a post-processing step to create valid tour (see Chapter 4). Another consideration is what to predict exactly. For example, predicting *probabilities* (e.g. where to move next in a routing problem) rather than values (e.g. the expected total tour length or cost) may help generalization, as probabilities are invariant to factors such as scaling of the data. This aligns with our preference for *policy based* RL (Section 2.2.3): predicting actions rather than values. In any case, even with a suitable neural network architecture, the model will be sensitive to input value scaling or rotation (e.g. coordinates or distances with routing problems), although this can, to some extent, be accounted for by normalizing the input (which is a good idea in general (Stöttner, 2019)).

DEFINING AN EFFICIENT TRAINING PARADIGM The third challenge relates to how to train a machine learning model for combinatorial optimization. The most general method is to use reinforcement learning (RL, Section 2.2.3), in which a policy is trained *end-to-end* using rewards obtained from the objective function. The downside is that RL requires many samples to learn, and each individual sample may be computationally costly, especially if it considers a *hybrid algorithm* which internally uses expensive computations as well. Therefore, one requires creative training paradigms that exploit existing knowledge. For example, if we can find good or even optimal solutions using an existing (but computationally costly) algorithm, we can use this to create a training dataset of good quality solutions *once*, which can then be used for many training iterations in a supervised learning setting (see Chapter 4). This is more efficient than using RL, and especially useful while designing the model, which requires a lot of testing. AlphaGo (Silver et al., 2017) even iterates this idea for better results: using *Monte Carlo tree search* (MCTS) to build a training dataset with good moves, a model is trained to improve the MCTS algorithm, such that a better dataset can be generated to train an even stronger model, etc. Nonetheless, even with a clever training paradigm, training is extremely costly. However, for a specific problem (and distribution of instances), training is only required once, after which the trained model can be used to solve many new problem instances.

THE MODEL SHOULD BE WORTH IT(S COMPUTATION) Even if the cost of training a model is acceptable, it is non-trivial that the model is also beneficial when solving a new problem instance *at test time*, as the impact of the model must be 'worth' the computation it requires. Since a combinatorial optimization problem has a finite solution space, *any* reasonable optimization algorithm will find the optimal solution when given infinite computation time, especially exhaustive (or even random) search. Therefore, a better (machine-learning based) algorithm should either find the optimal solution with less computation, or find a better solution within a given time limit. In other words, we should *always* evaluate algorithms in terms of both quality (cost of solution) and computation (see Figure 4). While training uses the most computation, it is still costly to use a large DNN model to make predictions. Therefore, if we use predictions in an optimization algorithm, we *invest* valuable computation time, which should *pay off*, e.g. by finding better solutions quickly. Otherwise, we may better use the same computation as additional search budget in a (highly optimized) search algorithm. The most successful machine learning based approaches thus *maximize the impact* of predictions while *minimizing the computation* (flops) invested for making those predictions. *Count your flops and make them count!*

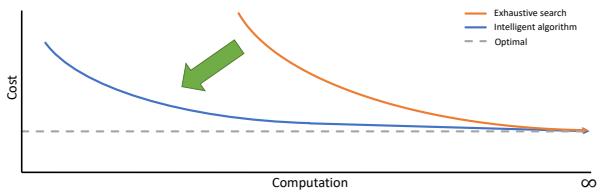


Figure 4: Performance (cost of solution) of an algorithm solving a problem, as a function of time/computation. Given infinite computation, all algorithms are equal so an algorithm should always be evaluated w.r.t. both performance and computation.

Part I

ROUTING AND REINFORCEMENT LEARNING

SUMMARY OF PART I

In Part i of this thesis, we focus on routing problems. We propose different models and methods for solving such problems using techniques from machine learning, especially deep (reinforcement) learning. This is challenging as we require models that can handle structured input data, such as TSP or VRP instances, as well as produce structured output in the form of feasible solutions.

In Chapter 3, we propose the *attention model*, which is a flexible model that can be applied to different routing problems. It treats the input as a graph of nodes (i.e. locations to visit), which gets processed by an *encoder*, resulting in abstract representations for each node. These are then used by the *decoder*, which sequentially selects the next node to visit, based on context information such as the current node and destination location. Feasibility of the sequentially constructed solution is ensured by a *masking* procedure that prevents the model from selecting nodes that would result in a violation of constraints, e.g. by visiting the same node twice or exceeding the vehicle capacity. The model is trained using a reinforcement learning (RL) algorithm known as REINFORCE (Williams, 1992), where we develop a variant that uses a *greedy rollout baseline*, which directs the model towards progress by comparing sampled solutions against greedy solutions, to determine the relative quality. While the resulting model does not outperform state-of-the-art algorithms for individual problems, it provides reasonable solutions for different problems quickly, and significantly outperforms classic construction heuristics which are of similar structure, as well as previous learning-based approaches.

Whereas the attention model introduced in Chapter 3 is flexible, and many variants of it have been applied in follow-up work (Peng et al., 2019; Xin et al., 2020; Kwon et al., 2020; Ma et al., 2021), it has two important limitations. First, the sequential construction requires the model to be evaluated at every step, which is computationally costly if one wants to combine it with search, e.g. generating many solutions to select the best one in order to improve performance of the model. Second, while the masking scheme is often effective for enforcing feasibility of the solutions, some problems have more complex constraints (such as time windows for routing problems), which cannot simply be enforced by a masking procedure as the construction can run into dead ends. As a solution to these challenges, in Chapter 4, we present *deep policy dynamic programming* (DPDP). DPDP uses a different neural network model, proposed by Joshi et al. (2019a), which does not require sequential construction, but outputs a *heatmap* with probabilities that an edge is 'good' (likely to be part of optimal solution) for all edges in the graph *at once*. This can then be combined with dynamic programming over a restricted space of promising solutions, identified by the neural network, which enables to find good solutions even in the presence of complex constraints. This is an example of a hybrid algorithm that uses machine learning to improve a 'classic' optimization algorithm: dynamic programming. Results show that this significantly improves the performance of dynamic programming algorithms for vehicle routing problems.

3 ATTENTION, LEARN TO SOLVE ROUTING PROBLEMS!

The recently presented idea to learn heuristics for combinatorial optimization problems is promising as it can save costly development. However, to push this idea towards practical implementation, we need better models and better ways of training. We contribute in both directions: we propose a model based on attention layers with benefits over the pointer network and we show how to train this model using REINFORCE with a simple baseline based on a deterministic greedy rollout, which we find is more efficient than using a value function. We significantly improve over recent learned heuristics for the travelling salesman problem (TSP), getting close to optimal results for problems up to 100 nodes. With the same hyperparameters, we learn strong heuristics for two variants of the vehicle routing problem (VRP), the orienteering problem (OP) and (a stochastic variant of) the prize collecting TSP (PCTSP), outperforming a wide range of baselines and getting results close to highly optimized and specialized algorithms.

3.1 INTRODUCTION

Imagine yourself travelling to a scientific conference. The field is popular, and surely you do not want to miss out on anything. You have selected several posters you want to visit, and naturally you must return to the place where you are now: the coffee corner. In which order should you visit the posters, to minimize your time walking around? This is the travelling scientist problem (TSP).

You realize that your problem is equivalent to the travelling salesman problem (conveniently also TSP). This seems discouraging as you know the problem is (NP-)hard (Garey and Johnson, 1979). Fortunately, complexity theory analyzes the worst case, and your Bayesian view considers this unlikely. In particular, you have a strong prior: the posters will probably be laid out regularly. You want a special algorithm that solves not any, but *this* type of problem instance. You have some months left to prepare. As a machine learner, you wonder whether your algorithm can be learned?

MOTIVATION Machine learning algorithms have replaced humans as the engineers of algorithms to solve various tasks. A decade ago, computer vision algorithms used hand-crafted features but today they are learned *end-to-end* by deep neural networks (DNNs). DNNs have outperformed classic approaches in speech recognition, machine translation, image captioning and other problems, by learning from data (LeCun et al., 2015). While DNNs are mainly used to make *predictions*, reinforcement learning (RL) has enabled algorithms to learn to make *decisions*, ei-

ther by interacting with an environment, e.g. to learn to play Atari games (Mnih et al., 2015), or by inducing knowledge through look-ahead search: this was used to master the game of Go (Silver et al., 2017).

The world is not a game, and we desire to train models that make decisions to solve real problems. These models must learn to select good solutions for a problem from a combinatorially large set of potential solutions. Classically, approaches to this problem of *combinatorial optimization* can be divided into *exact methods*, that guarantee finding optimal solutions, and *heuristics*, that trade off optimality for computational cost, although exact methods can use heuristics internally and vice versa. Heuristics are typically expressed in the form of rules, which can be interpreted as policies to make decisions. We believe that these policies can be parameterized using DNNs, and be trained to obtain new and stronger algorithms for many different combinatorial optimization problems, similar to the way DNNs have boosted performance in the applications mentioned before. In this chapter, we focus on routing problems: an important class of practical combinatorial optimization problems.

The promising idea to learn heuristics has been tested on the TSP (Bello et al., 2016). In order to push this idea, we need better models and better ways of training. Therefore, we propose to use a powerful model based on attention and we propose to train this model using REINFORCE (Williams, 1992) with a simple but effective greedy rollout baseline. The goal of our method is not to outperform a non-learned, specialized TSP algorithm such as Concorde (Applegate et al., 2006). Rather, we show the flexibility of our approach on multiple (routing) problems of reasonable size, with *a single set of hyperparameters*. This is important progress towards the situation where we can learn strong heuristics to solve a wide range of different practical problems for which no good heuristics exist.

3.2 RELATED WORK

The application of neural networks (NNs) for optimizing decisions in combinatorial optimization problems dates back to Hopfield and Tank (1985), who applied a Hopfield-network for solving small TSP instances. NNs have been applied to many related problems (Smith, 1999), although in most cases in an *online* manner, starting 'from scratch' and 'learning' a solution for every instance. More recently, (D)NNs have also been used *offline* to learn about an entire class of problem instances.

Vinyals et al. (2015a) introduce the *pointer network* (PN) as a model that uses attention to output a permutation of the input, and train this model offline to solve the (Euclidean) TSP, supervised by example solutions. Upon test time, their beam search procedure filters invalid tours. Bello et al. (2016) introduce an actor-critic algorithm to train the PN without supervised solutions. They consider each instance as a training sample and use the cost (tour length) of a sampled solution for an unbiased Monte Carlo estimate of the policy gradient. They introduce extra model depth in the decoder by an additional *glimpse* (Vinyals et al., 2016) at the embed-

dings, masking nodes already visited. For small instances ($n = 20$), they get close to the results by Vinyals et al. (2015a), they improve for $n = 50$ and additionally include results for $n = 100$. Nazari et al. (2018) replace the LSTM encoder of the PN by element-wise projections, such that the updated embeddings after state-changes can be effectively computed. They apply this model on the vehicle routing problem (VRP) with split deliveries and a stochastic variant.

Dai et al. (2017) do not use a separate encoder and decoder, but a single model based on graph embeddings. They train the model to output the *order* in which nodes are *inserted* into a partial tour, using a helper function to insert at the best possible location. Their 1-step DQN (Mnih et al., 2015) training method trains the algorithm per step and incremental rewards provided to the agent at every step effectively encourage greedy behavior. As mentioned in their appendix, they use the negative of the reward, which combined with discounting encourages the agent to insert the farthest nodes first, which is known to be an effective heuristic (Rosenkrantz et al., 2009).

Nowak et al. (2017) train a graph neural network (GNN) in a supervised manner to directly output a tour as an adjacency matrix, which is converted into a feasible solution by a beam search. The model is non-autoregressive, so cannot condition its output on the partial tour and the authors report an optimality gap of 2.7% for $n = 20$, worse than autoregressive approaches mentioned in this section. Kaempfer and Wolf (2018) train a model based on the Transformer architecture (Vaswani et al., 2017) that outputs a fractional solution to the multiple TSP (mTSP). The result can be seen as a solution to the linear relaxation of the problem and they use a beam search to obtain a feasible integer solution.

Independently of our work, Deudon et al. (2018) presented a similar model for TSP using attention. They show performance can improve using 2OPT local search, but do not show benefit of their model in direct comparison to the PN. We use a different decoder and improved training algorithm, both contributing to significantly improved results, *without* 2OPT and additionally show application to different problems. For a full discussion of the differences, we refer to Appendix A.2.4.

3.3 ATTENTION MODEL

We define our model, the *attention model*, in terms of the TSP. For other problems, the model is the same but the input, mask and decoder context need to be defined accordingly, which is discussed in Appendix A. We define a problem instance s as a graph with n nodes, where node $i \in \{1, \dots, n\}$ is represented by features $\mathbf{x}_i$. For TSP, $\mathbf{x}_i$ is the coordinate of node i and the graph is fully connected (with self-connections) but in general, the model can be considered a graph attention network (Velickovic et al., 2018) and take graph structure into account by a masking procedure (see Appendix A.1). We define a solution (tour) $\boldsymbol{\pi} = (\pi_1, \dots, \pi_n)$ as a permutation of the nodes, so $\pi_t \in \{1, \dots n\}$ and $\pi_t \neq \pi_{t'} \forall t \neq t'$. Our attention based encoder-

decoder model defines a stochastic policy $p(\boldsymbol{\pi}|s)$ for selecting a solution $\boldsymbol{\pi}$ given a problem instance s. It is factorized and parameterized by $\boldsymbol{\theta}$ as

$$p_\theta(\boldsymbol{\pi}|s) = \prod_{t=1}^{n} p_{\boldsymbol{\theta}}(\pi_t|s, \boldsymbol{\pi}_{1:t-1}). \tag{11}$$

The encoder produces embeddings of all input nodes. The decoder produces the sequence $\boldsymbol{\pi}$ of input nodes, one node at a time. It takes as input the encoder embeddings and a problem specific mask and context. For TSP, when a partial tour has been constructed, it cannot be changed and the 'remaining' problem is to find a path from the last node, through all unvisited nodes, to the first node. The order and coordinates of other nodes already visited are irrelevant. To know the first and last node, the decoder context consists (next to the graph embedding) of embeddings of the first and last node. Similar to Bello et al. (2016), the decoder observes a mask to know which nodes have been visited.

3.3.1 Encoder

The encoder that we use (Figure 5) is similar to the encoder used in the Transformer architecture by Vaswani et al. (2017), but we do not use positional encoding such that the resulting node embeddings are invariant to the input order. From the d_{x}-dimensional input features $\mathbf{x}_i$ (for TSP $d_{\mathrm{x}} = 2$), the encoder computes initial d_{h}-dimensional node embeddings $\mathbf{h}_i^{(0)}$ (we use $d_{\mathrm{h}} = 128$) through a learned linear projection with parameters W^{x} and $\mathbf{b}^{\mathrm{x}}$: $\mathbf{h}_i^{(0)} = W^{\mathrm{x}}\mathbf{x}_i + \mathbf{b}^{\mathrm{x}}$. The embeddings are updated using N attention layers, each consisting of two sublayers. We denote with $\mathbf{h}_i^{(\ell)}$ the node embeddings produced by layer $\ell \in \{1, .., N\}$. The encoder computes an aggregated embedding $\bar{\mathbf{h}}^{(N)}$ of the input graph as the mean of the final node embeddings $\mathbf{h}_i^{(N)}$: $\bar{\mathbf{h}}^{(N)} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{h}_i^{(N)}$. Both the node embeddings $\mathbf{h}_i^{(N)}$ and the graph embedding $\bar{\mathbf{h}}^{(N)}$ are used as input to the decoder.

ATTENTION LAYER Following the Transformer architecture (Vaswani et al., 2017), each attention layer consist of two sublayers: a multi-head attention (MHA) layer that executes message passing between the nodes and a node-wise fully connected feed-forward (FF) layer. Each sublayer adds a skip-connection (He et al., 2016b) and batch normalization (BN) (Ioffe and Szegedy, 2015) (which we found to work better than layer normalization (Ba et al., 2016)):

$$\hat{\mathbf{h}}_i = \mathrm{BN}^{\ell}\left(\mathbf{h}_i^{(\ell-1)} + \mathrm{MHA}_i^{\ell}\left(\mathbf{h}_1^{(\ell-1)}, \ldots, \mathbf{h}_n^{(\ell-1)}\right)\right) \tag{12}$$

$$\mathbf{h}_i^{(\ell)} = \mathrm{BN}^{\ell}\left(\hat{\mathbf{h}}_i + \mathrm{FF}^{\ell}(\hat{\mathbf{h}}_i)\right). \tag{13}$$

The layer index ℓ indicates that the layers do *not* share parameters. The MHA sublayer uses $M = 8$ heads with dimensionality $\frac{d_h}{M} = 16$, and the FF sublayer has one hidden (sub)sublayer with dimension 512 and ReLu activation. See Appendix A.1 for details.

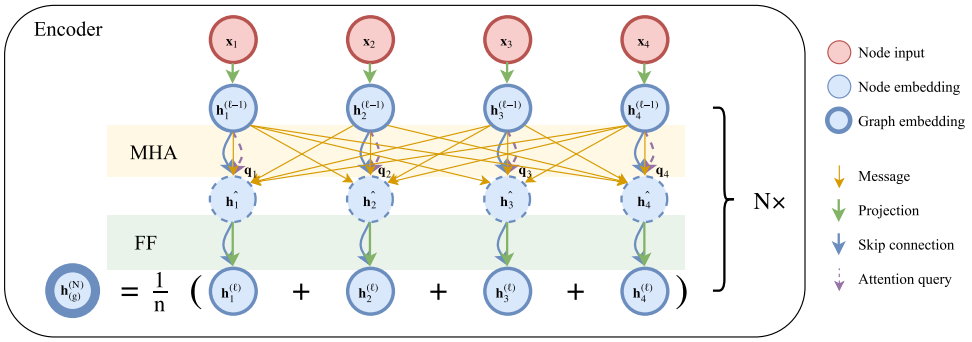


Figure 5: Attention based encoder. Input nodes are embedded and processed by N sequential layers, each consisting of a multi-head attention (MHA) and node-wise feed-forward (FF) sublayer. The graph embedding is computed as the mean of node embeddings. Best viewed in color.

3.3.2 Decoder

Decoding happens sequentially, and at timestep $t \in \{1, \dots n\}$, the decoder outputs the node π_t based on the embeddings from the encoder and the outputs $\pi_{t'}$ generated at time $t' < t$. During decoding, we augment the graph with a special *context node* (c) to represent the decoding context. The decoder computes an attention (sub)layer on top of the encoder, but with messages only to the context node for efficiency.[1] The final probabilities are computed using a single-head attention mechanism. See Figure 6 for an illustration of the decoding process.

CONTEXT EMBEDDING The context of the decoder at time t comes from the encoder and the output up to time t. As mentioned, for the TSP it consists of the embedding of the graph, the previous (last) node π_{t-1} and the first node π_1. For $t = 1$ we use learned d_h-dimensional parameters $\mathbf{v}^l$ and $\mathbf{v}^f$ as input placeholders:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \mathbf{h}_{\pi_1}^{(N)}\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{v}^l, \mathbf{v}^f\right] & t = 1. \end{cases} \tag{14}$$

Here $[\cdot, \cdot, \cdot]$ is the horizontal concatenation operator and we write the $(3 \cdot d_h)$-dimensional result vector as $\mathbf{h}_{(c)}^{(N)}$ to indicate we interpret it as the embedding of the special context node (c) and use the superscript (N) to align with the node embeddings $\mathbf{h}_i^{(N)}$. We could project the embedding back to d_h dimensions, but we absorb this transformation in the parameter W^Q in equation 15.

Now we compute a new context node embedding $\mathbf{h}_{(c)}^{(N+1)}$ using the (M-head) attention mechanism described in Appendix A.1. The keys and values come from the node embeddings $\mathbf{h}_i^{(N)}$, but we only compute a single query $\mathbf{q}_{(c)}$ (per head) from the context node (we omit the (N) for readability):

$$\mathbf{q}_{(c)} = W^Q \mathbf{h}_{(c)} \quad \mathbf{k}_i = W^K \mathbf{h}_i, \quad \mathbf{v}_i = W^V \mathbf{h}_i. \tag{15}$$

1 $n \times n$ attention between all nodes is expensive to compute in every step of the decoding process.

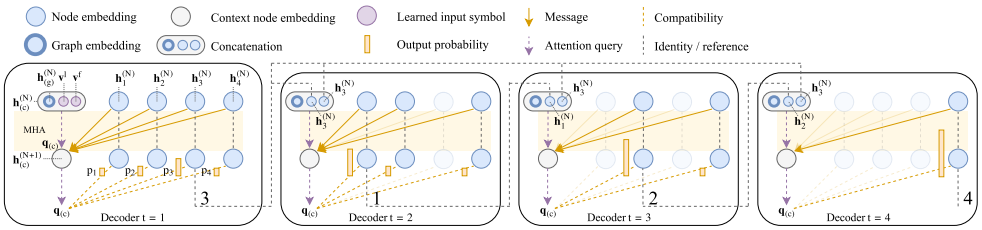


Figure 6: Attention based decoder for the TSP problem. The decoder takes as input the graph embedding and node embeddings. At each time step t, the context consist of the graph embedding and the embeddings of the first and last (previously output) node of the partial tour, where learned placeholders are used if $t = 1$. Nodes that cannot be visited (since they are already visited) are masked. The example shows how a tour $\boldsymbol{\pi} = (3, 1, 2, 4)$ is constructed. Best viewed in color.

We compute the compatibility of the query with all nodes, and mask (set $u_{(c)j} = -\infty$) nodes which cannot be visited at time t. For TSP, this simply means we mask the nodes already visited:

$$u_{(c)j} = \begin{cases} \frac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_\text{k}}} & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{16}$$

Here $d_\text{k} = \frac{d_\text{h}}{M}$ is the query/key dimensionality (see Appendix A.1). Again, we compute $u_{(c)j}$ and $\mathbf{v}_i$ for $M = 8$ heads and compute the final multi-head attention value for the context node using equations (72)–(74) from Appendix A.1, but with (c) instead of i. This mechanism is similar to our encoder, but does not use skip-connections, batch normalization or the feed-forward sublayer for maximal efficiency. The result $\mathbf{h}_{(c)}^{(N+1)}$ is similar to the *glimpse* described by Bello et al. (2016).

CALCULATION OF LOG-PROBABILITIES To compute the output probabilities $p_{\boldsymbol{\theta}}(\pi_t | s, \boldsymbol{\pi}_{1:t-1})$ in equation 11, we add one final decoder layer with a *single* attention head ($M = 1$ so $d_\text{k} = d_\text{h}$). For this layer, we *only* compute the compatibilities $u_{(c)j}$ using equation 16, but following Bello et al. (2016) we clip the result (before masking!) within $[-C, C]$ (C = 10) using tanh:

$$u_{(c)j} = \begin{cases} C \cdot \tanh\left(\frac{\mathbf{q}_{(c)}^T \mathbf{k}_j}{\sqrt{d_\text{k}}}\right) & \text{if } j \neq \pi_{t'} \quad \forall t' < t \\ -\infty & \text{otherwise.} \end{cases} \tag{17}$$

We interpret these compatibilities as unnormalized log-probabilities (logits) and compute the final output probability vector $\mathbf{p}$ using a softmax (similar to equation 72 in Appendix A.1):

$$p_i = p_{\boldsymbol{\theta}}(\pi_t = i | s, \boldsymbol{\pi}_{1:t-1}) = \frac{e^{u_{(c)i}}}{\sum_j e^{u_{(c)j}}}. \tag{18}$$

3.4 REINFORCE WITH GREEDY ROLLOUT BASELINE

Section 3.3 defined our model that given an instance s defines a probability distribution $p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)$, from which we can sample to obtain a solution (tour) $\boldsymbol{\pi}|s$. In order to train our model, we define the loss $\mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}[L(\boldsymbol{\pi})]$: the expectation of the cost $L(\boldsymbol{\pi})$ (tour length for TSP). We optimize $\mathcal{L}$ by gradient descent, using the REINFORCE (Williams, 1992) gradient estimator with baseline $b(s)$:

$$\nabla \mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)}\left[(L(\boldsymbol{\pi}) - b(s)) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)\right]. \tag{19}$$

A good baseline $b(s)$ reduces gradient variance and therefore increases speed of learning. A simple example is an exponential moving average $b(s) = M$ with *decay* β. Here $M = L(\boldsymbol{\pi})$ in the first iteration and gets updated as $M \leftarrow \beta M + (1-\beta)L(\boldsymbol{\pi})$ in subsequent iterations. A popular alternative is the use of a learned value function $\hat{v}(s, \boldsymbol{w})$, sometimes called a *critic*[2], where the parameters $\boldsymbol{w}$ are learned from the observations $(s, L(\boldsymbol{\pi}))$.

We propose to use a rollout baseline in a way that is similar to self-critical training by Rennie et al. (2017), but with periodic updates of the baseline policy. It is defined as follows: $b(s)$ is the cost of a solution from a *deterministic greedy rollout* of the policy defined by the best model so far.

MOTIVATION The goal of a baseline is to estimate the difficulty of the instance s, such that it can relate to the cost $L(\boldsymbol{\pi})$ to estimate the advantage of the solution $\boldsymbol{\pi}$ selected by the model. We make the following key observation: *The difficulty of an instance can (on average) be estimated by the performance of an algorithm applied to it.* This follows from the assumption that (on average) an algorithm will have a higher cost on instances that are more difficult. Therefore we form a baseline by applying (rolling out) the algorithm defined by our model during training. To eliminate variance we force the result to be deterministic by selecting greedily the action with maximum probability.

DETERMINING THE BASELINE POLICY As the model changes during training, we stabilize the baseline by freezing the greedy rollout policy $p_{\boldsymbol{\theta}^{\mathrm{BL}}}$ for a fixed number of steps (every epoch), similar to freezing of the target Q-network in DQN (Mnih et al., 2015). A stronger algorithm defines a stronger baseline, so we compare (with greedy decoding) the current training policy with the baseline policy at the end of every epoch, and replace the parameters $\boldsymbol{\theta}^{\mathrm{BL}}$ of the baseline policy only if the improvement is significant according to a paired t-test ($\alpha = 5\%$), on 10000 separate (evaluation) instances. If the baseline policy is updated, we sample new evaluation instances to prevent overfitting.

2 Formally, in *actor-critic* methods, the critic is a learned value function used to estimate the *return* following an action, but the same value function can also be used as a baseline (Sutton and Barto, 2018).

ANALYSIS With the greedy rollout as baseline $b(s)$, the function $L(\boldsymbol{\pi}) - b(s)$ is negative if the sampled solution $\boldsymbol{\pi}$ is better than the greedy rollout, causing actions to be reinforced, and vice versa. This way the model is trained to improve over its (greedy) self. We see similarities with self-play improvement by AlphaGo (Silver et al., 2017): sampling replaces tree search for exploration and the model is rewarded if it yields improvement ('wins') compared to the best model. Similar to AlphaGo, the evaluation at the end of each epoch ensures that we are always challenged by the best model.

ALGORITHM We use Adam (Kingma and Ba, 2015) as optimizer resulting in Algorithm 1.

Algorithm 1 REINFORCE with Rollout Baseline

Input: number of epochs E, steps per epoch T, batch size B, significance α
Init $\boldsymbol{\theta}$, $\boldsymbol{\theta}^{\text{BL}} \leftarrow \boldsymbol{\theta}$
for epoch $= 1, \dots, E$ **do**
  for step $= 1, \dots, T$ **do**
    $s_i \leftarrow \text{RandomInstance}() \;\; \forall i \in \{1, \dots, B\}$
    $\boldsymbol{\pi}_i \leftarrow \text{SampleRollout}(s_i, p_{\boldsymbol{\theta}}) \;\; \forall i \in \{1, \dots, B\}$
    $\boldsymbol{\pi}_i^{\text{BL}} \leftarrow \text{GreedyRollout}(s_i, p_{\boldsymbol{\theta}^{\text{BL}}}) \;\; \forall i \in \{1, \dots, B\}$
    $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left(L(\boldsymbol{\pi}_i) - L(\boldsymbol{\pi}_i^{\text{BL}})\right) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}_i)$
    $\boldsymbol{\theta} \leftarrow \text{Adam}(\boldsymbol{\theta}, \nabla \mathcal{L})$
  end for
  if $\text{OneSidedPairedTTest}(p_{\boldsymbol{\theta}}, p_{\boldsymbol{\theta}^{\text{BL}}}) < \alpha$ **then**
    $\boldsymbol{\theta}^{\text{BL}} \leftarrow \boldsymbol{\theta}$
  end if
end for

EFFICIENCY Each rollout constitutes an additional forward pass, increasing computation by 50%. However, as the baseline policy is fixed for an epoch, we can sample the data and compute baselines per epoch using larger batch sizes, allowed by the reduced memory requirement as the computations can run in pure inference mode. Empirically we find that it adds only 25% (see Appendix A.2.5), taking up 20% of total time. If desired, the baseline rollout can be computed in parallel (using an additional GPU) such that there is no increase in time per iteration.

3.5 EXPERIMENTS

We focus on routing problems: we consider the TSP, two variants of the VRP, the orienteering problem and the (stochastic) prize collecting TSP. These provide a range of different challenges, constraints and objectives and are *traditionally solved by different algorithms*. For the attention model (AM), we adjust the input, mask, decoder context and objective function for each problem (see Appendix A for details and data generation) and train on problem instances of $n = 20$, 50 and 100 nodes. For all problems, we use *the same hyperparameters*: those we found to work well on TSP.

HYPERPARAMETERS We initialize parameters Uniform$(-1/\sqrt{d}, 1/\sqrt{d})$, with d the input dimension. Every epoch we process 2500 batches of 512 instances (except for VRP with $n = 100$, where we use 2500 $\times$ 256 for memory constraints). For TSP, an epoch takes 5:30 minutes for $n = 20$, 16:20 for $n = 50$ (single GPU 1080Ti) and 27:30 for $n = 100$ (on 2 1080Ti's). We train for 100 epochs using training data generated on the fly. We found training to be stable and results to be robust against different seeds, where only in one case (PCTSP with $n = 20$) we had to restart training with a different seed because the run diverged. We use $N = 3$ layers in the encoder, which we found is a good trade-off between quality of the results and computational complexity. We use a constant learning rate $\eta = 10^{-4}$. Training with a higher learning rate $\eta = 10^{-3}$ is possible and speeds up initial learning, but requires decay (0.96 per epoch) to converge and may be a bit more unstable. See Appendix A.2.5. With the rollout baseline, we use an exponential baseline ($\beta = 0.8$) during the first epoch, to stabilize initial learning, although in many cases learning also succeeds without this 'warmup'. Our code in PyTorch (Paszke et al., 2019) is publicly available.[3]

DECODING STRATEGY AND BASELINES For each problem, we report performance on 10000 test instances. At test time we use *greedy* decoding, where we select the best action (according to the model) at each step, or *sampling*, where we sample 1280 solutions (in $<$ 1s on a single GPU) and report the best. More sampling improves solution quality at increased computation. In Table 1 we compare greedy decoding against baselines that also construct a single solution, and compare sampling against baselines that also consider multiple solutions, either via sampling or (local) search. For each problem, we also report the 'best possible solution': either optimal via Gurobi (Gurobi Optimization, LLC, 2022) (intractable for $n > 20$ except for TSP) or a strong problem specific algorithm.

RUN TIMES Run times are important but hard to compare: they can vary by two orders of magnitude as a result of implementation (Python vs C++) and hardware (GPU vs CPU). We take a practical view and report the time it takes to solve the test set of 10000 instances, either on a single GPU (1080Ti) or 32 instances in parallel on a 32 virtual CPU system (2 $\times$ Xeon E5-2630). This is conservative: our model is parallelizable while most of the baselines are single thread CPU implementations which cannot parallelize when running individually. Also we note that after training our run time can likely be reduced by model compression (Hinton et al., 2015). In Table 1 we do not report running times for the results which were reported by others as they are not directly comparable but we note that in general our model and implementation is fast: for instance Bello et al. (2016) report 10.3s for sampling 1280 TSP solutions (K80 GPU) which we do in less than one second (on a 1080Ti). For most algorithms it is possible to trade off runtime for performance. As reporting full trade-off curves is impractical we tried to pick reasonable spots, reporting the fastest if results were similar or reporting results with different time limits (for example we use Gurobi with time limits as heuristic).

3 https://github.com/wouterkool/attention-learn-to-route

Table 1: Attention model (AM) vs baselines. The gap % is w.r.t. the best value across all methods.

	Method	$n = 20$ Obj.	Gap	Time	$n = 50$ Obj.	Gap	Time	$n = 100$ Obj.	Gap	Time
TSP	Concorde	3.84	0.00%	(1m)	5.70	0.00%	(2m)	7.76	0.00%	(3m)
	LKH3	3.84	0.00%	(18s)	5.70	0.00%	(5m)	7.76	0.00%	(21m)
	Gurobi	3.84	0.00%	(7s)	5.70	0.00%	(2m)	7.76	0.00%	(17m)
	Gurobi (1s)	3.84	0.00%	(8s)	5.70	0.00%	(2m)		-	
	Nearest insertion	4.33	12.91%	(1s)	6.78	19.03%	(2s)	9.46	21.82%	(6s)
	Random insertion	4.00	4.36%	(0s)	6.13	7.65%	(1s)	8.52	9.69%	(3s)
	Farthest insertion	3.93	2.36%	(1s)	6.01	5.53%	(2s)	8.35	7.59%	(7s)
	Nearest neighbor	4.50	17.23%	(0s)	7.00	22.94%	(0s)	9.68	24.73%	(0s)
	Vinyals et al. (gr.)	3.88	1.15%		7.66	34.48%			-	
	Bello et al. (gr.)	3.89	1.42%		5.95	4.46%		8.30	6.90%	
	Dai et al.	3.89	1.42%		5.99	5.16%		8.31	7.03%	
	Nowak et al.	3.93	2.46%			-			-	
	EAN (greedy)	3.86	0.66%	(2m)	5.92	3.98%	(5m)	8.42	8.41%	(8m)
	AM (greedy)	**3.85**	**0.34**%	(0s)	**5.80**	**1.76**%	(2s)	**8.12**	**4.53**%	(6s)
	OR-Tools	3.85	0.37%		5.80	1.83%		7.99	2.90%	
	Chr.f. + 2OPT	3.85	0.37%		5.79	1.65%			-	
	Bello et al. (s.)		-		5.75	0.95%		8.00	3.03%	
	EAN (gr. + 2OPT)	3.85	0.42%	(4m)	5.85	2.77%	(26m)	8.17	5.21%	(3h)
	EAN (sampling)	3.84	0.11%	(5m)	5.77	1.28%	(17m)	8.75	12.70%	(56m)
	EAN (s. + 2OPT)	3.84	0.09%	(6m)	5.75	1.00%	(32m)	8.12	4.64%	(5h)
	AM (sampling)	**3.84**	**0.08**%	(5m)	**5.73**	**0.52**%	(24m)	**7.94**	**2.26**%	(1h)
CVRP	Gurobi	6.10	0.00%			-			-	
	LKH3	6.14	0.58%	(2h)	10.38	0.00%	(7h)	15.65	0.00%	(13h)
	RL (greedy)	6.59	8.03%		11.39	9.78%		17.23	10.12%	
	AM (greedy)	**6.40**	**4.97**%	(1s)	**10.98**	**5.86**%	(3s)	**16.80**	**7.34**%	(8s)
	RL (beam 10)	6.40	4.92%		11.15	7.46%		16.96	8.39%	
	Random CW	6.81	11.64%		12.25	18.07%		18.96	21.18%	
	Random sweep	7.08	16.07%		12.96	24.91%		20.33	29.93%	
	OR-Tools	6.43	5.41%		11.31	9.01%		17.16	9.67%	
	AM (sampling)	**6.25**	**2.49**%	(6m)	**10.62**	**2.40**%	(28m)	**16.23**	**3.72**%	(2h)
SDVRP	RL (greedy)	6.51	4.19%		11.32	6.88%		17.12	5.23%	
	AM (greedy)	**6.39**	**2.34**%	(1s)	**10.92**	**3.08**%	(4s)	**16.83**	**3.42**%	(11s)
	RL (beam 10)	6.34	1.47%		11.08	4.61%		16.86	3.63%	
	AM (sampling)	**6.25**	**0.00**%	(9m)	**10.59**	**0.00**%	(42m)	**16.27**	**0.00**%	(3h)
OP (distance)	Gurobi	5.39	0.00%	(16m)		-			-	
	Gurobi (1s)	4.62	14.22%	(4m)	1.29	92.03%	(6m)	0.58	98.25%	(7m)
	Gurobi (10s)	5.37	0.33%	(12m)	10.96	32.20%	(51m)	1.34	95.97%	(53m)
	Gurobi (30s)	5.38	0.05%	(14m)	13.57	16.09%	(2h)	3.23	90.28%	(3h)
	Compass	5.37	0.36%	(2m)	16.17	0.00%	(5m)	33.19	0.00%	(15m)
	Tsili (greedy)	4.08	24.25%	(4s)	12.46	22.94%	(4s)	25.69	22.59%	(5s)
	AM (greedy)	**5.19**	**3.64**%	(0s)	**15.64**	**3.23**%	(1s)	**31.62**	**4.75**%	(5s)
	GA (Python)	5.12	4.88%	(10m)	10.90	32.59%	(1h)	14.91	55.08%	(5h)
	OR-Tools (10s)	4.09	24.05%	(52m)		-			-	
	Tsili (sampling)	5.30	1.62%	(28s)	15.50	4.14%	(2m)	30.52	8.05%	(6m)
	AM (sampling)	**5.30**	**1.56**%	(4m)	**16.07**	**0.60**%	(16m)	**32.68**	**1.55**%	(53m)
PCTSP	Gurobi	3.13	0.00%	(2m)		-			-	
	Gurobi (1s)	3.14	0.07%	(1m)		-			-	
	Gurobi (10s)	3.13	0.00%	(2m)	4.54	1.36%	(32m)		-	
	Gurobi (30s)	3.13	0.00%	(2m)	4.48	0.03%	(54m)		-	
	AM (greedy)	**3.18**	**1.62**%	(0s)	**4.60**	**2.66**%	(2s)	**6.25**	**4.46**%	(5s)
	ILS (C++)	3.16	0.77%	(16m)	4.50	0.36%	(2h)	**5.98**	**0.00**%	(12h)
	OR-Tools (10s)	3.14	0.05%	(52m)	4.51	0.70%	(52m)	6.35	6.21%	(52m)
	OR-Tools (60s)	**3.13**	**0.01**%	(5h)	**4.48**	**0.00**%	(5h)	6.07	1.56%	(5h)
	ILS (Python 10x)	5.21	66.19%	(4m)	12.51	179.05%	(3m)	23.98	300.95%	(3m)
	AM (sampling)	3.15	0.45%	(5m)	4.52	0.74%	(19m)	6.08	1.67%	(1h)
SPCTSP	REOPT (all)	3.34	2.38%	(17m)	4.68	1.04%	(2h)	6.22	1.10%	(12h)
	REOPT (half)	3.31	1.38%	(25m)	**4.64**	**0.00**%	(3h)	**6.16**	**0.00**%	(16h)
	REOPT (first)	3.31	1.60%	(1h)	4.66	0.44%	(22h)		-	
	AM (greedy)	**3.26**	**0.00**%	(0s)	4.65	0.33%	(2s)	6.32	2.69%	(5s)

3.5.1 Problems

TRAVELLING SALESMAN PROBLEM (TSP) For the TSP, we report optimal results by Gurobi, as well as by Concorde (Applegate et al., 2006) (faster than Gurobi as it is specialized for TSP) and LKH3 (Helsgaun, 2017), an advanced heuristic solver that empirically also finds optimal solutions in time comparable to Gurobi. We compare against *nearest*, *random* and *farthest* insertion, as well as *nearest neighbor*, which is the only non-learned baseline algorithm that also constructs a tour directly in order (i.e. is structurally similar to our model). For details, see Appendix A.2.3. Additionally we compare against the learned heuristics in Section 3.2, most importantly Bello et al. (2016), as well as Google OR-Tools (Perron and Furnon, 2022), reported by Bello et al. (2016), and Christofides + 2OPT local search reported by Vinyals et al. (2015a). Results for Dai et al. (2017) are (optimistically) computed from the *optimality gaps* they report on 15-20, 40-50 and 50-100 node graphs, respectively. Using a *single* greedy construction we outperform traditional baselines and we are able to achieve significantly closer to optimal results than previous learned heuristics (from around 1.5% to 0.3% above optimal for $n = 20$). Naturally, the difference with Bello et al. (2016) gets diluted when sampling many solutions (as with many samples even a random policy performs well), but we still obtain significantly better results, *without* tuning the softmax temperature. For completeness, we also report results from running the *encode, attend & nagivate* (EAN) code[4] which is concurrent work by Deudon et al. (2018) (for details see Appendix A.2.4). Our model outperforms EAN, even if EAN is improved with 2OPT local search. Appendix A.2.5 presents the results visually, including generalization results for different n.

VEHICLE ROUTING PROBLEM (VRP) In the capacitated VRP (CVRP) (Toth and Vigo, 2014), each node has a demand and multiple routes should be constructed (starting and ending at the depot), such that the total demand of the nodes in each route does not exceed the vehicle capacity. We also consider the split delivery VRP (SDVRP), which allows to split customer demands over multiple routes. We implement the datasets described by Nazari et al. (2018) and compare against their reinforcement learning (RL) framework and the strongest baselines they report. Comparing greedy decoding, we obtain significantly better results. We cannot directly compare our sampling (1280 samples) to their beam search with size 10 (they do not report sampling or larger beam sizes), but note that our greedy method also outperforms their beam search in most (larger) cases, getting (in <1 second/instance) much closer to LKH3 (Helsgaun, 2017), a strong algorithm which has been used to find best known solutions to CVRP benchmarks. See Appendix A.3.4 for greedy example solution plots.

ORIENTEERING PROBLEM (OP) The OP (Golden et al., 1987) is an important problem used to model many real world problems. Each node has an associated *prize*, and the goal is to construct a single tour (starting and ending at the depot) that *maximizes* the sum of prizes of nodes visited while being shorter than a maximum

4 https://github.com/MichelDeudon/encode-attend-navigate

(given) length. We consider the prize distributions proposed in Fischetti et al. (1998): *constant*, *uniform* (in Appendix A.4.4), and increasing with the *distance* to the depot, which we report here as this is the hardest problem. As 'best possible solution' we report Gurobi (intractable for $n > 20$) and *Compass*, the recent and strong genetic algorithm (GA) by Kobeaga et al. (2018), which is only 2% better than sampling 1280 solutions with our method (objective is maximization). We outperform a Python GA[5] (which seems not to scale), as well as the construction phase of the heuristic by Tsiligirides (1984) (comparing greedy or 1280 samples) which is structurally similar to the one learned by our model. OR-Tools fails to find feasible solutions in a few percent of the cases for $n > 20$.

PRIZE COLLECTING TSP (PCTSP) In the PCTSP (Balas, 1989), each node has not only an associated prize, but also an associated penalty. The goal is to collect at least a *minimum* total prize, while minimizing the total tour length plus the sum of penalties of unvisited nodes. This problem is difficult as an algorithm has to trade off the penalty for not visiting a node with the marginal cost/tour length of visiting (which depends on the other nodes visited), while also satisfying the minimum total prize constraint. We compare against OR-Tools with 10 or 60 seconds of local search, as well as open source C++[6] and Python[7] implementations of iterated local search (ILS). Although the attention model does not find better solutions than OR-Tools with 60s of local search, it finds almost equally good results in significantly less time. The results are also within 2% of the C++ ILS algorithm (but obtained much faster), which was the best open-source algorithm for PCTSP we could find.

STOCHASTIC PCTSP (SPCTSP) The stochastic variant of the PCTSP (SPCTSP) we consider shows how our model can deal with uncertainty naturally. In the SPCTSP, the *expected* node prize is known upfront, but the real collected prize only becomes known upon visitation. With penalties, this problem is a generalization of the stochastic k-TSP (Ene et al., 2018). Since our model constructs a tour one node at the time, we only need to use the real prizes to compute the remaining prize constraint. By contrast, any algorithm that selects a fixed tour may fail to satisfy the prize constraint so an algorithm *must* be adaptive. As a baseline, we implement an algorithm that plans a tour, executes part of it and then re-optimizes using the C++ ILS algorithm. We either execute *all* node visits (so planning additional nodes if the result does not satisfy the prize constraint), *half* of the planned node visits (for $O(\log n)$ replanning iterations) or only the *first* node visit, for maximum adaptivity. We observe that our model outperforms all baselines for $n = 20$. We think that failure to account for uncertainty (by the baselines) in the prize might result in the need to visit one or two additional nodes, which is relatively costly for small instances but relatively cheap for larger n. Still, our method is beneficial as it provides competitive solutions at a fraction of the computational cost, which is important in online settings.

5 https://github.com/mc-ride/orienteering
6 https://github.com/jordanamecler/PCTSP
7 https://github.com/rafael2reis/salesman

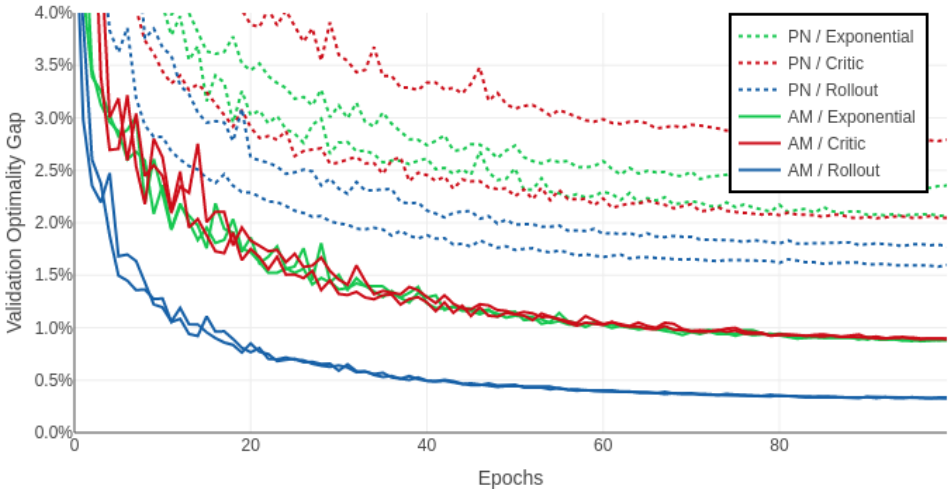


Figure 7: Held-out validation set optimality gap as a function of the number of epochs for the attention model (AM) and pointer network (PN) with different baselines (two different seeds).

3.5.2 Attention model vs. pointer network and different baselines

Figure 7 compares the performance of the TSP20 attention model (AM) and our implementation of the pointer network (PN) during training. We use a validation set of size 10000 with greedy decoding, and compare to using an exponential ($\beta = 0.8$) and a critic (see Appendix A.2.1) baseline. We used two random seeds and a decaying learning rate of $\eta = 10^{-3} \times 0.96^{\text{epoch}}$. This performs best for the PN, while for the AM results are similar to using $\eta = 10^{-4}$ (see Appendix A.2.5). This clearly illustrates how the improvement we obtain is the result of both the AM and the rollout baseline: the AM outperforms the PN using any baseline and the rollout baseline improves the quality and convergence speed for both AM and PN. For the PN with critic baseline, we are unable to reproduce the 1.5% reported by Bello et al. (2016) (also when using an LSTM based critic), but our reproduction is closer than others have reported (Dai et al., 2017; Nazari et al., 2018). In Table 1 we compare against the original results. Compared to the rollout baseline, the exponential baseline is around 20% faster per epoch, whereas the critic baseline is around 13% slower (see Appendix A.2.5), so the picture does not change significantly if time is used as x-axis.

3.6 DISCUSSION

In this chapter we have introduced a model and training method which both contribute to significantly improved results on learned heuristics for TSP and additionally learned strong (single construction) heuristics for multiple routing problems, which are traditionally solved by problem-specific approaches. We believe that our method is a powerful starting point for learning heuristics for other combinatorial

optimization problems defined on graphs, if their solutions can be described as sequential decisions. In practice, operational constraints often lead to many variants of problems for which no good (human-designed) heuristics are available such that the ability to learn heuristics could be of great practical value.

Compared to previous works, by using attention instead of recurrence (LSTMs) we introduce invariance to the input order of the nodes, increasing learning efficiency. Also this enables parallelization, for increased computational efficiency. The multi-head attention mechanism can be seen as a message passing algorithm that allows nodes to communicate relevant information over different channels, such that the node embeddings from the encoder can learn to include valuable information about the node *in the context of the graph*. This information is important in our setting where decisions relate directly to the nodes in a graph. Being a graph based method, our model has increased scaling potential (compared to LSTMs) as it can be applied on a sparse graph and operate locally.

Scaling to larger problem instances is an important direction for future research, where we think we have made an important first step by using a graph based method, which can be sparsified for improved computational efficiency. Another challenge is that many problems of practical importance have feasibility constraints that cannot be satisfied by a single forward construction with a simple masking procedure. Therefore, in the next chapter, we investigate how we can combine a learned policy with a form of search, to consider multiple solutions at the same time.

4 DEEP POLICY DYNAMIC PROGRAMMING

Routing problems are a class of combinatorial problems with many practical applications. Recently, end-to-end deep learning methods have been proposed to learn approximate solution heuristics for such problems. In contrast, classical dynamic programming (DP) algorithms guarantee optimal solutions, but scale badly with the problem size. We propose deep policy dynamic programming *(DPDP), which aims to combine the strengths of learned neural heuristics with those of DP algorithms. DPDP prioritizes and restricts the DP state space using a policy derived from a deep neural network, which is trained to predict edges from example solutions. We evaluate our framework on the travelling salesman problem (TSP), the vehicle routing problem (VRP) and TSP with time windows (TSPTW) and show that the neural policy improves the performance of (restricted) DP algorithms, making them competitive to strong alternatives such as LKH, while also outperforming most other 'neural approaches' for solving TSPs, VRPs and TSPTWs with 100 nodes.*

4.1 INTRODUCTION

Dynamic programming (DP) (Bertsekas, 2017) is a powerful framework for solving optimization problems by solving smaller subproblems through the principle of optimality (Bellman, 1952). Famous examples are Dijkstra's algorithm (Dijkstra, 1959) for the shortest route between two locations, and the classic Held-Karp algorithm for the travelling salesman problem (TSP) (Held and Karp, 1962; Bellman, 1962). Despite their long history, dynamic programming algorithms for vehicle routing problems (VRPs) have seen limited use in practice, primarily due to their bad scaling performance. More recently, a line of research has attempted the use of machine learning (especially deep learning) to automatically learn heuristics for solving routing problems (Vinyals et al., 2015a; Bello et al., 2016; Nazari et al., 2018; Kool et al., 2019a; Chen and Tian, 2019). While the results are promising (see also Chapter 3), most learned heuristics are not (yet) competitive to 'traditional' algorithms such as LKH (Helsgaun, 2017) and lack (asymptotic) guarantees on their performance.

In this chapter, we propose *deep policy dynamic programming* (DPDP) as a framework for solving vehicle routing problems. The key of DPDP is to combine the strengths of deep learning and DP, by restricting the DP state space (the search space) using a policy derived from a neural network. In Figure 8 it can be seen how the neural network indicates promising parts of the search space as a *heatmap* over the edges of the graph. This heatmap used by the DP algorithm to find a good solution. DPDP is more powerful than some related ideas (Yang et al., 2018; Heeswijk and La Poutré, 2019; Xu et al., 2020; Cappart et al., 2021; Li et al., 2018) as

(a) Travelling salesman problem

(b) Vehicle routing problem

(c) TSP with time windows

Figure 8: Heatmap predictions (red) and solutions (colored) by DPDP (VRP depot edges omitted for clarity). The heatmap indicates only a small fraction of all edges as promising, while including (almost) all edges from the solution.

it combines supervised training of a large neural network with just a *single* model evaluation at test time, to enable running a large scale guided search using DP. The DP framework is flexible as it can model a variety of realistic routing problems with difficult practical constraints (Gromicho et al., 2012a). We illustrate this by testing DPDP on the TSP, the capacitated VRP and the TSP with (hard) time window constraints (TSPTW).

In more detail, the starting point of our proposed approach is a *restricted dynamic programming* algorithm (Malandraki and Dial, 1996; Gromicho et al., 2012a), which heuristically reduces the search space by retaining at most B solutions per iteration. The selection process is important as it defines the part of the DP state space considered and, thus, the quality of the solution found (see Fig. 9). DPDP defines the selection using a (sparse) heatmap of promising route segments, obtained by preprocessing the problem instance using a (deep) graph neural network (GNN) (Joshi et al., 2019a). This brings the power of neural networks to DP, inspired by the success of neural networks that improved tree search (Silver et al., 2018) or branch-and-bound algorithms (Gasse et al., 2019; Nair et al., 2020).

In this work, we thus aim for a 'neural boost' of DP algorithms, by using a GNN for scoring partial solutions. Prior work on 'neural' vehicle routing has focused on auto-regressive models (Vinyals et al., 2015a; Bello et al., 2016; Deudon et al., 2018; Kool et al., 2019a), but they have high computational cost when combined with (any form of) search, as the model needs to be evaluated for each partial solution considered. Instead, we use a model to predict a heatmap indicating promising edges (Joshi et al., 2019a), and define the *score* of a partial solution as the 'heat' of the edges it contains (plus an estimate of the 'heat-to-go' or *potential* of the solution). As the neural network only needs to be evaluated *once* for each instance, this enables a *much larger search* (defined by B), making a good trade-off between quality and computational cost. Additionally, we can apply a threshold to the heatmap to define a sparse graph on which to run the DP algorithm, reducing the runtime by eliminating many solutions.

Figure 9 illustrates DPDP. In Section 4.4, we show that DPDP significantly improves over 'classic' restricted DP algorithms. Additionally, we show that DPDP

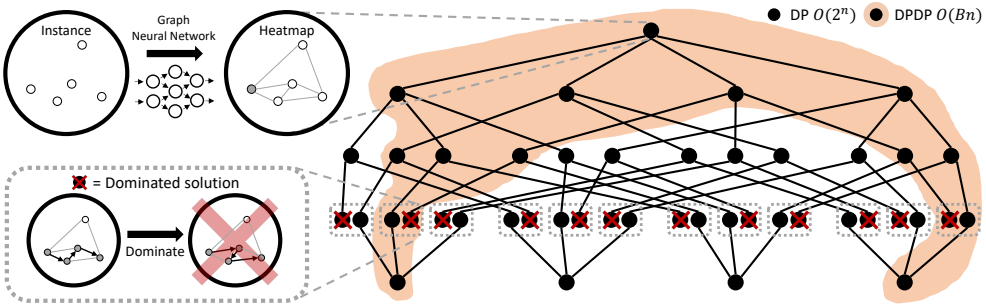


Figure 9: DPDP for the TSP. A GNN creates a (sparse) heatmap indicating promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most B solutions are expanded according to the heatmap policy, restricting the size of the search space. Partial solutions are dominated by shorter (lower cost) solutions with the same DP state: the same nodes visited (marked grey) and current node (indicated by dashed rectangles).

outperformes most other 'neural' approaches for TSP, VRP and TSPTW and is competitive with the highly-optimized LKH solver (Helsgaun, 2017) for VRP, while achieving similar results much faster for TSP and TSPTW. For TSPTW, DPDP also outperforms the best open-source solver we could find (Da Silva and Urrutia, 2010), illustrating the power of DPDP to handle difficult hard constraints (time windows).

4.2 RELATED WORK

DP (Bertsekas, 2017) has a long history as an exact solution method for routing problems (Laporte, 1992; Toth and Vigo, 2014), e.g. the TSP with time windows (Dumas et al., 1995) and precedence constraints (Mingozzi et al., 1997), but is limited to small problems due to the curse of dimensionality. Restricted DP has been used to address, e.g., the time dependent TSP (Malandraki and Dial, 1996), and has been generalized into a flexible framework for VRPs with different types of practical constraints (Gromicho et al., 2012a). DP approaches have also been shown to be useful in settings with difficult practical issues such as time-dependent travel times and driving regulations (Kok et al., 2010) or stochastic demands (Novoa and Storer, 2009). For more examples of DP for routing (and scheduling), see Hoorn (2016). For sparse graphs, alternative formulations can be used (Cook and Seymour, 2003).

Despite the flexibility, DP methods have not gained much popularity compared to heuristic approaches such as R&R (Schrimpf et al., 2000), ALNS (Ropke and Pisinger, 2006), LKH (Helsgaun, 2017), HGS (Vidal et al., 2012; Vidal, 2022) or FILO (Accorsi and Vigo, 2021), which, while effective, have limited flexibility as special operators are needed for different types of problems. While restricted DP was shown to have superior performance on *realistic* VRPs with many constraints (Gromicho et al., 2012a), the performance gap of around 10% for standard (benchmark) VRPs (with time windows) is too large to popularize this approach. We argue that the missing ingredient is a strong but computationally cheap policy for selecting which solutions to consider, which is the motivation behind DPDP.

In the machine learning community, deep neural networks (DNNs) have recently boosted performance on various tasks (LeCun et al., 2015). After the first DNN model was trained (using example solutions) to construct TSP tours (Vinyals et al., 2015a), many improvements have been proposed, e.g. different training strategies such as reinforcement learning (RL) (Bello et al., 2016; Joshi et al., 2019b; Delarue et al., 2020; Kwon et al., 2020) and model architectures, which enabled the same idea to be used for other routing problems (Nazari et al., 2018; Kool et al., 2019a; Deudon et al., 2018; Peng et al., 2019; Falkner and Schmidt-Thieme, 2020; Xin et al., 2020; Ma et al., 2021) (see also Chapter 3). Most constructive neural methods are *auto-regressive*, evaluating the model many times to predict one node at the time, but other works have considered predicting a heatmap of promising edges *at once* (Nowak et al., 2017; Joshi et al., 2019a; Fu et al., 2021), which allows a tour to be constructed (using sampling or beam search) without further evaluating the model. An alternative to constructive methods is 'learning to search', where a neural network is used to guide a search procedure such as local search (Chen and Tian, 2019; Lu et al., 2020; Gao et al., 2020; Wu et al., 2021; Hottung and Tierney, 2020; Kim et al., 2021; Li et al., 2021; Xin et al., 2021; Hottung et al., 2021). Scaling to instances beyond 100 nodes remains challenging (Ma et al., 2020; Fu et al., 2021).

The combination of machine learning with DP has been proposed in limited settings (Yang et al., 2018; Heeswijk and La Poutré, 2019; Xu et al., 2020). Most related to our approach, a DP algorithm for TSPTW, guided by an RL agent, was implemented using an existing solver (Cappart et al., 2021), which is less efficient than DPDP (see Section 4.4.3). Also similar to our approach, a neural network predicting edges has been combined with tree search and local search for maximum independent set (MIS) (Li et al., 2018). Whereas DPDP directly builds on the idea of predicting promising edges (Li et al., 2018; Joshi et al., 2019a), it uses these more efficiently through a policy with *potential function* (see Section 4.3.2), and by using DP rather than tree search or beam search, we exploit known problem structure in a principled and general manner. As such, DPDP obtains strong performance without using extra heuristics such as local search. For a wider view on machine learning for routing problems and combinatorial optimization, see Mazyavkina et al. (2020), Vesselinova et al. (2020), and Bai et al. (2021).

4.3 DEEP POLICY DYNAMIC PROGRAMMING

DPDP uses an existing graph neural network (Joshi et al., 2019a), suitably adapted for VRP and TSPTW, to predict a heatmap of promising edges. This heatmap is used in the DP algorithm in two ways: 1) to exclude edges with a value below the *heatmap threshold* of 10^{-5} from the graph and 2) to define a *scoring policy* to select candidate solutions in each iteration. In more detail, as illustrated in Fig. 9, the DP algorithm starts with a *beam* of a single initial (empty) solution, and proceeds by iterating the following steps: (1) all solutions on the beam are expanded, (2) dominated solutions are removed for each *DP state*, (3) the B best solutions according to the scoring policy define the beam for the next iteration. The objective function is used to select the

best solution from the final beam. The resulting algorithm is a *beam search* over the *DP state space*, with *beam size* B. This is different from a 'standard' beam search, which considers the *solution space* by not removing dominated solutions. DPDP is asymptotically optimal as using $B = n \cdot 2^n$ for a TSP with n nodes guarantees optimal results, but by choosing a smaller B, DPDP can trade off performance for computational cost.

DPDP is a generic framework that can be applied to different problems, by defining the following ingredients: (1) the **variables** to track while constructing solutions, (2) the **initial solution**, (3) **feasible actions** to expand solutions, (4) rules to define **dominated solutions** and (5) the **scoring policy**, based on the neural network, for selecting the B solutions to keep. A solution is always defined by a sequence of actions, which allows the DP algorithm to construct the final solution by backtracking. In the next sections, we describe the neural network and define the DPDP ingredients for the TSP, VRP and TSPTW.

4.3.1 The graph neural network

We use the original (pre-trained) model from Joshi et al. (2019a) (which we describe in detail in Appendix B.1 for self-containment) for the TSP, but we modify the neural network architecture and train new models to support the VRP and TSPTW, as we describe in Sections 4.3.3 and 4.3.4. In general, the resulting model uses problem-specific node input features and edge input features, which get transformed into initial representations of the nodes and edges. These representations then get updated sequentially using a number of *graph convolutional layers*, which exchange information between the nodes and edges. The final edge representation is used to make the prediction whether the edge is promising, i.e. whether it has a high probability of being part of the optimal solution.

The model is trained using a large training dataset of problem instances with optimal (or high-quality) solutions, obtained using an existing solver. While it takes a significant amount of resources to create this dataset and train the model (each of which can take up to a number of days on a single machine), training of the model is, in principle, only required once given a specific distribution of problem instances. We consider only instances with $n = 100$ nodes, but the model can handle instances of different graph sizes, although good generalization may be limited to graphs with sizes close to the size trained for (Kool et al., 2019a; Joshi et al., 2019b).

4.3.2 Travelling salesman problem

We implement DPDP for Euclidean TSPs with n nodes on a (sparse) graph, where the cost for edge (i, j) is given by c_{ij}, the Euclidean distance between the nodes i and j. The objective is to construct a tour that visits all nodes (and returns to the start node) and minimizes the total cost of its edges.

For each partial solution, defined by a sequence of actions $\boldsymbol{a}$, the **variables** we track are $\text{cost}(\boldsymbol{a})$, the total *cost* (distance), $\text{current}(\boldsymbol{a})$, the current node, and $\text{visited}(\boldsymbol{a})$, the set of visited nodes (including the start node). Without loss of generality, we let 0 be the start node, so we initialize the beam at step $t = 0$ with the empty **initial solution** with $\text{cost}(\boldsymbol{a}) = 0$, $\text{current}(\boldsymbol{a}) = 0$ and $\text{visited}(\boldsymbol{a}) = \{0\}$. At step t, the action $a_t \in \{0, ..., n-1\}$ indicates the next node to visit, and is a **feasible action** for a partial solution $\boldsymbol{a} = (a_0, ..., a_{t-1})$ if (a_{t-1}, a_t) is an edge in the graph and $a_t \notin \text{visited}(\boldsymbol{a})$, or, when all nodes are visited, if $a_t = 0$ to return to the start node. When expanding the solution to $\boldsymbol{a}' = (a_0, ..., a_t)$, we can compute the tracked variables incrementally as:

$$\text{cost}(\boldsymbol{a}') = \text{cost}(\boldsymbol{a}) + c_{\text{current}(a), a_t}, \tag{20}$$

$$\text{current}(\boldsymbol{a}') = a_t, \tag{21}$$

$$\text{visited}(\boldsymbol{a}') = \text{visited}(\boldsymbol{a}) \cup \{a_t\}. \tag{22}$$

A (partial) solution $\boldsymbol{a}$ is a **dominated solution** if there exists a (dominating) solution $\boldsymbol{a}^*$ such that $\text{visited}(\boldsymbol{a}^*) = \text{visited}(\boldsymbol{a})$, $\text{current}(\boldsymbol{a}^*) = \text{current}(\boldsymbol{a})$ and $\text{cost}(\boldsymbol{a}^*) < \text{cost}(\boldsymbol{a})$. We refer to the tuple $(\text{visited}(\boldsymbol{a}), \text{current}(\boldsymbol{a}))$ as the *DP state*, so removing all dominated partial solutions, we keep exactly one minimum-cost solution for each unique DP state[1]. A solution can only dominate other solutions with the same set of visited nodes, so we only need to remove dominated solutions from sets of solutions with the same number of actions. This is why the DP algorithm can be executed in iterations (as explained): at step t all solutions in the beam have t actions and $t+1$ visited nodes (including the start node). The resulting memory need is thus limited to $O(B)$ states, with B the beam size.

We define the **scoring policy** using the pretrained model from Joshi et al. (2019a), which takes as input node coordinates and edge distances to predict a raw heatmap value $\hat{h}_{ij} \in (0, 1)$ for each edge (i, j). The model was trained to predict optimal solutions, so $\hat{h}_{ij}$ can be seen as the probability that edge (i, j) is in the optimal tour. We force the heatmap to be symmetric thus we define $h_{ij} = \max\{\hat{h}_{ij}, \hat{h}_{ji}\}$. The policy is defined using the heatmap values, in such a way to select the (partial) solutions with the largest total *heat*, while also taking into account the (heat) *potential* for the unvisited nodes. The policy thus selects the B solutions which have the highest *score*, defined as $\text{score}(\boldsymbol{a}) = \text{heat}(\boldsymbol{a}) + \text{potential}(\boldsymbol{a})$, with $\text{heat}(\boldsymbol{a}) = \sum_{i=1}^{t-1} h_{a_{i-1}, a_i}$, i.e. the sum of the heat of the edges, which can be computed incrementally when expanding a solution. The potential is added as an estimate of the 'heat-to-go' (similar to the heuristic in A^* search) for the remaining nodes, and avoids the 'greedy pitfall' of selecting the best edges while skipping over nearby nodes, which would prevent good edges from being used later. It is defined as $\text{potential}(\boldsymbol{a}) = \text{potential}_0(\boldsymbol{a}) + \sum_{i \notin \text{visited}(\boldsymbol{a})} \text{potential}_i(\boldsymbol{a})$ with $\text{potential}_i(\boldsymbol{a}) = w_i \sum_{j \notin \text{visited}(\boldsymbol{a})} \frac{h_{ji}}{\sum_{k=0}^{n-1} h_{ki}}$, where w_i is the node *potential weight* given by $w_i = (\max_j h_{ji}) \cdot (1 - 0.1(\frac{c_{i0}}{\max_j c_{j0}} - 0.5))$. By normalizing the heatmap values for incoming edges, the (remaining) potential for node i is initially equal to w_i but decreases as good edges become infeasible due to neighbors being visited. The node

1 If we have multiple partial solutions with the same state and cost, we can arbitrarily choose one to dominate the other(s), for example the one with the lowest index of the current node.

potential weight w_i is equal to the maximum incoming edge heatmap value (an upper bound to the heat contributed by node i), which gets multiplied by a factor 0.95 to 1.05 to give a higher weight to nodes closer to the start node, which we found helps to encourage the algorithm to keep edges that enable to return to the start node. The overall heat + potential function identifies promising partial solutions and is computationally cheap. It is a heuristic estimate of the total heat of the complete solution, but it is not an estimate of the cost objective (which has a different unit), neither it is a *bound* on the total heat or cost objective.

4.3.3 Vehicle routing problem

For the VRP, we add a special depot node DEP to the graph. Node i has a demand d_i, and the goal is to minimize the cost for a set of routes that visit all nodes. Each route must start and end at the depot, and the total demand of its nodes cannot exceed the vehicle capacity denoted by CAPACITY.

Additionally to the TSP **variables** cost($\boldsymbol{a}$), current($\boldsymbol{a}$) and visited($\boldsymbol{a}$), we keep track of capacity($\boldsymbol{a}$), which is the *remaining* capacity in the current route/vehicle. A solution starts at the depot, so we initialize the beam at step $t = 0$ with the empty **initial solution** with cost($\boldsymbol{a}$) $= 0$, current($\boldsymbol{a}$) $=$ DEP, visited($\boldsymbol{a}$) $= \varnothing$ and capacity($\boldsymbol{a}$) $=$ CAPACITY. For the VRP, we do not consider visiting the depot as a separate action. Instead, we define $2n$ actions, where $a_t \in \{0, ..., 2n-1\}$. The actions $0, ..., n-1$ indicate a *direct* move from the current node to node a_t, whereas the actions $n, ..., 2n-1$ indicate a move to node $a_t - n$ *via the depot*. **Feasible actions** are those that move to unvisited nodes via edges in the graph and obey the following constraints. For the first action a_0 there is no choice and we constrain (for convenience of implementation) $a_0 \in \{n, ..., 2n-1\}$. A direct move ($a_t < n$) is only feasible if $d_{a_t} \leq$ capacity($\boldsymbol{a}$) and updates the state similar to TSP but reduces remaining capacity by d_{a_t}. A move via the depot is always feasible (respecting the graph edges and assuming $d_i \leq$ CAPACITY $\forall i$) as it resets the vehicle CAPACITY before subtracting demand, but incurs the 'via-depot cost' $c_{ij}^{\text{DEP}} = c_{i,\text{DEP}} + c_{\text{DEP},j}$. When all nodes are visited, we allow a special action to return to the depot. This somewhat unusual way of representing a VRP solution has desirable properties similar to the TSP formulation: at step t we have exactly t nodes visited, and we can run the DP in iterations, removing dominated solutions at each step t.

For VRP, a partial solution $\boldsymbol{a}$ is a **dominated solution** dominated by $\boldsymbol{a}^*$ if visited($\boldsymbol{a}^*$) $=$ visited($\boldsymbol{a}$) and current($\boldsymbol{a}^*$) $=$ current($\boldsymbol{a}$) (i.e. $\boldsymbol{a}^*$ corresponds to the same DP state) and cost($\boldsymbol{a}^*$) $\leq$ cost($\boldsymbol{a}$) and capacity($\boldsymbol{a}^*$) $\geq$ capacity($\boldsymbol{a}$), with *at least one of the two inequalities being strict*. This means that for each DP state, given by the set of visited nodes and the current node, we do not only keep the (single) solution with lowest cost (as in the TSP algorithm), but keep the complete set of pareto-efficient solutions in terms of cost and remaining vehicle capacity. This is because a higher cost partial solution may still be preferred if it has more remaining vehicle capacity, and vice versa.

For the VRP **scoring policy**, we modify the model (Joshi et al., 2019a) (described in Appendix B.1) to include the depot node and demands. We mark the depot as a special node type, which affects the initial node representation similarly to edge types, and we add additional edge types for connections to the depot. Additionally, each node gets an extra input (next to its coordinates) corresponding to d_i/CAPACITY (where we set $d_{\text{DEP}} = 0$). The model is trained on example solutions from LKH (Helsgaun, 2017) (see Section 4.4.2), which are not optimal, but still provide a useful training signal. Compared to TSP, the definition of the heat is slightly changed to accommodate for the 'via-depot actions' and is best defined incrementally using the 'via-depot heat' $h_{ij}^{\text{DEP}} = h_{i,\text{DEP}} \cdot h_{\text{DEP},j} \cdot 0.1$, where multiplication is used to keep heat values interpretable as probabilities and in the range $(0, 1)$. The additional penalty factor of 0.1 for visiting the depot encourages the algorithm to minimize the number of vehicles/routes. The heat of the initial state is 0 and when expanding a solution $\boldsymbol{a}$ to $\boldsymbol{a}'$ using action a_t, the heat is incremented with either $h_{\text{current}(\boldsymbol{a}),a_t}$ (if $a_t < n$) or $h^{\text{DEP}}_{\text{current}(\boldsymbol{a}),a_t-n}$ (if $a_t \geq n$). The potential is defined similarly to TSP, replacing the start node 0 by DEP.

4.3.4 Travelling salesman problem with time windows

For the TSPTW, we also have a special depot/start node 0. The goal is to create a single tour that visits each node i in a time window defined by (l_i, u_i), where the travel time from i to j is equal to the cost/distance c_{ij}, i.e. we assume a speed of 1 (w.l.o.g. as we can rescale time). It is allowed to wait if arrival at node i is before l_i, but arrival cannot be after u_i. We minimize the total *cost* (*excluding* waiting time), but to minimize *makespan* (including waiting time), we only need to train on different example solutions. Due to the hard constraints, TSPTW is typically considered more challenging than plain TSP, for which every solution is feasible.

The **variables** we track and **initial solution** are equal to TSP except that we add $\text{time}(\boldsymbol{a})$ which is initially 0 $(= l_0)$. **Feasible actions** $a_t \in \{0, ..., n-1\}$ are those that move to unvisited nodes via edges in the graph such that the arrival time is no later than u_{a_t} and do not directly eliminate the possibility to visit other nodes in time[2]. Expanding a solution $\boldsymbol{a}$ to $\boldsymbol{a'}$ using action a_t updates the time as $\text{time}(\boldsymbol{a'}) = \max\{\text{time}(\boldsymbol{a}) + c_{\text{current}(a),a_t}, l_{a_t}\}$.

For each DP state, we keep all efficient solutions in terms of cost and time, so a partial solution $\boldsymbol{a}$ is a **dominated solution** dominated by $\boldsymbol{a}^*$ if $\boldsymbol{a}^*$ has the same DP state ($\text{visited}(\boldsymbol{a}^*) = \text{visited}(\boldsymbol{a})$ and $\text{current}(\boldsymbol{a}^*) = \text{current}(\boldsymbol{a})$) and is strictly better in terms of cost and time, i.e. $\text{cost}(\boldsymbol{a}^*) \leq \text{cost}(\boldsymbol{a})$ and $\text{time}(\boldsymbol{a}^*) \leq \text{time}(\boldsymbol{a})$, with *at least one of the two inequalities being strict*.

The model (Joshi et al., 2019a) for the **scoring policy** is adapted to include the time windows (l_i, u_i) as node features (scaled to correspond to a speed of 1 for the input distances and coordinates, which are scaled to the range $[0, 1]$), and we use a special embedding for the depot similar to VRP. Due to the time dimension,

2 E.g., arriving at node i at $t = 10$ is not feasible if node j has $u_j = 12$ and $c_{ij} = 3$.

a TSPTW solution is *directed*, and edge (i, j) may be good whereas (j, i) may be not, so we adapt the model to enable predictions $h_{ij} \neq h_{ji}$ (see Appendix B.1). We generated example training solutions using (heuristic) DP with a large beam size, which was faster than LKH. Given the heat predictions, the score (heat + potential) is exactly as for TSP.

4.4 EXPERIMENTS

We implement DPDP using PyTorch (Paszke et al., 2019) to leverage GPU computation. For details, see Appendix B.2. Our code is publicly available.[3] DPDP has very few hyperparameters, but the heatmap threshold of 10^{-5} and details like the functional form of e.g. the scoring policy are 'educated guesses' or manually tuned on a few validation instances and can likely be improved. The runtime is influenced by implementation choices which were tuned on a few validation instances.

4.4.1 Travelling salesman problem

In Table 2 we report our main results for DPDP with beam sizes of 10K (10 thousand) and 100K, for the TSP with 100 nodes on a commonly used test set of 10000 instances (Kool et al., 2019a). We report cost and *gap* to the optimal solution found using Concorde (Applegate et al., 2006) (following (Kool et al., 2019a)) and compare against LKH (Helsgaun, 2017) and Gurobi (Gurobi Optimization, LLC, 2022), as well as recent results of the strongest methods using neural networks ('neural approaches') from literature. Running times for solving 10000 instances *after training* should be taken as rough indications as some are on different machines, typically with 1 GPU or a many-core CPU (8 - 32). The costs indicated with * are not directly comparable due to slight dataset differences (Fu et al., 2021). Times for generating heatmaps (if applicable) is reported separately (as the first term) from the running time for MCTS (Fu et al., 2021) or DP. DPDP achieves close to optimal results, strictly outperforming the neural baselines achieving better results in less time (except the attention model trained with POMO (Kwon et al., 2020), see Section 4.4.2).

4.4.2 Vehicle routing problem

For the VRP, we train the model using 1 million instances of 100 nodes, generated according to the distribution described by Nazari et al. (2018) and solved using one run of LKH (Helsgaun, 2017). We train using a batch size of 48 and a learning rate of 10^{-3} (selected as the result of manual trials to best use our GPUs), for (at most) 1500 epochs of 500 training steps (following Joshi et al. (2019a)) from which we

3 https://github.com/wouterkool/dpdp

Table 2: Mean cost, gap and *total time* to solve 10000 TSP/VRP test instances.

Problem	TSP100			VRP100		
Method	Cost	Gap	Time	Cost	Gap	Time
Concorde (Applegate et al., 2006)	7.765	0.000 %	6m		-	
HGS (Vidal et al., 2012; Vidal, 2022)		-		15.563	0.000 %	6h11m
Gurobi (Gurobi Optimization, LLC, 2022)	7.776	0.151 %	31m		-	
LKH (Helsgaun, 2017)	7.765	0.000 %	42m	15.647	0.536 %	12h57m
GNN heatmap + beam search (Joshi et al., 2019a)	7.87	1.39 %	40m		-	
Learning 2-opt heuristics (Costa et al., 2020)	7.83	0.87 %	41m		-	
Merged GNN heatmap + MCTS (Fu et al., 2021)	7.764*	0.04 %	4m + 11m		-	
Attention model + sampling (Kool et al., 2019a)	7.94	2.26 %	1h	16.23	4.28 %	2h
Step-wise attention model (Xin et al., 2020)	8.01	3.20 %	29s	16.49	5.96 %	39s
Attn. model + coll. policies (Kim et al., 2021)	7.81	0.54 %	12h	15.98	2.68 %	5h
Learning improv. heuristics (Wu et al., 2021)	7.87	1.42 %	2h	16.03	3.00 %	5h
Dual-aspect coll. Transformer (Ma et al., 2021)	7.77	0.09 %	5h	15.71	0.94 %	9h
Attention model + POMO (Kwon et al., 2020)	7.77	0.14 %	1m	15.76	1.26 %	2m
NeuRewriter (Chen and Tian, 2019)		-		16.10	3.45 %	1h
Dynamic attn. model + 2-opt (Peng et al., 2019)		-		16.27	4.54 %	6h
Neural LNS (Hottung and Tierney, 2020)		-		15.99	2.74 %	1h
Learn to improve (Lu et al., 2020)		-		15.57*	-	4000h
DPDP 10K	7.765	0.009 %	10m + 16m	15.830	1.713 %	10m + 50m
DPDP 100K	7.765	0.004 %	10m + 2h35m	15.694	0.843 %	10m + 5h48m
DPDP 1M				15.627	0.409 %	10m + 48h27m

select the saved checkpoint with the lowest validation loss. We use the validation and test sets by Kool et al. (2019a).

Table 2 shows the results, where the gap is relative to Hybrid Genetic Search (HGS)[4], a SOTA heuristic VRP solver (Vidal et al., 2012; Vidal, 2022). HGS is faster and improves around 0.5% over LKH (Helsgaun, 2017), which is typically considered the baseline in related work. We present the results for LKH, as well as the strongest neural approaches and DPDP with beam sizes up to 1 million. Some results used 2000 (different) instances (Lu et al., 2020) and cannot be directly compared[5]. DPDP outperforms all other neural baselines, except the attention model trained with POMO (Kwon et al., 2020), which delivers good results very quickly by exploiting symmetries in the problem. However, as it cannot (easily) improve further with additional runtime, we consider this contribution orthogonal to DPDP. DPDP is competitive to LKH (see also Section 4.4.4).

MORE REALISTIC INSTANCES We also train the model and run experiments with instances with 100 nodes from a more realistic and challenging data distribution (Uchoa et al., 2017). This distribution, commonly used in the routing community, has greater variability, in terms of node clustering and demand distributions. LKH failed to solve two of the test instances, which is because LKH by default uses a fixed number of routes equal to a lower bound, given by $\left\lceil \frac{\sum_{i=0}^{n-1} d_i}{\text{CAPACITY}} \right\rceil$, which may be infeasible[6]. Therefore we solve these instances by rerunning LKH with an unlimited number of allowed routes (which gives worse results, see Section 4.4.4).

DPDP was run on a machine with 4 GPUs, but we also report (estimated) runtimes for 1 GPU (1080Ti), and we compare against 16 or 32 CPUs for HGS and LKH. In Table 3 it can be seen that the difference with LKH is, as expected, slightly larger

4 https://github.com/vidalt/HGS-CVRP
5 The running time of 4000 hours (167 days) is estimated from 24min/instance (Lu et al., 2020).
6 For example, three nodes with a demand of two cannot be assigned to two routes with a capacity of three.

Table 3: Mean cost, gap and *total time* to solve 10000 realistic VRP100 instances.

METHOD	COST	GAP	TIME (1 GPU OR 16 CPUS)	TIME (4 GPUS OR 32 CPUS)
HGS (VIDAL ET AL., 2012; VIDAL, 2022)	18050	0.000 %	7H53M	3H56M
LKH (HELSGAUN, 2017)	18133	0.507 %	25H32M	12H46M
DPDP 10K	18414	2.018 %	10M + 50M	2M + 13M
DPDP 100K	18253	1.127 %	10M + 5H48M	2M + 1H27M
DPDP 1M	18168	0.659 %	10M + 48H27M	2M + 12H7M

than for the simpler dataset, but still below 1% for beam sizes of 100K-1M. We also observed a higher validation loss, so it may be possible to improve results using more training data. Nevertheless, finding solutions within 1% of the specialized SOTA HGS algorithm, and even closer to LKH, is impressive for these challenging instances, and we consider the runtime (for solving 10K instances) acceptable, especially when using multiple GPUs.

4.4.3 TSP with time windows

For the TSP with hard time window constraints, we use the data distribution by Cappart et al. (2021) and use their set of 100 test instances with 100 nodes. These were generated with small time windows, resulting in a small feasible search space, such that even with very small beam sizes, our DP implementation solves these instances optimally, eliminating the need for a policy. Therefore, we also consider a more difficult distribution similar to Da Silva and Urrutia (2010), which has larger time windows which are more difficult as the feasible search space is larger[7] (Dumas et al., 1995). For details, see Appendix B.1. For both distributions, we generate training data and train the model exactly as we did for the VRP.

Table 4 shows the results for both data distributions, which are reported in terms of the difference to General Variable Neighborhood Search (GVNS) (Da Silva and Urrutia, 2010), the best open-source solver for TSPTW we could find[8], using 30 runs. For the small time window setting, both GVNS and DPDP find optimal solutions for all 100 instances in just 7 seconds (in total, either on 16 CPUs or a single GPU). LKH fails to solve one instance, but finds close to optimal solutions, but around 50 times slower. BaB-DQN* and ILDS-DQN* (Cappart et al., 2021), methods combining an existing solver with an RL trained neural policy, take around 15 minutes *per instance* (orders of magnitudes slower) to solve most instances to optimality. Due to complex set-up, we were unable to run BaB-DQN* and ILDS-DQN* ourselves for the setting with larger time windows. In this setting, we find DPDP outperforms both LKH (where DPDP is orders of magnitude faster) and GVNS, in both speed and solution quality. This illustrates that DPDP, due to its nature, is especially well suited to handle constrained problems.

7 Up to a limit, as making the time windows infinite size reduces the problem to plain TSP.
8 https://github.com/sashakh/TSPTW

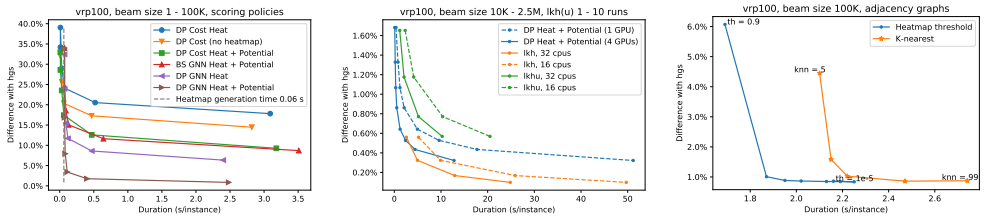
Table 4: Mean cost, gap and *total time* to solve TSPTW100 instances.

Problem / Method	Small time windows (100 inst.) (Cappart et al., 2021) Cost	Gap	Fail	Time	Large time windows (10K inst.) (Da Silva and Urrutia, 2010) Cost	Gap	Time
GVNS 30x (Da Silva and Urrutia, 2010)	5129.58	0.000 %		7s	2432.112	0.000 %	37m15s
GVNS 1x (Da Silva and Urrutia, 2010)	5129.58	0.000 %		<1s	2457.974	1.063 %	1m4s
LKH 1x (Helsgaun, 2017)	5130.32	0.014 %	1.00 %	5m48s	2431.404	-0.029 %	34h58m
BaB-DQN* (Cappart et al., 2021)	5130.51	0.018 %		25h		-	
ILDS-DQN* (Cappart et al., 2021)	5130.45	0.017 %		25h		-	
DPDP 10K	5129.58	0.000 %		6s + 1s	2431.143	-0.040 %	10m + 8m7s
DPDP 100K	5129.58	0.000 %		6s + 1s	2430.880	- 0.051 %	10m + 1h16m

4.4.4 Ablations

SCORING POLICY To evaluate the value of different components of DPDP's **GNN Heat + Potential** scoring policy, we compare against other variants. **GNN Heat** is the version without the potential, whereas **Cost Heat + Potential** and **Cost Heat** are variants that use a 'heuristic' $\hat{h}_{ij} = \frac{c_{ij}}{\max_k c_{ik}}$ instead of the GNN. **Cost** directly uses the current cost of the solution, and can be seen as 'classic' restricted DP. Finally, **BS GNN Heat + Potential** uses beam search without dynamic programming, i.e. without removing dominated solutions. To evaluate only the scoring policy, each variant uses the fully connected graph (no heatmap threshold). Figure 10a shows the value of DPDP's potential function, although even without it results are still significantly better than 'classic' heuristic DP variants using cost-based scoring policies. Also, it is clear that using DP significantly improves over a standard beam search (by removing dominated solutions). Lastly, the figure illustrates how the time for generating the heatmap using the neural network, despite its significant value, only makes up a small portion of the total runtime.

BEAM SIZE With DPDP, we can trade off the performance vs. the runtime using the beam size B (and the graph sparsity, see below). Figure 10b illustrates this trade-off, where we evaluate DPDP on 100 validation instances for VRP, with different beam sizes from 10K to 2.5M. We also report the trade-off curve for LKH(U), which is the strongest baseline that can also solve different problems. We vary the runtime using 1, 2, 5 and 10 runs (returning the best solution). LKHU(nlimited) is the version which allows an unlimited number of routes (see Section 4.4.2). It is hard to compare GPU vs CPU, so we report (estimated) runtimes for different hardware, i.e. 1 or 4 GPUs (with 3 CPUs per GPU) and 16 or 32 CPUs. We report the difference (i.e. the gap) with HGS, analogous to how results are reported in Table 2. We emphasize that in most related work (e.g. Kool et al. (2019a)), the strongest baseline considered is one run of LKH, so we compare against a much stronger baseline. Also, our goal is not to outperform HGS (which is SOTA and specific to VRP) or LKH, but to show DPDP has reasonable performance, while being a flexible framework for other (routing) problems.



(a) Different scoring policies, as well as 'pure' beam search, for beam sizes 1, 10, 100, 1000, 10K, 100K.

(b) Beam sizes 10K, 25K, 50K, 100K, 250K, 500K, 1M, 2.5M compared against LKH(U) with 1, 2, 5 and 10 runs.

(c) Sparsities with heatmap thresholds 0.9, 0.5, 0.2, 0.1, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5} and knn $= 5, 10, 20, 50, 99$. Beam size 100K.

Figure 10: DPDP ablations on 100 validation instances of VRP with 100 nodes.

GRAPH SPARSITY Using the heatmap threshold, the DP algorithm uses a sparse graph to define feasible expansions, which reduces the runtime but may also sacrifice solution quality. For most edges, the model confidently predicts close to 0, such that they are ruled out, even using the default (low) heatmap threshold of 10^{-5}. We may rule out even more edges by increasing the threshold, which can be seen as a secondary way (besides varying the beam size) to trade off the performance and computational cost of DPDP. While this can be seen as a form of learned *problem reduction* (Sun et al., 2020), we also consider a heuristic alternative of using the K-nearest neighbor (KNN) graph.[9] In Figure 10c, we experiment with different heatmap thresholds from 10^{-5} to 0.9 and different values for KNN from 5 to 99 (fully connected). The heatmap threshold strategy clearly outperforms the KNN strategy as it yields the same results using sparser graphs (and lower runtimes). This illustrates that the heatmap threshold strategy is more informed than the KNN strategy, confirming the value of the neural network predictions.

4.5 DISCUSSION

In this chapter we introduced *deep policy dynamic programming*, which combines machine learning and dynamic programming for solving vehicle routing problems. The method yields close to optimal results for TSPs with 100 nodes and is competitive to the highly optimized LKH (Helsgaun, 2017) solver for VRPs with 100 nodes. On the TSPTW, DPDP also outperforms LKH, being significantly faster, as well as GVNS (Da Silva and Urrutia, 2010), the best open source solver we could find. Given that DPDP was not specifically designed for TSPTW, and thus can likely be improved, we consider this an impressive and promising achievement.

The constructive nature of DPDP (combined with search) naturally supports hard constraints such as time windows, which are typically considered challenging in neural combinatorial optimization (Bello et al., 2016; Kool et al., 2019a) and are also difficult for local search heuristics (as they need to maintain feasibility while adapting a solution). Given our results on TSP, VRP and TSPTW, and the flexibility of

9 For the symmetric TSP and VRP, we add KNN edges in both directions. For the VRP, we also connect each node to the depot (and vice versa) to ensure feasibility.

DP as a framework, we think DPDP has great potential for solving many more variants of routing problems, and possibly even other problems that can be formulated using DP (e.g. job shop scheduling (Gromicho et al., 2012b)). We hope that our work brings machine learning research for combinatorial optimization closer to the operations research (especially vehicle routing) community, by combining machine learning with DP and evaluating the resulting new framework on different data distributions used by different communities (Nazari et al., 2018; Uchoa et al., 2017; Cappart et al., 2021; Da Silva and Urrutia, 2010).

SCOPE, LIMITATIONS & FUTURE WORK Deep learning for combinatorial optimization is a recent research direction, which could significantly impact the way practical optimization problems get solved in the future. Currently, however, it is still hard to beat most SOTA problem specific solvers from the OR community. Despite our success for TSPTW, DPDP is not yet a practical alternative in general, but we do consider our results as highly encouraging for further research. We believe such research could yield significant further improvement by addressing key current limitations: (1) the scalability to larger instances, (2) the dependency on example solutions and (3) the heuristic nature of the scoring function. First, while 100 nodes is not far from the size of common benchmarks (100 - 1000 for VRP (Uchoa et al., 2017) and 20 - 200 for TSPTW (Da Silva and Urrutia, 2010)), scaling is a challenge, mainly due to the 'fully-connected' $O(n^2)$ graph neural network. Future work could reduce this complexity following e.g. Lee et al. (2019). The dependency on example solutions from an existing solver also becomes more prominent for larger instances, but could potentially be removed by 'bootstrapping' using DP itself as we, in some sense, have done for TSPTW (see Section 4.3.4). Future work could iterate this process to train the model 'tabula rasa' (without example solutions), where DP could be seen analogous to MCTS in *AlphaZero* (Silver et al., 2018). Lastly, the heat + potential score function is a well-motivated but heuristic function that was manually designed as a function of the predicted heatmap. While it worked well for the three problems we considered, it may need suitable adaption for other problems. Training this function end-to-end (Daumé III and Marcu, 2005; Wiseman and Rush, 2016), while keeping a low computational footprint, would be an interesting topic for future work.

Part II

SAMPLING AND STATISTICAL ESTIMATION

SUMMARY OF PART II

In Part i of this thesis, we have introduced machine learning based approaches for solving vehicle routing problems, based on deep neural network models. These models represent probability distributions over *combinatorial spaces of solutions* and, from a machine learning perspective, can be seen as *directed (acyclic) graphical models*. To train such models efficiently, we need techniques for drawing samples and estimating the gradient of an objective or loss function with respect to the model parameters.

The problems considered in Part i are mainly *deterministic* optimization problems, for which there is no value in sampling the same solution twice. Therefore, in Part ii of this thesis, we develop techniques for efficient *sampling without replacement* from graphical models, and estimating gradients using such samples without replacement. These techniques are generally applicable: they can, for example, also be used with machine translation models (Ott et al., 2019) or *latent variable models* with structured latent spaces (Yin et al., 2019). Therefore, we describe these techniques in general machine learning terminology and also consider experimental settings beyond combinatorial optimization.

First, in Chapter 5, we introduce *ancestral Gumbel-top-k sampling* (Kool et al., 2020a), as a method for drawing unique samples (i.e. samples without replacement) from directed acyclic graphical models over discrete (combinatorial) domains. Ancestral Gumbel-top-k sampling is a generalization of *stochastic beam search* (Kool et al., 2019c), which was designed specifically for *sequence models*: neural networks that produce sequences such as sentences or the attention model in Chapter 3. We analyze the properties of ancestral Gumbel-top-k sampling and, as an example of its generality, we show how it can be used to draw diverse samples from machine translation models and estimate the quality of generated translations.

In Chapter 6, we build on ancestral Gumbel-top-k sampling, as we derive an estimator that can be used to estimate the gradient of model parameters using a set of samples without replacement. We show how this can be used to improve training performance of a latent variable model (Yin et al., 2019) and finally to improve training speed and final performance of the attention model for the travelling salesman problem introduced in Chapter 3.

5 ANCESTRAL GUMBEL-TOP-k SAMPLING

We develop ancestral Gumbel-top-k sampling: a generic and efficient method for sampling without replacement from discrete-valued Bayesian networks, which includes multivariate discrete distributions, Markov chains and sequence models. The method uses an extension of the Gumbel-max trick to sample without replacement by finding the top k of perturbed log-probabilities among all possible configurations of a Bayesian network. Despite the exponentially large domain, the algorithm has a complexity linear *in the number of variables and sample size k. Our algorithm allows to set the number of parallel processors m, to trade off the number of iterations versus the total cost (iterations times m) of running the algorithm. For $m = 1$ the algorithm has minimum total cost, whereas for $m = k$ the number of iterations is minimized, and the resulting algorithm is known as* stochastic beam search.[1] *We provide extensions of the algorithm and discuss a number of related algorithms. We analyze the properties of ancestral Gumbel-top-k sampling and compare against alternatives on randomly generated Bayesian networks with different levels of connectivity. In the context of (deep) sequence models, we show its use as a method to generate diverse but high-quality translations and statistical estimates of translation quality and entropy.*

5.1 INTRODUCTION

Sampling from graphical models is a widely studied problem in machine learning. In many applications, such as neural machine translation, one may desire to obtain *multiple* samples, but wish to avoid duplicates, i.e. *sample without replacement*. In general, this is non-trivial: for example rejection sampling may take long if entropy is low.

In this chapter, we extend the idea of sampling through optimization (Papandreou and Yuille, 2011; Hazan and Jaakkola, 2012; Tarlow et al., 2012; Hazan et al., 2013; Ermon et al., 2013; Maddison et al., 2014; Chen and Ghahramani, 2016; Balog et al., 2017) to generate multiple samples *without replacement*. The most well-known example of sampling by optimization is the Gumbel-max trick (Gumbel, 1954; Maddison et al., 2014), which samples from the categorical distribution by optimizing (i.e. taking the argmax of) log-probabilities perturbed with independent Gumbel noise. Whereas the Gumbel-max trick only considers the argmax (top 1), taking the *top k* of perturbed log-probabilities actually results in a sample *without replacement* from the categorical distribution (Yellott, 1977; Vieira, 2014).

1 This chapter presents ancestral Gumbel-top-k sampling (Kool et al., 2020a), which was developed as an extension of stochastic beam search Kool et al. (2019c).

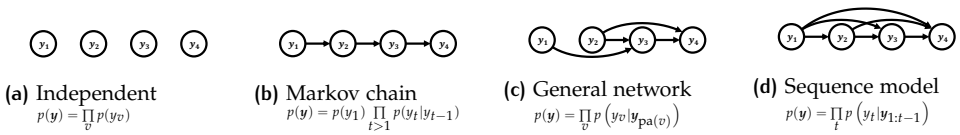


Figure 11: Examples of Bayesian networks.

We consider sampling from discrete-valued Bayesian networks (since sampling without replacement from continuous domains is trivial), which means that we sample from a discrete multivariate distribution which is represented by a probabilistic directed acyclic graphical model (for examples, see Figure 11). By treating each possible configuration of the variables in the network as a category in a single (flat) categorical distribution, we can use 'Gumbel-top-k sampling' to sample k configurations without replacement. To efficiently sample from the exponentially large domain, we use a top-down sampling procedure (Maddison et al., 2014) combined with an efficient branch-and-bound algorithm, which runs in time *linear* in the number of variables and number of samples.

The algorithm presented in this chapter, which we refer to as *ancestral Gumbel-top-k sampling*, is a generalization of *stochastic beam search* (originally presented in Kool et al. (2019c)), which allows to trade off 'parallelizability' against total required computation and is applicable to general Bayesian networks. As such, it serves the same purposes and can be used to generate representative and unique sequences from sequence models, for example for tasks such as neural machine translation (Sutskever et al., 2014; Bahdanau et al., 2015) and image captioning (Vinyals et al., 2015b), where the diversity can be controlled by the sampling temperature. Additionally, being a sampling method, it can be used to construct statistical estimators as we show in Section 5.5. As we will show in Chapter 6, the ability to sample without replacement enables the construction of lower variance gradient estimators (Kool et al., 2019b; Kool et al., 2020b).

5.2 PRELIMINARIES

This section introduces Bayesian networks, deep learning and the Gumbel-max trick.

5.2.1 Bayesian networks

A Bayesian network, also known as *belief network*, is a probabilistic directed acyclic graphical model that represents a joint probability distribution over a set of variables, which are nodes in a directed acyclic graph. We index the variables by $v \in \mathcal{V}$, where a node v can take values $y_v \in D_v$. For an arbitrary subset $\mathcal{S} \subseteq \mathcal{V}$, we write the corresponding set of values as $\boldsymbol{y}_\mathcal{S} = (y_v : v \in \mathcal{S})$, with domain $D_\mathcal{S} = \prod_{v \in \mathcal{S}} D_v$. For the complete network we write $\boldsymbol{y} = \boldsymbol{y}_\mathcal{V}$ with domain $D = D_\mathcal{V}$. The probability

distribution for y_v is defined conditionally on the parents $\text{pa}(v) \subseteq V \setminus \{v\}$, with values $\boldsymbol{y}_{\text{pa}(v)}$. This way, the distribution $p(\boldsymbol{y})$ is given by

$$p(\boldsymbol{y}) = \prod_{v \in \mathcal{V}} p\left(y_v | \boldsymbol{y}_{\text{pa}(v)}\right). \tag{23}$$

Any directed acyclic graph has at least one *topological* order (see e.g. Kahn (1962)), which is an ordering in which each node is preceded by its parent nodes. For an example of a Bayesian network in topological order, see Figure 11c. Examples of Bayesian networks include *multivariate categorical distributions* where $y_v, v \in \mathcal{V}$ are independent (Figure 11a), *Markov chains* where y_t depends only on y_{t-1} (assuming $\mathcal{V} = \{1, ..., T\}$), and (finite length) *sequence models* where y_t depends on the complete 'prefix' $\boldsymbol{y}_{1:t-1}$ and the topological order is natural.

To keep notation compact, in general, we will not make the distinction between variables and their realizations. As each variable y_v has a finite domain D_v, there is a finite number of possible realizations, or *configurations*, for the complete Bayesian network, specified by the domain $D = \prod_{v \in \mathcal{V}} D_v$. A configuration $\boldsymbol{y} \in D$ has probability given by equation 23 and therefore, ignoring the graphical structure, we can treat $\boldsymbol{y}$ as a 'flat' categorical variable over the domain D, where each category corresponds to a possible configuration $\boldsymbol{y}$.

5.2.2 Deep learning

Our focus is on modern deep learning (LeCun et al., 2015) applications, especially sampling from models represented as (discrete-valued) *stochastic computation graphs* (Schulman et al., 2015), which can be considered Bayesian networks. Such models specify conditional distributions for variables using neural networks, dependent on parameters $\boldsymbol{\theta}$ and an input or *context* $\boldsymbol{x}$. In a discrete setting, such models usually output probabilities for all $y_v \in D_v$ in a single *model evaluation*, by computing a *softmax* (with temperature $\tau \geq 0$) over unnormalized log-probabilities $\phi_{\boldsymbol{\theta}}(y_v | \boldsymbol{y}_{\text{pa}(v)}, \boldsymbol{x})$:

$$p_{\boldsymbol{\theta}}(y_v | \boldsymbol{y}_{\text{pa}(v)}, \boldsymbol{x}) = \frac{\exp\left(\phi_{\boldsymbol{\theta}}(y_v | \boldsymbol{y}_{\text{pa}(v)}) / \tau\right)}{\sum_{y'_v \in D_v} \exp\left(\phi_{\boldsymbol{\theta}}(y'_v | \boldsymbol{y}_{\text{pa}(v)}) / \tau\right)} \quad \forall y_v \in D_v. \tag{24}$$

In deep learning, model evaluations involve millions of computations, which is why we seek to minimize them. Additionally, to make efficient use of modern hardware, algorithms should be efficiently *parallelizable*. In the remainder of this chapter, we are concerned with sampling (without replacement) given fixed values of $\boldsymbol{\theta}$ and $\boldsymbol{x}$ so we will omit these in the notation.

5.2.3 The Gumbel-max trick

Treating the Bayesian network as specifying a categorical distribution, we can use the Gumbel-max trick (Gumbel, 1954; Maddison et al., 2014) to sample from it by finding the configuration $\boldsymbol{y}$ with the largest *perturbed log-probability*. We define $\phi_{\boldsymbol{y}}$ as the *log-probability* of $\boldsymbol{y} \in D$:

$$\phi_{\boldsymbol{y}} = \log p(\boldsymbol{y}) = \sum_{v \in \mathcal{V}} \log p\left(y_v | \boldsymbol{y}_{\mathrm{pa}(v)}\right). \tag{25}$$

Next we define $G_{\phi_{\boldsymbol{y}}}$ as the *perturbed log-probability* of $\boldsymbol{y}$, which is obtained by adding (independent!) Gumbel noise $G_{\boldsymbol{y}} \sim \mathrm{Gumbel}(0)$ to $\phi_{\boldsymbol{y}}$, where $\mathrm{Gumbel}(\phi)$ is the Gumbel distribution with CDF

$$F_{\phi}(g) = \exp(-\exp(\phi - g)). \tag{26}$$

Using inverse transform sampling, this noise is generated as $G_{\boldsymbol{y}} = F_0^{-1}(U_{\boldsymbol{y}}) = -\log(-\log U_{\boldsymbol{y}})$, where $U_{\boldsymbol{y}} \sim \mathrm{Uniform}(0, 1)$. By the shifting property of the Gumbel distribution, we have

$$G_{\phi_{\boldsymbol{y}}} := G_{\boldsymbol{y}} + \phi_{\boldsymbol{y}} = \phi_{\boldsymbol{y}} - \log(-\log U_{\boldsymbol{y}}) \sim \mathrm{Gumbel}(\phi_{\boldsymbol{y}}). \tag{27}$$

For any subset $B \subseteq D$ it holds that (Maddison et al., 2014)

$$\max_{\boldsymbol{y} \in B} G_{\phi_{\boldsymbol{y}}} \sim \mathrm{Gumbel}\left(\log \sum_{\boldsymbol{y} \in B} \exp \phi_{\boldsymbol{y}}\right), \tag{28}$$

$$\arg\max_{\boldsymbol{y} \in B} G_{\phi_{\boldsymbol{y}}} \sim \mathrm{Categorical}\left(\frac{\exp \phi_{\boldsymbol{y}}}{\sum_{\boldsymbol{y}' \in B} \exp \phi_{\boldsymbol{y}'}}, \boldsymbol{y} \in B\right). \tag{29}$$

Equation 28 is a useful property that states that the maximum of a set of Gumbel variables is a Gumbel variable with as location the LOGSUMEXP of the individual Gumbel locations. Equation 29 states the most important result: the configuration $\boldsymbol{y}$ corresponding to the largest perturbed log-probability is a sample from the desired categorical distribution (since $\exp \phi_{\boldsymbol{y}} = p(\boldsymbol{y})$). Additionally, the max (equation 28) and argmax (equation 29) are independent, which is an important property that is used in this chapter. For details, see Maddison et al. (2014). Figure 12 gives a visual illustration of the Gumbel-max trick.

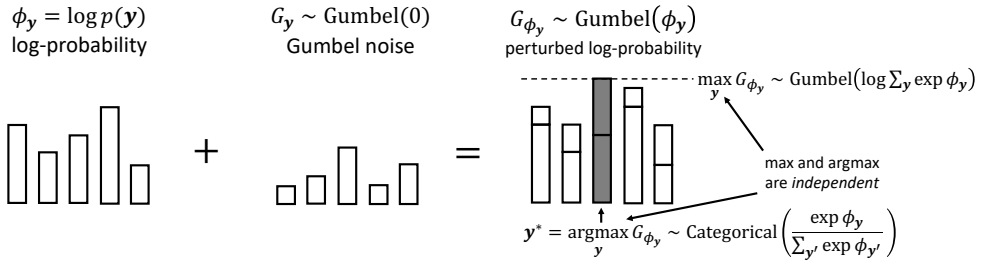


Figure 12: The Gumbel-max trick: the argmax of perturbed log-probabilities has a categorical distribution. The maximum has an independent Gumbel distribution.

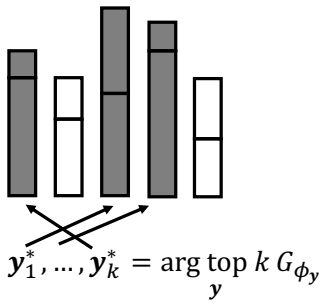


Figure 13: Gumbel-top-k sampling: the top k perturbed log-probabilities are a sample without replacement.

5.2.4 Gumbel-top-k sampling

An extension of the Gumbel-max trick can be used to sample from the categorical distribution without replacement (Yellott, 1977; Vieira, 2014). To this end, let $\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_k^* = \arg\operatorname{top} k\, G_{\phi_{\boldsymbol{y}}}$, i.e. $\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_k^*$ are the configurations with the k largest perturbed log-probabilities in decreasing order (see Figure 13). Denoting with $D_j^* = D \setminus \{\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_{j-1}^*\}$ the domain (without replacement) for the j-th sampled element, the probability for this *ordered sample without replacement* is given by

$$p\left(\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_k^*\right) = \prod_{j=1}^{k} p\left(\boldsymbol{y}_j^* \middle| \boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_{j-1}^*\right) \tag{30}$$

$$= \prod_{j=1}^{k} P\left(\boldsymbol{y}_j^* = \arg\max_{\boldsymbol{y} \in D_j^*} G_{\phi_{\boldsymbol{y}}} \middle| \max_{\boldsymbol{y} \in D_j^*} G_{\phi_{\boldsymbol{y}}} < G_{\phi_{\boldsymbol{y}_{j-1}^*}}\right) \tag{31}$$

$$= \prod_{j=1}^{k} P\left(\boldsymbol{y}_j^* = \arg\max_{\boldsymbol{y} \in D_j^*} G_{\phi_{\boldsymbol{y}}}\right) \tag{32}$$

$$= \prod_{j=1}^{k} \frac{\exp \phi_{\boldsymbol{y}_j^*}}{\sum_{\boldsymbol{y} \in D_j^*} \exp \phi_{\boldsymbol{y}}} \tag{33}$$

$$= \prod_{j=1}^{k} \frac{p(\boldsymbol{y}_j^*)}{1 - \sum_{\ell=1}^{j-1} p(\boldsymbol{y}_\ell^*)}. \tag{34}$$

To understand the derivation, note that conditioning on $\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_{j-1}^*$ means that $\boldsymbol{y}_1^*, \ldots, \boldsymbol{y}_{j-1}^*$ are the configurations with the $j-1$ largest perturbed log-probabilities, so $\boldsymbol{y}_j^*$, the configuration with the j-th largest perturbed log-probability, must be the arg max of the remaining log-probabilities. Additionally, we know that $\max_{\boldsymbol{y} \in D_j^*} G_{\phi_{\boldsymbol{y}}}$, the maximum of the remaining log-probabilities, must be smaller than $G_{\phi_{\boldsymbol{y}_{j-1}^*}}$, which is the smallest of the $j-1$ largest perturbed log-probabilities. This allows us to rewrite equation 30 as equation 31. The step from equation 31 to equation 32 follows from the independence of the max and arg max (Section 5.2.3) and the step from equation 32 to equation 33 uses the Gumbel-max trick.

The form of equation 33 is also known as the Plackett-Luce model (Plackett, 1975; Luce, 1959) and the form of equation 34 highlights its interpretation as sampling without replacement, where the probabilities get renormalized after each sampled

configuration. We refer to sampling without replacement by taking the top k of Gumbel perturbed log-probabilities as *Gumbel-top-k sampling*. This is mathematically equivalent to weighted reservoir sampling (Efraimidis and Spirakis, 2006) which can be seen as a streaming implementation of Gumbel-top-k sampling.

5.3 ANCESTRAL GUMBEL-TOP-k SAMPLING

Ancestral Gumbel-top-k sampling is an efficient implementation of Gumbel-top-k sampling for sampling from a probability distribution specified as a Bayesian network. It exploits the graphical structure to sample variables one at a time, conditionally on their parent variables, similar to ancestral sampling. The algorithm finds the top k configurations with largest perturbed log-probabilities (which determine the sample without replacement) *implicitly*, i.e. without sampling perturbed log-probabilities for all possible configurations of the Bayesian network. It does so by bounding the perturbed log-probabilities for parts of the domain, such that they can be pruned from the search if they are guaranteed to not contain a top k perturbed log-probability. Therefore, it can be considered a *branch-and-bound* algorithm (see e.g. Lawler and Wood (1966)).

To derive ancestral Gumbel-top-k sampling, we start with *explicit* Gumbel-top-k sampling to sample k configurations $\boldsymbol{y}$ without replacement from the Bayesian network. This requires instantiating *all* configurations $\boldsymbol{y} \in D$, sampling their perturbed log-probabilities and finding the k largest to obtain the sample. See for an example Figure 14, where the leaf nodes represent all 8 possible configurations for a Bayesian network with 3 binary variables. As the size of the domain D is exponential in the number of variables $|\mathcal{V}|$, this is not practically feasible in general, and we will derive ancestral Gumbel-top-k sampling as an alternative, equivalent, sampling procedure that does *not* require to instantiate the complete domain. Instead, it uses a *top-down* sampling procedure to sample perturbed log-probabilities for *partial configurations* $\boldsymbol{y}_S$ (internal nodes in Figure 14) first, which is equivalent but allows to obtain a bound on the perturbed log-probabilities of completions of $\boldsymbol{y}_S$ (descendants of $\boldsymbol{y}_S$ in the tree). This allows the dashed parts of the tree in Figure 14 to be pruned from the search, while obtaining the same result.

5.3.1 The probability tree diagram

To help develop our theory, we will first assume a *fixed* topological order of the nodes $\mathcal{V} = \{1, ..., T\}$. For $t \leq T$, we will use the notation $\boldsymbol{y}_{1:t} = \boldsymbol{y}_{\{1,...,t\}} = (y_1, ..., y_t)$ to indicate a *prefix* of $\boldsymbol{y}$, which is a *partial configuration* of the Bayesian network with domain $D_{1:t} = \prod_{v \in \{1,...,t\}} D_v$. This topological order allows us to represent possible configurations of the Bayesian network as the *probability tree diagram* in Figure 14, where level t has $|D_{1:t}|$ nodes: one node for each possible partial configuration $\boldsymbol{y}_{1:t} \in D_{1:t}$. Given a partial configuration $\boldsymbol{y}_{1:t}$ of the nodes $\{1, ..., t\}$, let $\mathcal{S}$ be a superset of

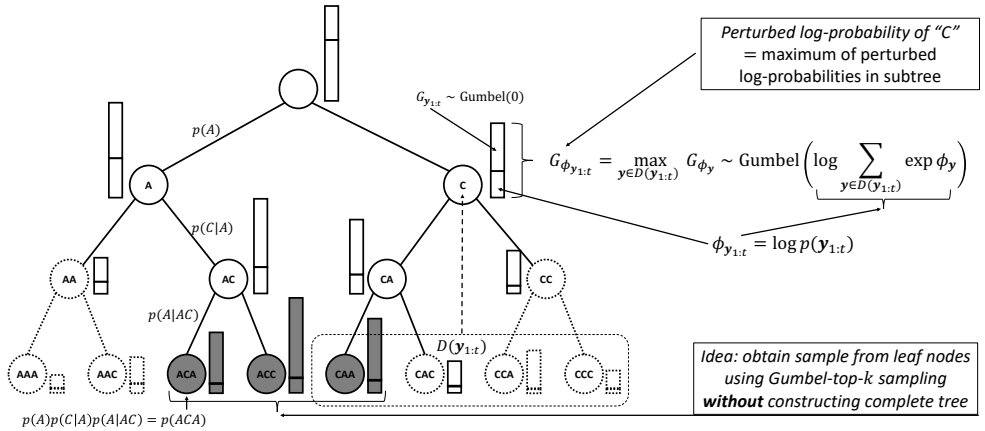


Figure 14: Example of Gumbel-top-k sampling ($k = 3$) for a Bayesian network $p(\boldsymbol{y}) = p(y_1)p(y_2|y_1)p(y_3|y_1, y_2)$, represented by *probability tree diagram*. The shaded leaf nodes correspond to the configurations $\boldsymbol{y} \in D$ with the largest perturbed log-probabilities, which is the resulting sample without replacement. We also indicate the perturbed log-probabilities for partial configurations $\boldsymbol{y}_{1:t}$ (internal nodes in the tree diagram), which are the maximum of the perturbed log-probabilities in the subtree. Using top-down sampling, the dashed parts of the tree do not need to be instantiated.

those nodes (i.e. $\{1, ..., t\} \subseteq \mathcal{S}$) and let $D_{\mathcal{S}|\boldsymbol{y}_{1:t}}$ be the domain of configurations $\boldsymbol{y}_{\mathcal{S}}$ given the partial assignment $\boldsymbol{y}_{1:t}$:

$$D_{\mathcal{S}|\boldsymbol{y}_{1:t}} = \{\boldsymbol{y}'_{\mathcal{S}} \in D_{\mathcal{S}} : \boldsymbol{y}'_{1:t} = \boldsymbol{y}_{1:t}\} = \prod_{v \in \{1,...,t\}} \{y_v\} \times D_{\mathcal{S}\setminus\{1,...,t\}}.$$

In particular, for $\mathcal{S} = \{1, ..., t+1\}$, $D_{1:t+1|\boldsymbol{y}_{1:t}} = \prod_{v \in \{1,...,t\}} \{y_v\} \times D_{t+1}$ is the set of possible extensions of $\boldsymbol{y}_{1:t}$ by the variable y_{t+1}, which defines the set of *direct child nodes* of $\boldsymbol{y}_{1:t}$ in the tree diagram. $D_{|\boldsymbol{y}_{1:t}} = D_{\mathcal{V}|\boldsymbol{y}_{1:t}}$ is the set of possible *complete configurations* compatible with (or given) $\boldsymbol{y}_{1:t}$, which corresponds to the set of leaf nodes in the subtree rooted at $\boldsymbol{y}_{1:t}$ in the tree diagram.

We define the *marginal probability* for the partial configuration $\boldsymbol{y}_{1:t}$ by marginalizing over all possible configurations $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$ that match the partial configuration:

$$\begin{aligned}
p(\boldsymbol{y}_{1:t}) &= \sum_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} p(\boldsymbol{y}) && (35)\\
&= \sum_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} \prod_{v \in \{1,...,t\}} p\left(y_v|\boldsymbol{y}_{\text{pa}(v)}\right) \prod_{v \in \{t+1,...,T\}} p\left(y_v|\boldsymbol{y}_{\text{pa}(v)}\right)\\
&= \prod_{v \in \{1,...,t\}} p\left(y_v|\boldsymbol{y}_{\text{pa}(v)}\right) \sum_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} \prod_{v \in \{t+1,...,T\}} p\left(y_v|\boldsymbol{y}_{\text{pa}(v)}\right)\\
&= \prod_{v \in \{1,...,t\}} p\left(y_v|\boldsymbol{y}_{\text{pa}(v)}\right)\\
&= p(\boldsymbol{y}_{1:t-1}) p\left(y_t|\boldsymbol{y}_{\text{pa}(t)}\right). && (36)
\end{aligned}$$

Note that $\boldsymbol{y}_{1:t-1}$ is the direct parent of the node $\boldsymbol{y}_{1:t}$ so equation 36 allows efficient computation of the probability $p(\boldsymbol{y}_{1:t-1})$ by multiplying the conditional probability $p\left(y_t|\boldsymbol{y}_{\text{pa}(t)}\right)$ (note that $\text{pa}(t) \subseteq \{1, ..., t-1\}$) with the marginal probability $p(\boldsymbol{y}_{1:t-1})$ of the parent in the tree diagram.

5.3.2 Explicit Gumbel-top-k sampling

By using Gumbel-top-k sampling (see Section 5.2.4) *explicitly*, we can (inefficiently!) obtain a sample without replacement of size k from the Bayesian network as follows:

- Compute $\phi_{\boldsymbol{y}} = \log p_{\boldsymbol{\theta}}(\boldsymbol{y})$ for *all configurations* $\boldsymbol{y} \in D$. This means that the complete tree diagram is instantiated, as in Figure 14.
- For $\boldsymbol{y} \in D$, sample $G_{\phi_{\boldsymbol{y}}} \sim \text{Gumbel}(\phi_{\boldsymbol{y}})$, so $G_{\phi_{\boldsymbol{y}}}$ is the perturbed log-probability of $\boldsymbol{y}$.
- Let $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_k^* = \arg\operatorname{top} k\, G_{\phi_{\boldsymbol{y}}}$, then $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_k^*$ is a sample of configurations from the distribution $p(\boldsymbol{y})$ (equation 23) without replacement.

Note that the result corresponds to a subset of leaf nodes in the tree diagram. As explicit Gumbel-top-k sampling requires to instantiate the complete tree diagram to compute perturbed log-probabilities for all leaf nodes, this is computationally prohibitive. Therefore, we construct an equivalent process that we call ancestral Gumbel-top-k sampling, which *implicitly* finds the configurations $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_k^*$ with computation that is only *linear* in the number of samples k and the number of variables T.

5.3.3 Perturbed log-probabilities of partial configurations

So far, we have only defined the perturbed log-probabilities $G_{\phi_{\boldsymbol{y}}}$ for *complete configurations* $\boldsymbol{y} \in D$, corresponding to leaf nodes in the tree diagram. To derive ancestral Gumbel-top-k sampling, we find it convenient to also define the perturbed log-probabilities for *partial configurations* $\boldsymbol{y}_{1:t} \in D_{1:t}$, which correspond to internal nodes in the tree diagram. From equation 35, it follows that a partial configuration $\boldsymbol{y}_{1:t}$ has log-probability

$$\phi_{\boldsymbol{y}_{1:t}} = \log p(\boldsymbol{y}_{1:t}) = \log \sum_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} p(\boldsymbol{y}) = \log \sum_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} \exp \phi_{\boldsymbol{y}}.$$

For a partial configuration $\boldsymbol{y}_{1:t}$, we now consider the *maximum of the perturbed log-probabilities* $G_{\phi_{\boldsymbol{y}}}$ of compatible complete configurations $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$. This corresponds to the maximum of perturbed log-probabilities in the subtree below $\boldsymbol{y}_{1:t}$ in Figure 14. By the Gumbel-max trick (equation 28), it has a Gumbel distribution with location $\phi_{\boldsymbol{y}_{1:t}}$. Therefore, we use the notation $G_{\phi_{\boldsymbol{y}_{1:t}}}$:

$$G_{\phi_{\boldsymbol{y}_{1:t}}} = \max_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}} G_{\phi_{\boldsymbol{y}}} \sim \text{Gumbel}(\phi_{\boldsymbol{y}_{1:t}}). \tag{37}$$

Since $G_{\phi_{\boldsymbol{y}_{1:t}}} \sim \text{Gumbel}(\phi_{\boldsymbol{y}_{1:t}})$, we can rewrite it as

$$G_{\phi_{\boldsymbol{y}_{1:t}}} = \phi_{\boldsymbol{y}_{1:t}} + G_{\boldsymbol{y}_{1:t}} \tag{38}$$

where we have defined $G_{\boldsymbol{y}_{1:t}} \sim \text{Gumbel}(0)$. Equation 38 reveals that we can interpret $G_{\phi_{\boldsymbol{y}_{1:t}}}$ as the *perturbed log-probability* of partial configuration $\boldsymbol{y}_{1:t}$. Thus, the perturbed

log-probability of a partial configuration $\boldsymbol{y}_{1:t}$ is the maximum of perturbed log-probabilities of its possible completions $D_{|\boldsymbol{y}_{1:t}}$. This relation is key to the top-down sampling algorithm that we will derive.

Looking at the tree diagram in Figure 14, the maximum below a node $\boldsymbol{y}_{1:t}$ must be attained in one of the subtrees, such that the perturbed log-probability of $\boldsymbol{y}_{1:t}$ must be the maximum of perturbed log-probabilities of its possible one-variable extensions (children) $\boldsymbol{y}_{1:t+1} \in D_{1:t+1|\boldsymbol{y}_{1:t}}$. Formally, we can partition the domain $D_{|\boldsymbol{y}_{1:t}}$ based on the value of $y_{t+1} \in D_{t+1}$ as $D_{|\boldsymbol{y}_{1:t}} = \bigcup_{y_{t+1} \in D_{t+1}} D_{|\boldsymbol{y}_{1:t+1}}$, such that we can write equation 37 as

$$G_{\phi_{\boldsymbol{y}_{1:t}}} = \max_{y_{t+1} \in D_{t+1}} \max_{\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t+1}}} G_{\phi_{\boldsymbol{y}}} = \max_{y_{t+1} \in D_{t+1}} G_{\phi_{\boldsymbol{y}_{1:t+1}}}. \tag{39}$$

This means that we can compute $G_{\phi_{\boldsymbol{y}_{1:t}}}$ for all nodes in the tree recursively, sampling $G_{\phi_{\boldsymbol{y}}}$ for the leaf nodes $\boldsymbol{y} \in D$ and computing equation 39 recursively up the tree. We refer to this procedure as *bottom-up* sampling of the perturbed log-probabilities. Here we consider computing $G_{\phi_{\boldsymbol{y}_{1:t}}}$ using equation 39 as a sampling step because $G_{\phi_{\boldsymbol{y}_{1:t}}}$ is a random variable which has a degenerate (constant) distribution by conditioning on the children.

5.3.4 Top-down sampling of perturbed log-probabilities

As an alternative to sampling the perturbed log-probabilities 'bottom-up', we can reverse the process. Starting from the root of the tree diagram with an empty configuration, which we denote by $\boldsymbol{y}_\varnothing$ with length $t = 0$, we conditionally sample the perturbed log-probabilities for the children recursively.

For the root $\boldsymbol{y}_\varnothing$ at level $t = 0$ it holds that $D_{|\boldsymbol{y}_\varnothing} = D$ and the log-probability is

$$\phi_{\boldsymbol{y}_\varnothing} = \log \sum_{\boldsymbol{y} \in D} \exp \phi_{\boldsymbol{y}} = \log \sum_{\boldsymbol{y} \in D} p(\boldsymbol{y}) = \log 1 = 0.$$

This means that we can sample its perturbed log-probability $G_{\phi_{\boldsymbol{y}_\varnothing}} \sim \text{Gumbel}(0)$, or we can simply set $G_{\phi_{\boldsymbol{y}_\varnothing}} = 0$, which conditions the sampling process on the event that $\max_{\boldsymbol{y} \in D} G_{\phi_{\boldsymbol{y}}} = 0$, which does not affect the result since the sample is determined by the $\arg\max$ / $\arg\operatorname{top} k$, which is independent of the maximum (see Section 5.2.3).

Given the log-probability $\phi_{\boldsymbol{y}_{1:t}}$ of a node $\boldsymbol{y}_{1:t}$, we can efficiently compute the log-probabilities $\phi_{\boldsymbol{y}_{1:t+1}}$ for its children as $\phi_{\boldsymbol{y}_{1:t+1}} = \phi_{\boldsymbol{y}_{1:t}} + \log p(y_t | \boldsymbol{y}_{\text{pa}(t)})$ (this follows from equation 36). Similarly, given the *perturbed* log-probability $G_{\phi_{\boldsymbol{y}_{1:t}}}$ and the log-probabilities $\phi_{\boldsymbol{y}_{1:t+1}}$, we can *top-down* sample the perturbed log-probabilities $G_{\phi_{\boldsymbol{y}_{1:t+1}}}$ for each possible $y_{t+1} \in D_{t+1}$. However, there is a dependence between $G_{\phi_{\boldsymbol{y}_{1:t}}}$ and $G_{\phi_{\boldsymbol{y}_{1:t+1}}}$ given by $G_{\phi_{\boldsymbol{y}_{1:t}}} = \max_{y_{t+1} \in D_{t+1}} G_{\phi_{\boldsymbol{y}_{1:t+1}}}$ (equation 39). Therefore, when sampling $G_{\phi_{\boldsymbol{y}_{1:t+1}}}$, we need to make sure this dependency is satisfied, i.e. we need to sample a set of (independent) Gumbel variables *conditionally upon their maximum*.

5.3.5 Sampling a set of Gumbel variables conditionally on their maximum

To sample a set of Gumbel variables $G_{\phi_i} \sim \text{Gumbel}(\phi_i)$ with a given maximum T, we can:

- Sample G_{ϕ_i} for all i *independently*
- Let $Z = \max_i G_{\phi_i}$ be the observed maximum
- Let $\tilde{G}_{\phi_i} = F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(G_{\phi_i}))$, where $F_{\phi_i,Z}$ and $F^{-1}_{\phi_i,T}$ are the CDF and inverse CDF of truncated Gumbel Distributions.

The CDF of a Gumbel(ϕ) distribution truncated at T is given by

$$\begin{aligned} F_{\phi,T}(g) &= P(G \leq g | G \leq T) \\ &= \frac{P(G \leq g \cap G \leq T)}{P(G \leq T)} \\ &= \frac{P(G \leq \min(g, T))}{P(G \leq T)} \\ &= \frac{F_\phi(\min(g, T))}{F_\phi(T)}, \end{aligned}$$

where $F_\phi(g) = \exp(-\exp(\phi - g))$ is the CDF of the (untruncated) Gumbel(ϕ) distribution (equation 26). The *inverse* CDF of the truncated Gumbel distribution is given by

$$F^{-1}_{\phi,T}(u) = \phi - \log(\exp(\phi - T) - \log u),$$

such that the transformation $\tilde{G}_{\phi_i} = F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(G_{\phi_i}))$ can be written explicitly as

$$\begin{aligned} \tilde{G}_{\phi_i} &= F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(G_{\phi_i})) \\ &= \phi_i - \log(\exp(\phi_i - T) - \exp(\phi_i - Z) + \exp(\phi_i - G_{\phi_i})) \\ &= -\log(\exp(-T) - \exp(-Z) + \exp(-G_{\phi_i})). \end{aligned} \tag{40}$$

This shows that the samples with maximum Z are effectively 'shifted' towards their desired maximum T, in (negative) exponential space through the (inverse) truncated Gumbel CDF. See Appendix C.1 for a numerically stable implementation.

The validity of the sampling procedure is shown by conditioning on $\arg\max_j G_{\phi_j}$. For $i = \arg\max_j G_{\phi_j}$ it holds that

$$
\begin{aligned}
&P(\tilde{G}_{\phi_i} \leq g | i = \arg\max_j G_{\phi_j}) \\
=&\mathbb{E}_Z \left[P(\tilde{G}_{\phi_i} \leq g | i = \arg\max_j G_{\phi_j}, \max_j G_{\phi_j} = Z) \right] \\
=&\mathbb{E}_Z \left[P(\tilde{G}_{\phi_i} \leq g | G_{\phi_i} = Z) \right] \\
=&\mathbb{E}_Z \left[P(F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(G_{\phi_i})) \leq g | G_{\phi_i} = Z) \right] \\
=&\mathbb{E}_Z \left[P(F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(Z)) \leq g) \right] \\
=&\mathbb{E}_Z \left[P(T \leq g) \right] \\
=&P(T \leq g) \\
=&P(G_{\phi_i} \leq g | G_{\phi_i} = T) \\
=&P(G_{\phi_i} \leq g | \max_j G_{\phi_j} = T, i = \arg\max_j G_{\phi_j}).
\end{aligned}
$$

For $i \neq \arg\max_j G_{\phi_j}$ it holds that

$$
\begin{aligned}
&P(\tilde{G}_{\phi_i} \leq g | i \neq \arg\max_j G_{\phi_j}) \\
=&\mathbb{E}_Z \left[P(\tilde{G}_{\phi_i} \leq g | i \neq \arg\max_j G_{\phi_j}, \max_j G_{\phi_j} = Z) \right] \\
=&\mathbb{E}_Z \left[P(\tilde{G}_{\phi_i} \leq g | G_{\phi_i} < Z) \right] \\
=&\mathbb{E}_Z \left[P(F^{-1}_{\phi_i,T}(F_{\phi_i,Z}(G_{\phi_i})) \leq g | G_{\phi_i} < Z) \right] \\
=&\mathbb{E}_Z \left[P(G_{\phi_i} \leq F^{-1}_{\phi_i,Z}(F_{\phi_i,T}(g)) | G_{\phi_i} < Z) \right] \\
=&\mathbb{E}_Z \left[F_{\phi_i,Z}(F^{-1}_{\phi_i,Z}(F_{\phi_i,T}(g))) \right] \\
=&\mathbb{E}_Z \left[F_{\phi_i,T}(g) \right] \\
=&F_{\phi_i,T}(g) \\
=&P(G_{\phi_i} \leq g | G_{\phi_i} < T) \\
=&P(G_{\phi_i} \leq g | \max_j G_{\phi_j} = T, i \neq \arg\max_j G_{\phi_j}).
\end{aligned}
$$

Combining the two cases, it holds that

$$
\begin{aligned}
P(\tilde{G}_{\phi_i} \leq g) =&P(\tilde{G}_{\phi_i} \leq g | i = \arg\max_j G_{\phi_j}) P(i = \arg\max_j G_{\phi_j}) + \\
&P(\tilde{G}_{\phi_i} \leq g | i \neq \arg\max_j G_{\phi_j}) P(i \neq \arg\max_j G_{\phi_j}) \\
=&P(G_{\phi_i} \leq g | \max_j G_{\phi_j} = T, i = \arg\max_j G_{\phi_j}) P(i = \arg\max_j G_{\phi_j}) + \\
&P(G_{\phi_i} \leq g | \max_j G_{\phi_j} = T, i \neq \arg\max_j G_{\phi_j}) P(i \neq \arg\max_j G_{\phi_j}) \\
=&P(G_{\phi_i} \leq g | \max_j G_{\phi_j} = T).
\end{aligned}
$$

This procedure allows us to recursively sample $G_{\phi_{\boldsymbol{y}_{1:t}}}$ for the complete tree diagram top-down, which is equivalent to sampling the perturbed log-probabilities for the complete tree diagram bottom-up. This means that for the leaves, or complete configurations $\boldsymbol{y}$, it holds that $G_{\phi_{\boldsymbol{y}}} \sim \text{Gumbel}(\phi_{\boldsymbol{y}})$ *independently*. The benefit of the top-down sampling procedure is that we can choose to sample *only* the parts of the tree diagram needed to obtain the top k leaves.

5.3.6 Ancestral Gumbel-top-k sampling

At the heart of the ancestral Gumbel-top-k sampling is the idea that we can find the configurations corresponding to the top k perturbed log-probabilities, *without* instantiating the complete domain, by using top-down sampling. To do so, we maintain a *queue* Q of partial configurations $\boldsymbol{y}_{1:t}$, where no partial configuration $\boldsymbol{y}_{1:t} \in Q$ is a prefix of another one. By doing so, each element $\boldsymbol{y}_{1:t} \in Q$ can be seen as the root of a (disjoint) part of the domain $D_{|\boldsymbol{y}_{1:t}} \subseteq D$. We do this in such a way that the queue represents a *partitioning* of the complete domain, so $D = \bigcup_{\boldsymbol{y}_{1:t} \in Q} D_{|\boldsymbol{y}_{1:t}}$.

For each of the partial configurations in the queue, we also keep track of the perturbed log-probability $G_{\phi_{\boldsymbol{y}_{1:t}}}$, which is obtained by top-down sampling. The queue represents the set of leaf nodes of a partially constructed probability tree diagram (see Figure 15). At any point in time, we can remove an element $\boldsymbol{y}_{1:t}$ from the queue and add to the queue the set of extensions $\boldsymbol{y}_{1:t+1}$ for $y_{t+1} \in D^{t+1}$, for which we can sample the perturbed log-probabilities $G_{\phi_{\boldsymbol{y}_{1:t+1}}}$ conditionally upon the value of their parent $G_{\phi_{\boldsymbol{y}_{1:t}}}$, as in standard ancestral sampling. Repeating this process will ultimately result in a queue $Q = D$, sampling all complete configurations $\boldsymbol{y} \in D$, including their perturbed log-probabilities, such that the top k determines the sample without replacement. To improve efficiency, we can bound the size of the queue, limiting the total number of expansions, without changing the result.

5.3.7 Bounding of the queue size at k

Assuming we are interested in sampling k configurations without replacement, let $\kappa = \kappa(D)$ be the k-th largest of the perturbed log-probabilities $\{G_{\phi_{\boldsymbol{y}}} : \boldsymbol{y} \in D\}$ of complete configurations $\boldsymbol{y} \in D$. We call κ the *threshold* since, using Gumbel-top-k sampling, $\boldsymbol{y}$ will be part of the k samples without replacement if $G_{\phi_{\boldsymbol{y}}} \geq \kappa$.

Lemma 1. *(Lower bound) Let $\kappa(Q)$ be the k-th largest perturbed log-probability of (partial) configurations in the queue Q. Then $\kappa(Q)$ is a lower bound for κ, i.e. $\kappa \geq \kappa(Q)$.*

Proof. For each of the k largest perturbed log-probabilities $G_{\phi_{\boldsymbol{y}_{1:t}}}$ in the queue Q, it holds that $G_{\phi_{\boldsymbol{y}_{1:t}}} \geq \kappa(Q)$ (by definition). By the definition of $G_{\phi_{\boldsymbol{y}_{1:t}}}$ as the maximum of perturbed log-probabilities $G_{\phi_{\boldsymbol{y}}}$ for $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$ (equation 37), there must necessarily be a completion $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$ for which $G_{\phi_{\boldsymbol{y}}} = G_{\phi_{\boldsymbol{y}_{1:t}}} \geq \kappa(Q)$. Since for $\boldsymbol{y}_{1:t} \in Q$ the corresponding domains $D_{|\boldsymbol{y}_{1:t}}$ are disjoint, this means that there are (at least)

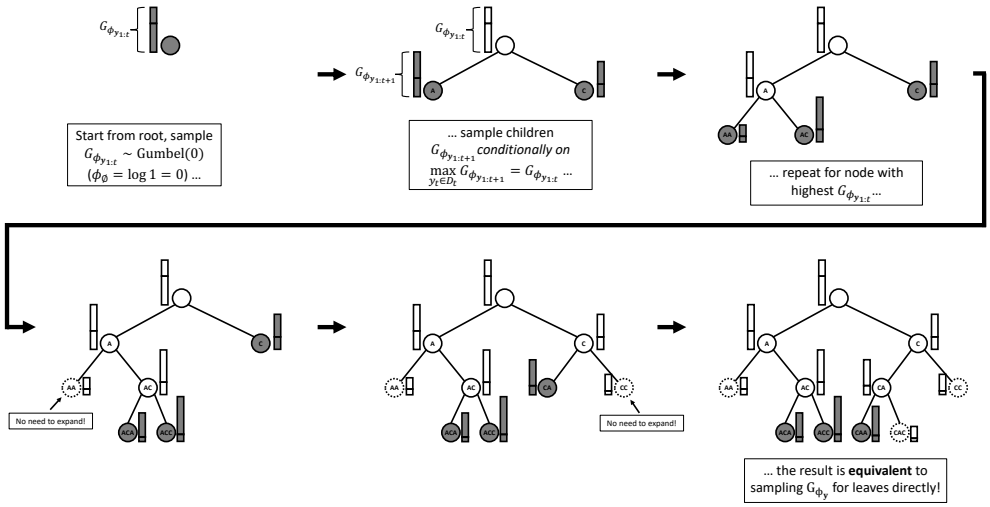


Figure 15: Ancestral Gumbel-top-k sampling, with $k = 3$ and $m = 1$. Configurations in the queue are shaded and in each iteration, the single ($m = 1$) incomplete configuration with the highest perturbed log-probability is expanded, sampling the perturbed log-probabilities for the extensions conditionally. This takes 5 iterations and has a total cost of 5 model evaluations to obtain $k = 3$ samples without replacement.

k distinct configurations $\boldsymbol{y} \in D$ such that $G_{\phi_y} \geq \kappa(Q)$. Since there are at least k configurations $\boldsymbol{y} \in D$ for which $G_{\phi_y} \geq \kappa(Q)$, and κ is the k-th largest perturbed log-probability in D, it must hold that $\kappa \geq \kappa(Q)$. □

Lemma 2. *(Upper bound) For $\boldsymbol{y}_{1:t} \in Q$, the perturbed log-probability $G_{\phi_{\boldsymbol{y}_{1:t}}}$ is an upper bound for the perturbed log-probabilities of its possible completions, i.e. $G_{\phi_{\boldsymbol{y}_{1:t}}} \geq G_{\phi_{\boldsymbol{y}}}$ for $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$.*

Proof. It follows directly from the the definition of $G_{\phi_{\boldsymbol{y}_{1:t}}}$ (equation 37) that

$$G_{\phi_{\boldsymbol{y}_{1:t}}} = \max_{\boldsymbol{y}' \in D_{|\boldsymbol{y}_{1:t}}} G_{\phi_{\boldsymbol{y}'}} \geq G_{\phi_{\boldsymbol{y}}} \quad \forall \boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}.$$

□

Theorem 3. *(Limit queue size) All except the k configurations with the largest perturbed log-probabilities in the queue Q can be discarded while still resulting in a sample without replacement of size k.*

Proof. For $\boldsymbol{y}_{1:t} \in Q$ that does *not* belong to one of the top k perturbed log-probabilities, it holds that $G_{\phi_{\boldsymbol{y}_{1:t}}} < \kappa(Q)$ (assuming uniqueness of $G_{\phi_{\boldsymbol{y}_{1:t}}}$ since the domain is continuous) such that for its possible completions $\boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}$ it holds that (using Lemma 1 and 2)

$$G_{\phi_{\boldsymbol{y}}} \leq G_{\phi_{\boldsymbol{y}_{1:t}}} < \kappa(Q) \leq \kappa \quad \forall \boldsymbol{y} \in D_{|\boldsymbol{y}_{1:t}}.$$

If $\boldsymbol{y}_{1:t}$ is not one of the top k in Q, then $G_{\phi_{\boldsymbol{y}}} < \kappa$ for all possible completions of $\boldsymbol{y}_{1:t}$, and any such completion will *not* be in the sample without replacement of size k, so $\boldsymbol{y}_{1:t}$ can be discarded from the queue without affecting the result. □

As a consequence of Theorem 3, if we desire to sample k configurations, then at any point in time, we need to keep at most the k partial configurations with largest perturbed log-probabilities in the queue. This can be implemented by maintaining Q as a priority queue of limited size k, where the priorities are given by the perturbed log-probabilities. Repeatedly removing the first *incomplete* configuration (with the highest perturbed log-probability) from the queue, and replacing it by its extensions, ultimately results in a queue with k complete configurations, corresponding to the k largest perturbed log-probabilities, as is illustrated in Figure 15. By its equivalence to explicit Gumbel-top-k sampling, this is a sample without replacement of size k (in order!). By repeatedly expanding the partial configuration with highest perturbed log-probability, the algorithm is a *best first search* algorithm.

In a practical implementation, we can directly 'yield' the first complete configuration once it is found, remove it from the queue and decrease the queue size limit by 1. This will yield the first sample in exactly T iterations (where T is the number of variables), as if we used standard ancestral sampling. Then, it will 'jump' back to the partial configuration $\boldsymbol{y}_{1:t}$ with the highest perturbed log-probability and repeat the best first search from there, lazily generating each next sample as the algorithm proceeds. By reusing partially sampled configurations, the algorithm evaluates the model for each possible partial configuration at most once. It is 'anytime optimal' in the sense that to yield $k' \leq k$ samples, it only computes model evaluations that are strictly necessary. The algorithm is a special case of Algorithm 2 with $m = 1$ (see Section 5.3.8) and a fixed variable selection strategy (Section 5.3.9).

Algorithm 2 AncestralGumbelTop$k$Sampling($p_{\boldsymbol{\theta}}$, k, m)

Input: Bayesian network $p_{\boldsymbol{\theta}}(y_v|\text{pa}(v))\ \forall v \in \mathcal{V}$, max. sample size k, parallel expansion parameter m, variable selection strategy $\mathfrak{V}$
Initialize RESULT and QUEUE empty
add $(\mathcal{S} = \varnothing, \boldsymbol{y}_{\mathcal{S}} = \varnothing, \phi_{\boldsymbol{y}_{\mathcal{S}}} = 0, G_{\phi_{\boldsymbol{y}_{\mathcal{S}}}} = 0)$ to QUEUE
while QUEUE not empty **do**
  EXPAND $\leftarrow$ take and remove top m from QUEUE according to $\tilde{G}$
  for $(\mathcal{S}, \boldsymbol{y}_{\mathcal{S}}, \phi_{\boldsymbol{y}_{\mathcal{S}}}, G_{\phi_{\boldsymbol{y}_{\mathcal{S}}}}) \in$ EXPAND (in parallel) **do**
    if $\mathcal{S} = \mathcal{V}$ **then**
      add $(\mathcal{S}, \boldsymbol{y}_{\mathcal{S}}, \phi_{\boldsymbol{y}_{\mathcal{S}}}, G_{\phi_{\boldsymbol{y}_{\mathcal{S}}}})$ to QUEUE
    else
      select v from $\{v \in \mathcal{V} \setminus \mathcal{S} : \text{pa}(v) \subseteq \mathcal{S}\}$ according to variable selection strategy $\mathfrak{V}$
      let $\mathcal{S}' = \mathcal{S} \cup \{v\}$
      compute $\phi_{\boldsymbol{y}_{\mathcal{S}'}} \leftarrow \phi_{\boldsymbol{y}_{\mathcal{S}}} + \log p_{\boldsymbol{\theta}}(y_v|\boldsymbol{y}_{pa(v)})\quad \forall y_v \in D_v$ (here $\boldsymbol{y}_{\mathcal{S}'} = \boldsymbol{y}_{\mathcal{S}} \cup \{y_v\}$)
      sample $G_{\phi_{\boldsymbol{y}_{\mathcal{S}'}}} \sim \text{Gumbel}(\phi_{\boldsymbol{y}_{\mathcal{S}'}})\quad \forall y_v \in D_v$ conditionally given maximum $G_{\phi_{\boldsymbol{y}_{\mathcal{S}}}}$
      add $(\mathcal{S}', \boldsymbol{y}_{\mathcal{S}'}, \phi_{\boldsymbol{y}_{\mathcal{S}'}}, G_{\phi_{\boldsymbol{y}_{\mathcal{S}'}}})$ to QUEUE for all $y_v \in D_v$
    end if
  end for
  QUEUE $\leftarrow$ take top k from QUEUE according to $\tilde{G}$
  while first element from QUEUE is complete, i.e. has $\mathcal{S} = \mathcal{V}$ **do**
    remove first element from QUEUE and add it to RESULT
    $k \leftarrow k - 1$
  end while
end while
Return RESULT

5.3.8 Parallelizing the algorithm

Instead of only expanding the first partial configuration in the queue, we can assume availability of $m \leq k$ parallel processors and expand m partial configurations *in parallel*, replacing all of them by their (disjoint) expansions. This results in fewer required iterations (known as the *span* or *depth*) of the algorithm, but an increase of total *cost* of the algorithm, which is the number of iterations times m. This is because of the cost incurred for 'idle' parallel processors if initially the queue size is smaller than m, and because of redundant model evaluations if nodes are expanded which, in a fully sequential setting, would have been 'pushed' off the queue by other expansions (see Figure 16 for an example).

For $m = k$, the complete queue gets replaced in every iteration by the top k of all expansions, and the resulting algorithm is a limited width *breadth first search* which was originally presented in Kool et al. (2019c) as *stochastic beam search* (Figure 16). While this algorithm requires the fewest iterations, as it guarantees k samples are generated in exactly T iterations (where T is the number of variables), it is the least efficient in terms of total cost.

Ultimately, the optimal choice of m depends on properties of the model and available parallel hardware, e.g. the number of GPUs[2] (or parallel processors on a single GPU) in modern deep learning applications. For example, if a single model evaluation can already fully utilize the GPUs (e.g. if it is a large and highly parallelizable neural network model), then setting $m = 1$ is optimal as this computes only strictly necessary model evaluations. On the other hand, if GPUs cannot be fully utilized for individual model evaluations, or additional GPUs are freely available, one should use $m = k$ to minimize the number of iterations which then directly translates to minimizing running time.

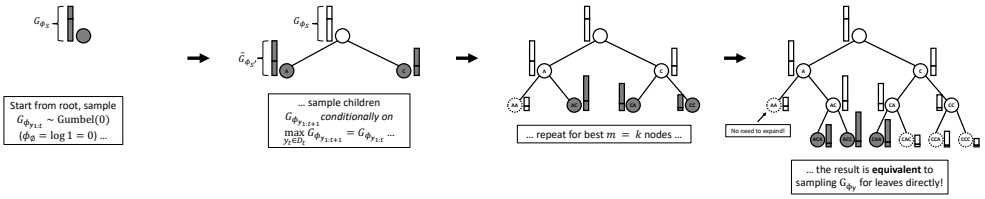


Figure 16: Ancestral Gumbel-top-k sampling, with $k = m$ $(= 3)$, also known as stochastic beam search (Kool et al., 2019c). This version expands in each iteration all partial configurations in the queue in parallel, traversing the tree diagram per level, i.e. using breadth first search. Compared to ancestral sampling with $m = 1$, this reduces the number of iterations from 5 to 3, but has a total cost of 3 iterations $\times$ 3 processors $= 9$. The total cost can be divided into a cost of 6 (rather than 5) for model evaluations, in this case since 'CC' is expanded unnecessarily, and a cost of 3 for idle processors if the queue (marked by shaded nodes) size is smaller than m: this cost is 2 in the first iteration and 1 in the second. The result is equivalent to using $m = 1$ (Figure 15) or any other $m < k$.

2 GPU is an abbreviation of *graphical processing unit*.

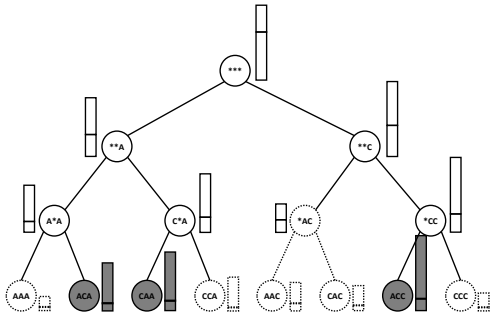


Figure 17: Example of top-down sampling using a dynamic variable selection strategy for a graphical model $p(\boldsymbol{y}) = p(y_1)p(y_2)p(y_3)$.

5.3.9 Variable selection strategy

We can consider a tree diagram as a tree representing the possible paths (from root to leaf) that standard ancestral sampling can take to generate a sample. In the tree diagram in Figure 14, each level t corresponds to the fixed variable y_t, such that the sample is always generated in the order $y_1, ..., y_T$. However, if the graphical model has multiple topological orderings, any such order yields a valid tree diagram and thus valid order for ancestral sampling. Moreover, the order does not have to be globally fixed, but can vary depending on the partial configuration sampled so far.

In general, when ancestral sampling, we are free to choose any variable y_t to sample next, as long as its parents $\boldsymbol{y}_{\mathrm{pa}(t)}$ have been sampled. For example, for a fully factorized model $p(\boldsymbol{y}) = p(y_1)p(y_2)p(y_3)$, we are free to start sampling with y_3, and, *depending on the value of* y_3, select either y_1 or y_2 to sample next. This is an example of a *variable selection strategy*, which is represented by the tree diagram in Figure 17. While the resulting distribution is the same, the variable selection strategy affects which parts of the tree can be discarded (dashed nodes in Figure 17) and thus the number of iterations that are required by the algorithm. Sampling low-entropy variables first is a good idea, as this allows larger partial configurations to be 'reused' for subsequent samples (see Section 5.5.1). Algorithm 2 is presented in the form that can take an arbitrary variable selection strategy.

5.4 RELATED ALGORITHMS

In this section we discuss a number of algorithms related to ancestral Gumbel-top-k sampling.

5.4.1 The 'trajectory generator'

For $m = 1$, our ancestral Gumbel-top-k sampling algorithm is similar to the 'trajectory generator' with $\epsilon = 0$ described by Lorberbom et al. (2020), where it is used

in the context of reinforcement learning (Sutton and Barto, 2018) to generate *direct policy gradients*. Our algorithm replaces a partial configuration $\boldsymbol{y}_{1:t}$ in the queue by *all* expansions $\boldsymbol{y}_{1:t+1}$, motivated by the idea that this is efficient if we consider computing $p(\boldsymbol{y}_{1:t+1}|\boldsymbol{y}_{1:t})$ for all y_{t+1} to be a single model evaluation, e.g. the forward pass of a neural network with a softmax output layer (equation 24). In the probability tree diagram, this means that if node is expanded, all childnodes are added to the tree. In the context of reinforcement learning, the algorithm by Lorberbom et al. (2020) is assumed to operate on a 'state-reward tree', where generating a new node corresponds to taking an action which requires an interaction with the environment. Therefore, they only add a *single* child node to the tree in each iteration.

In the implementation by Lorberbom et al. (2020), elements in the queue do not (only) represent leaf nodes of the tree, but the idea that they form a partition of the domain is still maintained. In particular, in our implementation, upon expansion we partition the domain $D_{|\boldsymbol{y}_{1:t}}$ represented by $\boldsymbol{y}_{1:t}$ as $D_{|\boldsymbol{y}_{1:t}} = \cup_{y_{t+1} \in D_t} D_{|\boldsymbol{y}_{1:t+1}}$, whereas Lorberbom et al. (2020) expand only a single child node and partition the domain as $D_{|\boldsymbol{y}_{1:t}} = D_{|\boldsymbol{y}_{1:t+1}} \cup (D_{|\boldsymbol{y}_{1:t}} \setminus D_{|\boldsymbol{y}_{1:t+1}})$. When a single child node $\boldsymbol{y}_{1:t+1}$ is created, this corresponds to the maximum and thus it inherits the perturbed log-probability from the parent. For the parent, a new perturbed log-probability is sampled, truncated at the previous maximum, which then becomes the maximum of perturbed log-probabilities for the remaining domain $D_{|\boldsymbol{y}_{1:t}} \setminus D_{|\boldsymbol{y}_{1:t+1}}$ corresponding to the child nodes that have not yet been instantiated.

5.4.2 (Stochastic) beam search

Beam search is a widely used method for approximate inference in various domains such as machine translation (Sutskever et al., 2014; Bahdanau et al., 2015) and image captioning (Vinyals et al., 2015b). In machine learning, beam search is typically a test-time procedure, but there are works that include beam search in the training loop (Daumé et al., 2009; Wiseman and Rush, 2016; Edunov et al., 2018b; Negrinho et al., 2018; Edunov et al., 2018a). Beam search suffers from limited diversity of the results, and variants have been developed that encourage diversity (Li et al., 2016; Shao et al., 2017; Vijayakumar et al., 2018).

We argue that adding stochasticity is also a principled way to increase diversity in a beam search: this motivated the development of stochastic beam search (Kool et al., 2019c). While stochastic beam search is equivalent to ancestral Gumbel-top-k sampling (only) for $m = k$, the result (a sample without replacement) is equivalent for *any* $m \leq k$. As such, we also consider ancestral Gumbel-top-k sampling as a principled alternative to a heuristically randomized/diversified beam search, even though it is *not* (in general) a beam search.

We analyze the result of ancestral Gumbel-top-k sampling by comparing stochastic beam search to a naïve alternative implementation of a randomized beam search. In particular, imagine that we use an ordinary beam search, but replace the deterministic top-k operation by sampling without replacement in each step, e.g. using

Gumbel-top-k sampling, but *without* top-down sampling conditionally. In this naïve approach, a low-probability partial configuration will only be extended to completion, if it gets to be re-sampled, independently, again with low probability, at each step during the beam search. The result is a much lower probability to sample this configuration than assigned by the model.

Intuitively, we should somehow *commit* to a sampling 'decision' made at step t. However, a hard commitment to generate exactly one completion for each of the k partial configurations at step t would prevent generating any two completions from the same partial configuration. By using top-down sampling, stochastic beam search propagates the Gumbel perturbation of a partial configuration to its extensions, which can be seen as a *soft* commitment to that partial configuration. This means that it gets extended as long as its total perturbed log-probability is among the top k, but 'falls off' the beam if, despite the consistent perturbation, another partial configuration becomes more promising. By this procedure, the result is a sample without replacement, which is true to the model, suggesting that stochastic beam search, or equivalently, ancestral Gumbel-top-k sampling, is a principled alternative to a heuristically randomized beam search.

5.4.3 Threshold sampling

While Gumbel-top-k sampling uses a fixed sample size of k, a related algorithm is *threshold sampling* (Duffield et al., 2005), which instead sets a fixed threshold and returns a variable sized sample. In analogy with Gumbel-top-k sampling, threshold sampling returns all configurations in the domain for which the perturbed log-probability exceeds a fixed threshold, instead of the k largest. Threshold sampling is a special case of *Poisson sampling* as each configuration is included in the sample with probability *independent* of the other configurations. This allows the sample to be used for statistical estimation using the Horvitz-Thompson estimator (Horvitz and Thompson, 1952). To overcome the limitation of a variable sample size, *priority sampling* (Duffield et al., 2007) uses the $(k+1)$-th largest value as empirical threshold to obtain a fixed sample size k. This is equivalent to Gumbel-top-k sampling and we experiment with the resulting estimator in Section 5.5.3. An alternative method to control the sample size is to use an adaptive threshold (Ting, 2017).

5.4.4 Rejection sampling

As an alternative to our algorithm, we can draw samples *without* replacement by rejecting duplicates from samples drawn *with* replacement. However, if the domain is large and the entropy low (e.g. with a translation model where there are only a few valid translations), then rejection sampling requires many samples and consequently many (expensive) model evaluations. Also, we have to estimate how many samples to draw (in parallel) or draw samples sequentially. Our algorithm allows us to set $m = k$, which guarantees generating k unique samples with exactly kT

model evaluations in *a single pass* of T steps, which is equal to the computational requirement for taking k samples with replacement. When allowing our algorithm to generate samples sequentially, setting $m < k$, it can generate k samples with fewer model evaluations than standard sampling with replacement.

5.4.5 Sequential ancestral sampling without replacement

As an alternative to ancestral Gumbel-top-k sampling, we can also derive an algorithm based on 'standard' sequential sampling without replacement. This means sampling the first configuration $\boldsymbol{y}_1^*$, then renormalizing the probability distribution to the domain $D \setminus \{\boldsymbol{y}_1^*\}$ to sample $\boldsymbol{y}_2^*$, et cetera. The sequential method we propose is similar to Wong and Easton (1980), but instead of an arbitrary binary tree, we use the tree structure (Figure 14) induced by ancestral sampling from the graphical model. This method was concurrently proposed by Shi et al. (2020).

We represent configurations $\boldsymbol{y} \in D$ as 'buckets' on a horizontal axis, with size equal to their probability $p(\boldsymbol{y})$. Sampling from $p(\boldsymbol{y})$ is equivalent to uniformly selecting a point on this axis and returning the configuration of the corresponding bucket. See an example in Figure 18. Sampling without replacement means that to obtain the second sample, we should remove the bucket corresponding to the first sample, as indicated by the dark shading in Figure 18, and sample uniformly a position on the horizontal axis that is not shaded. When using ancestral sampling, we can determine the bucket by first determining the larger bucket, corresponding to the assignment of the first variable y_1, then sequentially narrowing it down to the final bucket by sampling $y_2, ..., y_T$. Note that, although this consists of multiple sampling steps, this can be done using a single random number, as illustrated by the dotted line in Figure 18, such that the result will be equivalent to the result without ancestral sampling with the same random number.

When sampling without replacement, we desire unique *complete configurations*, but we may have duplicate *partial configurations*. This means that if we desire to use sequential ancestral sampling to obtain a second sample without replacement, we cannot remove any partial configurations from the domain, but we need to remove the *probability mass* already sampled from the partial configurations, as illustrated in Figure 18. We can keep track of the probability mass already sampled for each partial configuration by building a prefix tree (or *trie*) representing the samples, removing the probability mass from each partial configuration by backtracking the path that was used to generate each sample. Each next sample can then be generated by sampling a path down this tree using the adjusted probabilities to sample without replacement, until we arrive at a partial configuration that has not been sampled before, which means that the remaining variables in the configuration can be sampled straightforwardly and the tree is grown to include this path.

Similar to ancestral Gumbel-top-k sampling, sequential ancestral sampling without replacement has the benefits that it can reuse model evaluations for partial configurations and generate samples one at the time. As downside, this sequential

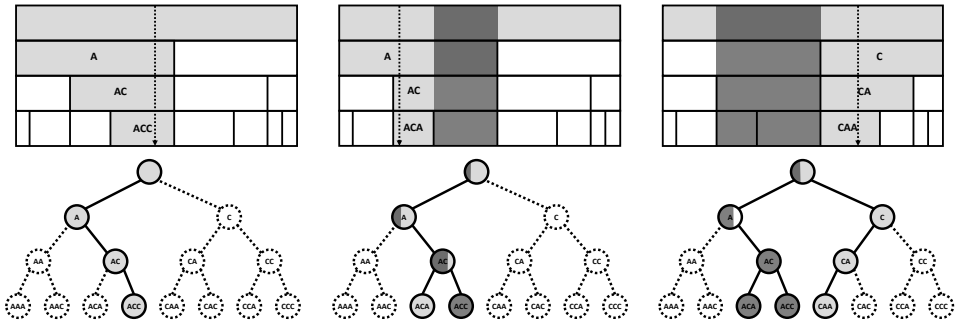


Figure 18: Sampling from a discrete domain visualized as selecting a point uniformly on a horizontal axis. The bucket (with size equal to sampling probability) which contains the random point is the sample. Ancestral sampling can be seen as sequentially narrowing down the 'bucket' that contains the sampled point, indicated by light shading. Sampling without replacement requires removing probability mass corresponding to the buckets that have already been sampled (indicated by dark shaded areas) and selecting the random point uniformly from the remaining area. When using sequential ancestral sampling, we can use a tree to keep track of the probability mass that should be removed for partial configurations that have been sampled before.

method requires more iterations as it starts from the root for each sample, whereas ancestral Gumbel-top-k sampling directly jumps to a partial configuration from the priority queue. Also, this sequential method requires careful administration of a complete tree structure whereas ancestral Gumbel-top-k sampling only requires to maintain a queue of partial configurations (without any tree structure). Lastly, sequential ancestral sampling without replacement cannot be parallelized efficiently.

5.4.6 Weighted reservoir sampling with exponential jumps

Weighted reservoir sampling (Efraimidis and Spirakis, 2006) is an algorithm for sampling without replacement from a stream of items given weights (unnormalized probabilities) to sample each item. Mathematically, it is equivalent to Gumbel-top-k sampling (see Section 5.2.4), but it is executed in a streaming manner, i.e. perturbing (unnormalized) log-probabilities as they arrive while keeping track of the k largest perturbations so far.

In the streaming implementation, each next item replaces an item in a priority queue if its perturbed log-probability exceeds the k-th largest so far. Efraimidis and Spirakis (2006) note that the total weight that is 'skipped' before a next item enters the queue follows an exponential distribution, which can also be sampled directly, such that one can directly make an 'exponential jump' to the next item to insert the queue, without sampling the perturbations for each item individually. This can be seen as a form of top-down sampling.

Using weighted reservoir sampling, or equivalently, 'streaming' Gumbel-top-k sampling, we can derive yet another algorithm for sampling without replacement from a Bayesian network, by iterating over the complete domain (leaf nodes in the tree diagram in Figure 14) using a tree traversal algorithm, and keeping track of the k-largest perturbed log-probabilities so far. While this needs to enumerate the

complete domain, we can use the 'exponential jumps' to directly jump to the next sequence in the domain. In doing so, complete subtrees can be skipped directly as the total weight (probability mass) in the subtree is given by the probability of the partial configuration corresponding to the root of the subtree. This means it can be executed without instantiating the complete tree. Additionally, traversing the tree by expanding the highest probability children first will limit the number of additions to the priority queue as more likely elements are inserted earlier.

While this algorithm is closely connected to our method, it is *not* equivalent. In particular, using streaming Gumbel-top-k sampling with exponential jumps, the queue will be initially filled with k *complete* configurations, which get replaced later by other complete configurations with higher perturbed log-probabilities. The result can only be returned after the complete domain has been traversed. By contrast, ancestral Gumbel-top-k sampling maintains *partial* configurations in the queue, which can be used to bound parts of the domain, and additionally returns samples without replacement as they are found, in order.

5.5 EXPERIMENTS

This section presents the experiments and results.

5.5.1 Different methods for sampling without replacement

In our first experiment, we analyze three methods for sampling without replacement:

- *Ancestral Gumbel-top-k sampling* (Section 5.3), where we experiment with different values of m to control the paralellizability of the algorithm.
- *Rejection sampling* (Section 5.4.4) which generates samples *with replacement* (using standard ancestral sampling) sequentially and rejects duplicates. We also implement a 'parallel' version of this, that generates m samples *with replacement* in parallel, then rejects the duplicates and repeats this procedure until k unique samples are found.
- *Sequential ancestral sampling* without replacement (Section 5.4.5). This algorithm is inherently sequential, but we also implement a simple parallelizable version similar to rejection sampling. This version generates m samples (with replacement) in parallel, removes duplicates and then constructs the tree in Figure 18 to remove the corresponding probability mass from the distribution. Then it uses this tree to generate (again in parallel) m new samples, which are distinct from the first m samples but may contain duplicates, which are again removed, after which the tree is updated. This is repeated until in total k unique samples have been generated.

DATA GENERATION We generate a random Bayesian network of 10 Bernoulli variables y_i. We generate the variables in topological order, and variable i has a dependency on variable $j < i$ (so $j \in \text{pa}(i)$) with probability $c \in [0, 1]$, where c is the *connectivity* factor of the graph. For $c = 0$, the model is fully independent (Figure 11a), while for $0 < c < 1$ the result will (most likely) be a sparse Bayesian network (Figure 11c) and for $c = 1$ the result is a 'fully connected' sequence model (Figure 11d). The Bernoulli distribution for $y_i \in \{0, 1\}$ is a mixture between an independent prior and a distribution depending on the parents, which is given by

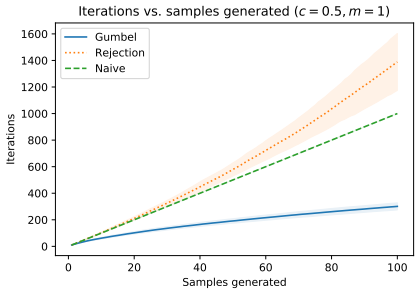
$$P(y_i = 1 | \boldsymbol{y}_{\text{pa}(i)}) = \alpha_i p_i^{\boldsymbol{y}_{\text{pa}(i)}} + (1 - \alpha_i) p_i.$$

Here $\alpha_i \sim \text{Uniform}(0, 1)$ is a parameter that determines how much the variable y_i is influenced by its parents $\text{pa}(i)$, $p_i \sim \text{Uniform}(0, 1)$ is the 'independent' or 'prior' probability that $y_i = 1$, and $p_i^{\boldsymbol{y}_{\text{pa}(i)}} \sim \text{Uniform}(0, 1) \quad \forall \boldsymbol{y}_{\text{pa}(i)} \in D_{\text{pa}(i)}$ determines the influence of the parents on the probability that $y_i = 1$, given as a table for each of the $2^{|\text{pa}(v)|}$ possible values in $D_{\text{pa}(i)}$.

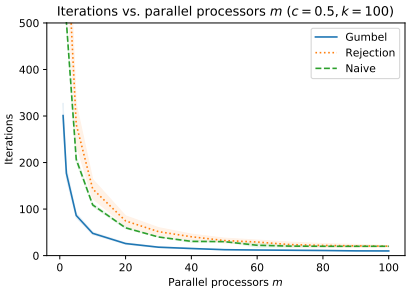
NUMBER OF ITERATIONS First we set $m = 1$, making all algorithms fully sequential. For our algorithm, we measure the number of iterations (removing $m = 1$ element from the queue and adding its expansions), which is equal to the number of model evaluations. For rejection sampling and the sequential ancestral sampling algorithm, we count as iteration the sampling of one variable, such that the number of iterations is also equal to the number of model evaluations and generating a single sample $\boldsymbol{y}$ with T variables takes T iterations/model evaluations. In Figure 19a we compare the number of iterations for generating different numbers of samples using the different methods. We clearly observe how our method requires the fewest iterations/model evaluations to generate the same number of samples.

PARALLELIZING THE ALGORITHMS By choosing the number of parallel processors m we can trade off the total time (iterations), also known as the *depth* or *span*, against the total *cost* of running the algorithm, which is m times the depth or span. Figure 19b shows for different values of m how many iterations it takes to generate $k = 100$ samples, i.e. it summarizes the result for the previous experiment run for different values of m. It is clear how increasing the parallelization quickly decreases the number of iterations required although the effect diminishes as m increases.

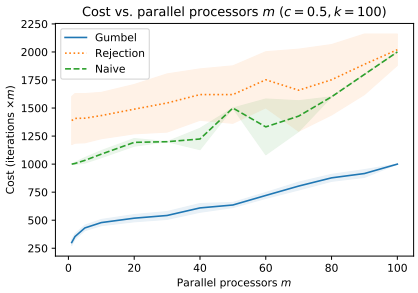
To visualize the overhead from the parallelization, Figure 19c visualizes the total cost, which is the number of iterations (Figure 19b) multiplied by m. For ancestral Gumbel-top-k sampling, the increased costs for larger m has two different sources as explained in Section 5.3.8: redundant model evaluations of partial configurations which would have been pushed off the queue in the sequential setting, and 'idle' parallel resources if the queue size is smaller than m. For rejection sampling, the parallel version is suboptimal because too many samples may be generated since they are generated in batches of size m. For the sequential ancestral sampling algorithm with batches ($m > 1$), there may still be duplicates in a single batch, and more than k samples may be generated in total.



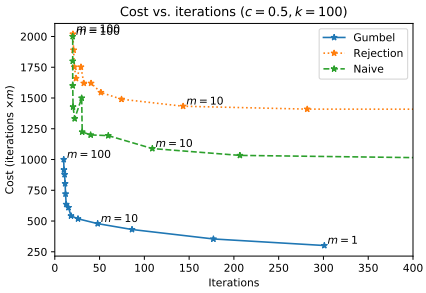
(a) Number of iterations / model evaluations vs. number of samples generated without replacement, without parallelism ($m = 1$).



(b) Number of iterations to generate $k = 100$ samples without replacement for different numbers of parallel processors m.



(c) Total *cost* (iterations $\times m$) as function of number of samples generated without replacement, without parallelism ($m = 1$).



(d) Trade-off between total cost and number of iterations, as controlled by the number of parallel processors m.

Figure 19: Results of sampling from 100 randomly generated models (Bayesian networks) with 10 Bernoulli variables each and a connectivity of $c = 0.5$. For each model, we generated $k = 100$ samples without replacement, using different methods and different numbers of parallel processors m. Rejection sampling and sequential ancestral sampling show a peak at $m = 50$ since they typically generate $3 \times 50 = 150$ samples (cost 1500) to obtain 100 unique samples, whereas $m = 40$ and $m = 60$ can suffice with generating 3×40 or $2 \times 60 = 120$ samples. Results are presented as mean and standard deviation over the 100 different models. Results with $c = 0$ or $c = 1$ (not shown) are similar in terms of iterations and costs.

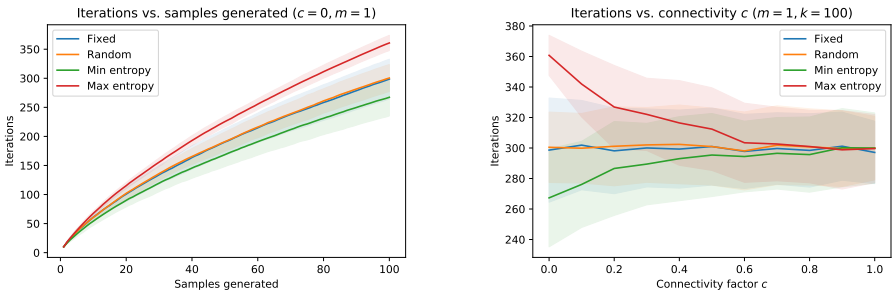
Ultimately, m determines the trade-off between the number of iterations and the total cost of the algorithm, which is visualized in Figure 19d. Our algorithm with $m = k = 100$ (i.e. stochastic beam search) uses 10 iterations (since there are 10 variables) and has a cost of $10 \times 100 = 1000$, which is the *minimum* for sequential ancestral sampling and rejection sampling. On the other extreme, the sequential algorithm with $m = 1$ uses around 300 iterations with a cost of 300. The difference is large because of the small setting, where, with $m = 100$, many processors are idle in 6 of the 10 iterations since $2^6 = 64 < 100$. We expect the difference to be smaller in real applications, but this experiment clearly shows that there is a trade-off, and suggests that limited parallelization (in this case $m = 10$) rapidly decreases the number of iterations without increasing total cost too much.

VARIABLE SELECTION STRATEGY As explained in Section 5.3.9, we can select *any* variable from $\{v \in V \setminus \mathcal{S} : \text{pa}(v) \subseteq \mathcal{S}\}$ to sample in each iteration. We experiment with the following variable selection strategies:

- *Fixed* uses the order in which the variables are generated. Since we generated the Bayesian network in topological order, this is valid.
- *Random* chooses the variable from $\{v \in V \setminus \mathcal{S} : \text{pa}(v) \subseteq \mathcal{S}\}$ to sample next uniformly at random.
- *Minimum entropy* computes the (conditional) entropy for all variables $\{v \in V \setminus \mathcal{S} : \text{pa}(v) \subseteq \mathcal{S}\}$ as $-\hat{p}_v \log(\hat{p}_v) - (1 - \hat{p}_v) \log(1 - \hat{p}_v)$, where $\hat{p}_v = P(y_v = 1 | \boldsymbol{y}_{\text{pa}(v)})$ and selects the variable v with minimum entropy.
- *Maximum entropy* selects the variable v with maximum entropy.

We note that selecting v based on entropy requires the model to be evaluated, which may be undesirable in some cases. However, theoretically these model evaluations can be cached/reused as most evaluations will be required eventually to sample the remaining variables. In the extreme case of $c = 0$, i.e. the fully independent model in Figure 11a, we can simply precompute all model evaluations (which is a good idea anyway), and using the entropy variable selection strategy reduces to sorting the variables by their entropy before starting the Gumbel-top-k sampling algorithm.

In Figure 20a we clearly see that selecting variables with minimum entropy first is the best strategy (it requires fewest iterations), whereas selecting based on maximum entropy performs worst, with a random strategy or using the fixed generation order (which can also be considered random) in between. As explained in Section 5.3.9, this is because selecting variables with a low entropy allows maximum reuse of partial configurations. In Figure 20b we show how the difference between different variable selection strategies decreases as there are more dependencies between variables, where for a 'fully connected' sequence model ($c = 1$, Figure 11d) there is no freedom in variable selection and all strategies perform the same.



(a) Number of iterations as function of number of samples generated without replacement, for a fully independent model ($c = 0$).

(b) Number of iterations to generate $k = 100$ samples without replacement for different levels of connectivity c of the model.

Figure 20: Results of generating $k = 100$ samples using ancestral Gumbel-top-k sampling with $m = 1$ and different variable selection strategies, for models with different levels of connectivity c. Results are presented as mean and standard deviation over 100 randomly generated models.

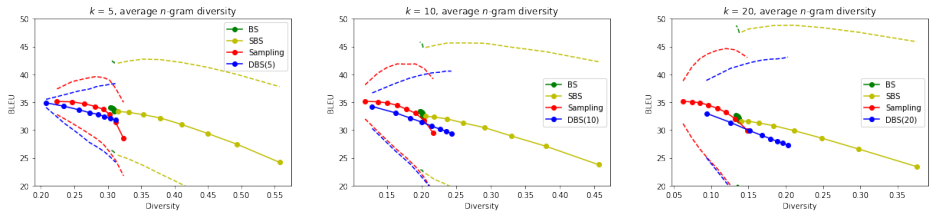
5.5.2 Diverse beam search

This experiment was originally presented in Kool et al. (2019c). The results were obtained using stochastic beam search (i.e. $m = k$), but are valid for any implementation of ancestral Gumbel-top-k sampling with parallelism factor $m < k$. In this experiment we compare stochastic beam search as a principled (stochastic) alternative to *diverse beam search* (Vijayakumar et al., 2018) in the context of neural machine translation to obtain a diverse set of translations for a single source sentence $\boldsymbol{x}$. Following the setup by Vijayakumar et al. (2018) we report both diversity as measured by the fraction of unique n-grams in the k translations, as well as mean and maximum BLEU score (Papineni et al., 2002) as an indication of the quality of the sample. The maximum BLEU score corresponds to 'oracle performance' reported by Vijayakumar et al. (2018), but we report the mean as well since a single good translation and $k - 1$ completely random sentences scores high on both maximum BLEU score and diversity, while being undesirable. A good method should increase diversity without sacrificing mean BLEU score.

We compare four different sentence generations methods: *beam search* (BS), *sampling* (with replacement), *stochastic beam search* (SBS) (sampling without replacement) and *diverse beam search* with G groups (DBS(G)) (Vijayakumar et al., 2018). For sampling and stochastic beam search, we control the diversity of the sentences using the *softmax temperature* τ (see equation 24) used to compute the model probabilities. We use $\tau = 0.1, 0.2, ..., 0.8$, where a higher τ results in higher diversity. Heuristically, we also vary τ for computing the scores with (deterministic) beam search. The diversity of diverse beam search is controlled by the *diversity strengths* parameter, which we vary between $0.1, 0.2, ..., 0.8$. We set the number of groups G equal to the sample size k, which Vijayakumar et al. (2018) reported as the best choice.

We modify the beam search in `fairseq` (Ott et al., 2019) to implement stochastic beam search[3], and use the `fairseq` implementations for beam search, sampling and diverse beam search. For theoretical correctness of the stochastic beam search, we

3 Our code is available at https://github.com/wouterkool/stochastic-beam-search.



(a) $k = 5$ **(b)** $k = 10$ **(c)** $k = 20$

Figure 21: Minimum, mean and maximum BLEU score vs. diversity for different sample sizes k. Points indicate different temperatures/diversity strengths, from 0.1 (low diversity, left in graph) to 0.8 (high diversity, right in graph).

disable length-normalization (Wu et al., 2016) and early stopping (and therefore also do not use these parameters for the other methods). We use the pretrained model from Gehring et al. (2017) and use the `wmt14.v2.en-fr.newstest2014` test set[4] consisting of 3003 sentences. We run the four methods with sample sizes $k = 5, 10, 20$ and plot the minimum, mean and maximum BLEU score among the k translations (averaged over the test set) against the average $d = \frac{1}{4}\sum_{n=1}^{4} d_n$ of 1, 2, 3 and 4-gram diversity, where n-gram diversity is defined as

$$d_n = \frac{\text{\# of unique } n\text{-grams in } k \text{ translations}}{\text{total \# of } n\text{-grams in } k \text{ translations}}.$$

In Figure 21, we represent the results as curves, indicating the trade-off between diversity and BLEU score. The points indicate datapoints and the dashed lines indicate the (averaged) minimum and maximum BLEU score. For the same diversity, stochastic beam search achieves higher mean/maximum BLEU score. Looking at a certain BLEU score, we observe that stochastic beam search achieves the same BLEU score as diverse beam search with a significantly larger diversity. For low temperatures (< 0.5), the maximum BLEU score of stochastic beam search is comparable to the deterministic beam search, so the increased diversity does not sacrifice the best element in the sample. Note that sampling achieves higher mean BLEU score at the cost of diversity, which may be because good translations are sampled repeatedly. However, the maximum BLEU score of both sampling and diverse beam search is lower than with beam search and stochastic beam search.

5.5.3 BLEU score estimation

In our second experiment, also presented originally in Kool et al. (2019c), we use sampling without replacement to evaluate the expected *sentence level* BLEU score for a translation $\boldsymbol{y}$ given a source sentence $\boldsymbol{x}$. Although we are often interested in *corpus level* BLEU score, estimation of sentence level BLEU score is useful, for example when training using minibatches to directly optimize BLEU score (Ranzato et al.,

4 Available at https://s3.amazonaws.com/fairseq-py/data/wmt14.v2.en-fr.newstest2014.tar.bz2.

2016). We leave dependence of the BLEU score on the source sentence $\boldsymbol{x}$ implicit, and write $f(\boldsymbol{y}) = \text{BLEU}(\boldsymbol{y}, \boldsymbol{x})$. We want to estimate the following expectation:

$$\mathbb{E}_{\boldsymbol{y}\sim p_\theta(\boldsymbol{y}|\boldsymbol{x})}\left[f(\boldsymbol{y})\right] = \sum_{\boldsymbol{y}\in D} p_\theta(\boldsymbol{y}|\boldsymbol{x}) f(\boldsymbol{y}). \tag{41}$$

Under a Monte Carlo (MC) sampling *with replacement* scheme with size k, we write S as the set[5] of sampled sequences and estimate equation 41 using

$$\mathbb{E}_{\boldsymbol{y}\sim p_\theta(\boldsymbol{y}|\boldsymbol{x})}\left[f(\boldsymbol{y})\right] \approx \frac{1}{k}\sum_{\boldsymbol{y}\in S} f(\boldsymbol{y}). \tag{42}$$

If the distribution $p_\theta(\boldsymbol{y}|\boldsymbol{x})$ has low entropy (for example if there are only few valid translations), then MC estimation may be inefficient since repeated samples are uninformative. We can use sampling without replacement as an improvement, but we need to use importance weights to correct for the changed sampling probabilities. Using Gumbel-top-k sampling, we can implement an estimator equivalent to the estimator described by Vieira (2017), which can be seen as a Horvitz-Thompson estimator (Horvitz and Thompson, 1952) used with priority sampling (Duffield et al., 2007):

$$\mathbb{E}_{\boldsymbol{y}\sim p_\theta(\boldsymbol{y}|\boldsymbol{x})}\left[f(\boldsymbol{y})\right] \approx \sum_{\boldsymbol{y}\in S} \frac{p_\theta(\boldsymbol{y}|\boldsymbol{x})}{q_{\theta,\kappa}(\boldsymbol{y}|\boldsymbol{x})} f(\boldsymbol{y}). \tag{43}$$

Using this estimator, we 'sacrifice' the k-th sample to obtain the empirical threshold κ (which is the k-th largest perturbed log-probability, see Section 5.3.7), and we define S as the set of the $k-1$ sequences corresponding to the $k-1$ largest perturbed log-probabilities. It holds that $\boldsymbol{y} \in S$ if $G_{\phi_{\boldsymbol{y}}} > \kappa$, which highlights the relation to threshold sampling (see Section 5.4.3). We define

$$q_{\theta,a}(\boldsymbol{y}|\boldsymbol{x}) = P(G_{\phi_{\boldsymbol{y}}} > a) = 1 - \exp(-\exp(\phi_{\boldsymbol{y}} - a)). \tag{44}$$

If we would assume a fixed threshold a and *variably* sized sample $S = \{\boldsymbol{y} \in D : G_{\phi_{\boldsymbol{y}}} > a\}$, then $q_{\theta,a}(\boldsymbol{y}|\boldsymbol{x}) = P(\boldsymbol{y} \in S)$ and $\frac{p_\theta(\boldsymbol{y}|\boldsymbol{x})}{q_{\theta,a}(\boldsymbol{y}|\boldsymbol{x})}$ is a standard importance weight. Surprisingly, using a *fixed* sample size k (and empirical threshold κ) also yields in an unbiased estimator, and we include a proof adapted from Duffield et al. (2007) and Vieira (2017) in Appendix C.2.

Empirically, the estimator in equation 43 has high variance, and in practice we find it is preferred to normalize the importance weights by $W(S) = \sum_{\boldsymbol{y}\in S} \frac{p_\theta(\boldsymbol{y}|\boldsymbol{x})}{q_{\theta,\kappa}(\boldsymbol{y}|\boldsymbol{x})}$ (Hesterberg, 1988):

$$\mathbb{E}_{\boldsymbol{y}\sim p_\theta(\boldsymbol{y}|\boldsymbol{x})}\left[f(\boldsymbol{y})\right] \approx \frac{1}{W(S)}\sum_{\boldsymbol{y}\in S} \frac{p_\theta(\boldsymbol{y}|\boldsymbol{x})}{q_{\theta,\kappa}(\boldsymbol{y}|\boldsymbol{x})} f(\boldsymbol{y}). \tag{45}$$

The estimator in equation 45 is biased but consistent: in the limit $k = |D|$ we sample the entire domain, so we have empirical threshold $\kappa = -\infty$ and $q_{\theta,\kappa}(\boldsymbol{y}|\boldsymbol{x}) = 1$ and $W(S) = 1$, such that the estimators in equation 45 and equation 41 are equal.

5 Formally, when sampling with replacement, S is a *multiset*.

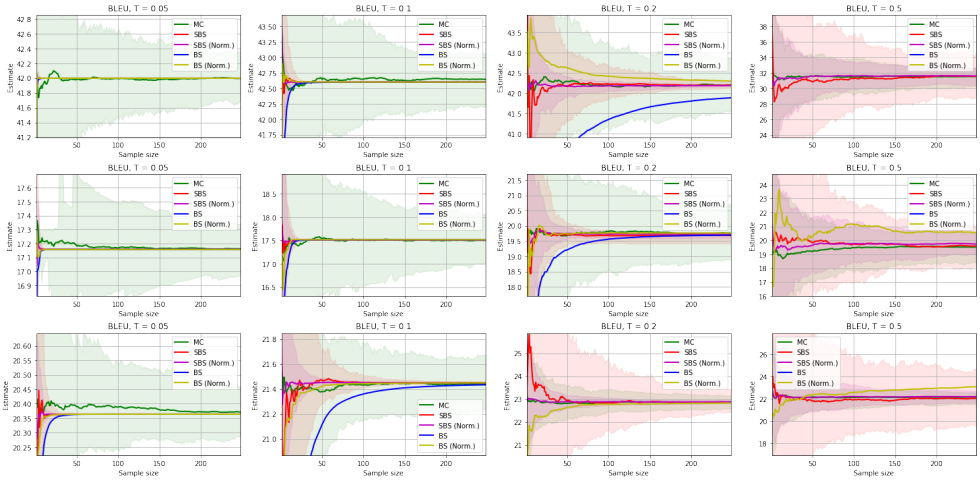


Figure 22: BLEU score estimates for three sentences sampled/decoded by different estimators for different temperatures.

We have to take care computing the importance weights as, depending on the entropy, the terms in the quotient $\frac{p_\theta(\boldsymbol{y}|\boldsymbol{x})}{q_{\theta,\kappa}(\boldsymbol{y}|\boldsymbol{x})}$ can become very small, and the computation of equation 44 can suffer from catastrophic cancellation (see Appendix C.3).

Because the model is not trained to use its own predictions as input, at test time errors can accumulate. As a result, when sampling with the default temperature $\tau = 1$, the expected BLEU score is very low (below 10). To improve quality of generated sentences we use lower temperatures and experiment with $\tau = 0.05, 0.1, 0.2, 0.5$. We then use different methods to estimate the BLEU score:

- *Monte Carlo* (MC), using equation 42.
- *Stochastic beam search* (SBS), where we compute estimates using the estimator in equation 43 and the normalized variant in equation 45.
- *Beam search* (BS), where we compute a deterministic beam S (the temperature τ affects the scoring) and compute $\sum_{\boldsymbol{y}\in S} p_\theta(\boldsymbol{y}|\boldsymbol{x}) f(\boldsymbol{y})$. This is not a statistical estimator, but a lower bound to the target (equation 41), which serves as a validation of the implementation and gives insight on how many sequences we need at least to capture most of the mass of the expectation in equation 41. We also compute the normalized version $\frac{\sum_{\boldsymbol{y}\in S} p_\theta(\boldsymbol{y}|\boldsymbol{x}) f(\boldsymbol{y})}{\sum_{\boldsymbol{y}\in S} p_\theta(\boldsymbol{y}|\boldsymbol{x})}$, which can heuristically be considered as a 'determinstic estimate'.

In Figure 22 we show the results of computing each estimate 100 times (BS only once as it is deterministic) for three different sentences[6] for temperatures $\tau = 0.05, 0.1, 0.2, 0.5$ and sample sizes $k = 1$ to 250. We report the empirical mean and 2.5-th and 97.5-th percentile. The normalized SBS estimator indeed achieves significantly lower variance than the unnormalized version and for $\tau < 0.5$, it significantly reduces variance compared to MC, without adding observable bias. For $\tau = 0.5$ the results are similar, but we are less interested in this case as the overall BLEU score is lower than for $\tau = 0.2$.

6 These are sentences 1500, 2000 and 2500 from the WMT14 data set.

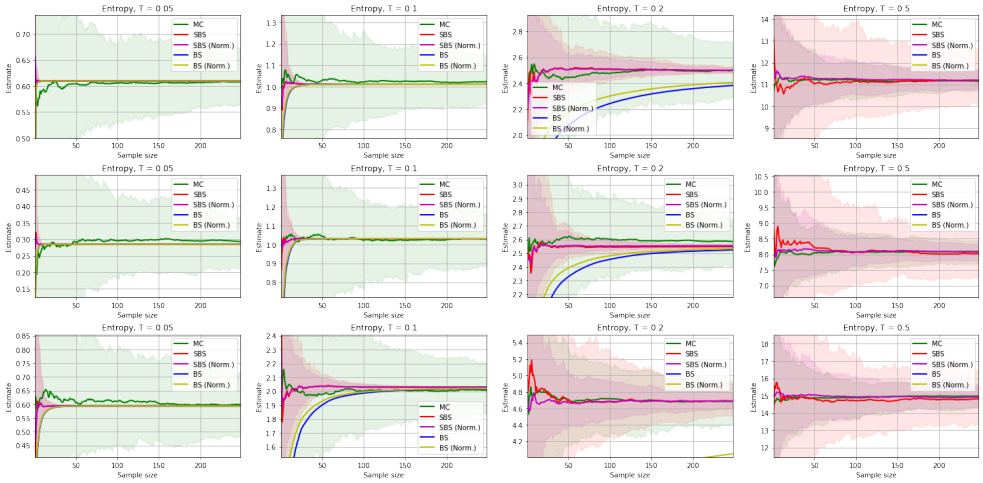


Figure 23: Entropy estimates for three sentences sampled/decoded by different estimators for different temperatures.

5.5.4 Conditional entropy estimation

Additionally to estimating the BLEU score we use $f(\boldsymbol{y}) = -\log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$ such that equation 41 becomes the model entropy (conditioned on the source sentence $\boldsymbol{x}$)

$$\mathbb{E}_{\boldsymbol{y}\sim p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})}\left[-\log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})\right].$$

Entropy estimation is useful in optimization settings where we want to include an entropy loss to ensure diversity. It is a different problem than BLEU score estimation as high BLEU score (for a good model) correlates positively with model probability, while for entropy rare $\boldsymbol{y}$ contribute the largest terms $-\log p_{\boldsymbol{\theta}}(\boldsymbol{y}|\boldsymbol{x})$. We use the same experimental setup as for the BLEU score and present the results in Figure 23. The results are similar: the normalized SBS estimate has significantly lower variance than MC for $\tau < 0.5$ while for $\tau = 0.5$, results are similar. This shows that stochastic beam search can be used to construct practical statistical estimators.

5.6 POSSIBLE EXTENSIONS OF ANCESTRAL GUMBEL-TOP-k SAMPLING

Our algorithm can be extended in various ways, to give efficient implementations of two existing algorithms or to perform efficient sampling *with* replacement.

5.6.1 Memory augmented policy optimization

Our algorithm can be extended to give an efficient implementation of memory augmented policy optimization (MAPO) (Liang et al., 2018), an extension of REINFORCE (Williams, 1992) that optimizes a policy in reinforcement learning using a

replay buffer of good experiences. MAPO computes an exact gradient for the b experiences (configurations) in the buffer, which requires b model evaluations, and uses a sample from the model outside of the buffer, obtained using (potentially inefficient) rejection sampling. We can adapt ancestral Gumbel-top-k sampling to implement MAPO efficiently, by modifying the priority queue to use a hierarchical criterion, that will put in the front partial configurations which correspond to partial configurations in the buffer,[7] and sort the remaining configurations based on the perturbed log-probability. When using $k = b + 1$, the result will be the b configurations in the buffer (with their model evaluations), and a single sample outside of the buffer.

5.6.2 Rao-Blackwellized stochastic gradients

While MAPO is an estimator that computes an exact gradient for a number of 'good' configurations, Liu et al. (2019) proposed a similar estimator that computes an exact gradient for high probability configurations (instead of 'good' ones), which may have a greater contribution overall. While they consider 1-dimensional categorical distributions only, we note that the estimator can also be used in multi-dimensional settings, where high probability categories can be found by an approximate algorithm such as (deterministic) beam search, but in this case it is challenging to obtain a sample from the remaining domain.

Similar to MAPO, we can make a modification of our algorithm, to yield *both* the high probability configurations, and a sample from the remaining domain. In particular, we can modify stochastic beam search (our algorithm with $m = k$) to maintain in the queue the $k - 1$ partial configurations with highest log-probability (*without* Gumbel perturbation) and one partial configuration which has the highest *perturbed* log-probability among the remaining configurations. This means that we have a 'deterministic' beam of size $k - 1$ as well as a single sample (partial configuration) outside of the beam.[8]

In general, it may be preferred to have fewer exact and more sampled configurations; see the relevant discussions in Fearnhead and Clifford (2003) and Liu et al. (2019). Using our algorithm, we could maintain $k - \ell$ configurations by their log-probability, and ℓ configurations by the perturbed log-probability. The resulting ℓ samples without replacement can be converted to (at least) ℓ samples with replacement using resampling (see below), or one can use an estimator based on sampling without replacement (Kool et al., 2019b; Kool et al., 2020b).

7 Existence of a partial configuration/trajectory in the buffer can be efficiently checked by representing the replay buffer as a prefix tree (*trie*), similar to Figure 18.

8 The result may be slightly different than deterministic beam search with size $k - 1$, since extensions of the k-th configuration (the sample) may have high log-probability and push other configurations off the beam. If desired, this can be heuristically prevented.

5.6.3 Sampling with replacement with sampling without replacement

Sampling without replacement can be used to sample *with replacement* by using a *resampling* algorithm. The basic idea is that using ancestral Gumbel-top-k sampling (Algorithm 2) 'lazily', we can obtain the first sample $\boldsymbol{y}_1^*$, for which sampling with/without replacement is the same. Then for the second sample, we can take $\boldsymbol{y}_1^*$ with probability $p(\boldsymbol{y}_1^*)$, or sample from $D \setminus \{\boldsymbol{y}_1^*\}$ with probability $1 - p(\boldsymbol{y}_1^*)$, which can be done by continuing incremental sampling without replacement. At any point, if $\boldsymbol{y}_1^*, ..., \boldsymbol{y}_n^*$ are the samples *without replacement* so far, we can get another sample *with replacement* by choosing $\boldsymbol{y}_1^*$ with probability $p(\boldsymbol{y}_1^*)$, $\boldsymbol{y}_2^*$ with probability $p(\boldsymbol{y}_2^*)$, et cetera or find the next sample $\boldsymbol{y}_{n+1}^*$ (without replacement) with probability $1 - \sum_{i=1}^{n} p(\boldsymbol{y}_i^*)$. Repeating this algorithm to obtain a desired number of samples has a lower computational cost (as measured by model evaluations) than standard sampling with replacement, as resampling is cheap (it does not require additional model evaluations) and the inner ancestral Gumbel-top-k sampling algorithm uses model evaluations efficiently.

5.7 DISCUSSION

We introduced *ancestral Gumbel-top-k sampling*, an algorithm that can efficiently draw samples without replacement from a probability distribution represented as a Bayesian network. It has a parameter to control the amount of parallelism of the algorithm, trading off the number of required iterations versus the total cost of running the algorithm, such that it can make efficient use of modern hardware. We discussed possible extensions of the algorithm, enabling implementations of the gradient estimators by Liang et al. (2018) and Liu et al. (2019) and a resampling algorithm to efficiently obtain samples *with* replacement.

We have discussed related algorithms and empirically shown the benefits of using ancestral Gumbel-top-k sampling: it enables to generate samples without replacement using a significantly lower computational cost than alternatives. We have analyzed the influence of the number of parallel processors experimentally, suggesting that limited parallelism quickly decreases the number of required iterations without increasing total cost too much. Additionally, we have shown how selecting the order of sampling based on entropy can reduce the cost of the algorithm, which is especially useful for sampling from fully independent models where the 'optimal' order of sampling can be determined upfront.

In the context of sequence models, Gumbel-top-k sampling relates to sampling (with replacement) and (deterministic) beam search and can be seen as a method that combines the advantages of both. Our experiments support the idea that Gumbel-top-k sampling can be used as a drop-in replacement in places where sampling (with replacement) or (deterministic) beam search is used. In fact, our experiments show that sampling without replacement can be used to yield lower-variance estimators and high-diversity samples from a machine translation model.

We hope that our method motivates future work to develop improved statistical learning methods, especially in the context of deep learning, based on sampling without replacement, a direction of research that has, in fact, already started (Kool et al., 2019b; Kool et al., 2020b) and is presented in Chapter 6. We believe that ancestral Gumbel-top-k sampling has potential to increase both computational and statistical efficiency in deep learning applications that involve discrete computations, such as combinatorial optimization (Part i of this thesis), neural machine translation (Sutskever et al., 2014; Bahdanau et al., 2015) and image captioning (Vinyals et al., 2015b).

6 ESTIMATING GRADIENTS WITH SAMPLES WITHOUT REPLACEMENT

We derive an unbiased estimator for expectations over discrete random variables based on sampling without replacement*, which reduces variance as it avoids duplicate samples. We show that our estimator can be derived as the Rao-Blackwellization of three different estimators. Combining our estimator with REINFORCE, we obtain a policy gradient estimator and we reduce its variance using a built-in control variate which is obtained without additional model evaluations. The resulting estimator is closely related to other gradient estimators. Experiments with a toy problem, a categorical variational auto-encoder and the travelling salesman problem show that our estimator is the only estimator that is consistently among the best estimators in both high and low entropy settings.*

6.1 INTRODUCTION

Put replacement in your basement! We derive the *unordered set estimator*[1]: an unbiased (gradient) estimator for expectations over discrete random variables based on (unordered sets of) samples *without replacement*. In particular, we consider the problem of estimating (the gradient of) the expectation of $f(\boldsymbol{x})$ where $\boldsymbol{x}$ has a discrete distribution p over the domain D, i.e.

$$\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}[f(\boldsymbol{x})] = \sum_{\boldsymbol{x}\in D} p(\boldsymbol{x}) f(\boldsymbol{x}). \tag{46}$$

This expectation comes up in reinforcement learning, discrete latent variable modelling (e.g. for compression), structured prediction (e.g. for translation), hard attention and many other tasks that use models with discrete operations in their computational graphs (see e.g. Jang et al. (2016)). In general, $\boldsymbol{x}$ has structure (such as a sequence), but we can treat it as a 'flat' distribution, omitting the bold notation, so x has a categorical distribution over D given by $p(x), x \in D$. Typically, the distribution has parameters $\boldsymbol{\theta}$, which are learnt through gradient descent. This requires estimating the gradient $\nabla_{\boldsymbol{\theta}}\mathbb{E}_{x\sim p_{\boldsymbol{\theta}}(x)}[f(x)]$, using a set of samples S. A gradient estimate $e(S)$ is *unbiased* if

$$\mathbb{E}_S[e(S)] = \nabla_{\boldsymbol{\theta}}\mathbb{E}_{x\sim p_{\boldsymbol{\theta}}(x)}[f(x)]. \tag{47}$$

The samples S can be sampled independently or using alternatives such as stratified sampling which reduce variance to increase the speed of learning. In this chapter, we derive an unbiased gradient estimator that reduces variance by avoiding duplicate samples, i.e. by sampling S without replacement. This is challenging as samples

1 Code available at https://github.com/wouterkool/estimating-gradients-without-replacement.

without replacement are dependent and have marginal distributions that are different from $p(x)$. We further reduce the variance by deriving a built-in control variate, which maintains the unbiasedness and does not require additional samples.

6.2 RELATED WORK

Many algorithms for estimating gradients for discrete distributions have been proposed. A general and widely used estimator is REINFORCE (Williams, 1992). Biased gradients based on a continuous relaxations of the discrete distribution (known as Gumbel-Softmax or Concrete) were jointly introduced by Jang et al. (2016) and Maddison et al. (2016). These can be combined with the straight-through estimator (Bengio et al., 2013) if the model requires discrete samples, or be used to construct control variates for REINFORCE, as in REBAR (Tucker et al., 2017) or RELAX (Grathwohl et al., 2018). Many other methods use control variates and other techniques to reduce the variance of REINFORCE (Paisley et al., 2012; Ranganath et al., 2014; Gregor et al., 2014; Mnih and Gregor, 2014; Gu et al., 2016; Mnih and Rezende, 2016). Some works rely on explicit summation of the expectation, either for the marginal distribution (Titsias and Lázaro-Gredilla, 2015) or globally summing some categories while sampling from the remainder (Liang et al., 2018; Liu et al., 2019). Another approach is to use a finite difference approximation to the gradient (Lorberbom et al., 2020). Yin et al. (2019) introduced ARSM, which uses multiple model evaluations where the number adapts automatically to the uncertainty.

In the structured prediction setting, there are many algorithms for optimizing a quantity under a sequence of discrete decisions, using (weak) supervision, multiple samples (or deterministic model evaluations), or a combination both (Ranzato et al., 2016; Shen et al., 2016; He et al., 2016a; Norouzi et al., 2016; Bahdanau et al., 2017; Edunov et al., 2018b; Leblond et al., 2018; Negrinho et al., 2018). Most of these algorithms are biased and rely on pretraining using maximum likelihood or gradually transitioning from supervised to reinforcement learning. Using Gumbel-Softmax based approaches in a sequential setting is difficult as the bias accumulates because of mixing errors (Gu et al., 2018).

6.3 PRELIMINARIES

Throughout this chapter, we will denote with B^k an *ordered* sample without replacement of size k and with S^k an *unordered* sample (of size k) from the categorical distribution p.

RESTRICTED DISTRIBUTION When sampling without replacement, we remove the set $C \subset D$ already sampled from the domain and we denote with $p^{D\setminus C}$ the distribution *restricted to the domain* $D \setminus C$:

$$p^{D\setminus C}(x) = \frac{p(x)}{1 - \sum_{c\in C} p(c)}, \quad x \in D \setminus C. \tag{48}$$

ORDERED SAMPLE WITHOUT REPLACEMENT B^k Let $B^k = (b_1, ..., b_k), b_i \in D$ be an *ordered sample without replacement*, which is generated from the distribution p as follows: first, sample $b_1 \sim p$, then sample $b_2 \sim p^{D\setminus\{b_1\}}$, $b_3 \sim p^{D\setminus\{b_1,b_2\}}$, etc. i.e. elements are sampled one by one without replacement. Using this procedure, B^k can be seen as a (partial) ranking according to the Plackett-Luce model (Plackett, 1975; Luce, 1959) and the probability of obtaining the vector B^k is

$$p(B^k) = \prod_{i=1}^{k} p^{D\setminus B^{i-1}}(b_i) = \prod_{i=1}^{k} \frac{p(b_i)}{1 - \sum_{j<i} p(b_j)}. \tag{49}$$

We can also restrict B^k to the domain $D \setminus C$, which means that $b_i \notin C$ for $i = 1, ..., k$:

$$p^{D\setminus C}(B^k) = \prod_{i=1}^{k} \frac{p^{D\setminus C}(b_i)}{1 - \sum_{j<i} p^{D\setminus C}(b_j)} = \prod_{i=1}^{k} \frac{p(b_i)}{1 - \sum_{c\in C} p(c) - \sum_{j<i} p(b_j)}. \tag{50}$$

UNORDERED SAMPLE WITHOUT REPLACEMENT Let $S^k \subseteq D$ be an *unordered* sample without replacement from the distribution p, which can be generated simply by generating an ordered sample and discarding the order. We denote elements in the sample with $s \in S^k$ (so without index) and we write $\mathcal{B}(S^k)$ as the set of all $k!$ permutations (orderings) B^k that correspond to (could have generated) S^k. It follows that the probability for sampling S^k is given by:

$$\begin{aligned} p(S^k) &= \sum_{B^k \in \mathcal{B}(S^k)} p(B^k) \\ &= \sum_{B^k \in \mathcal{B}(S^k)} \prod_{i=1}^{k} \frac{p(b_i)}{1 - \sum_{j<i} p(b_j)} \\ &= \left(\prod_{s\in S^k} p(s) \right) \cdot \sum_{B^k \in \mathcal{B}(S^k)} \prod_{i=1}^{k} \frac{1}{1 - \sum_{j<i} p(b_j)}. \end{aligned} \tag{51}$$

The last step follows since $B^k \in \mathcal{B}(S^k)$ is an ordering of S^k, such that $\prod_{i=1}^{k} p(b_i) = \prod_{s\in S} p(s)$. Naive computation of $p(S^k)$ is $O(k!)$, but in Appendix D.2 we show how to compute it efficiently.

When sampling from the distribution restricted to $D \setminus C$, we sample $S^k \subseteq D \setminus C$ with probability:

$$p^{D\setminus C}(S^k) = \left(\prod_{s\in S^k} p(s) \right) \cdot \sum_{B^k \in \mathcal{B}(S^k)} \prod_{i=1}^{k} \frac{1}{1 - \sum_{c\in C} p(c) - \sum_{j<i} p(b_j)}. \tag{52}$$

GUMBEL-TOP-k SAMPLING As an alternative to sequential sampling, we can also use Gumbel-top-k sampling (Section 5.2.4) and sample B^k and S^k by taking the top k of Gumbel variables (Yellott, 1977; Vieira, 2014; Kim et al., 2016). Using notation similar to Chapter 5, we define the *perturbed log-probability* $g_{\phi_i} = \phi_i + g_i$, where $\phi_i = \log p(i)$ and $g_i \sim \text{Gumbel}(0)$. Then let $b_1 = \arg\max_{i \in D} g_{\phi_i}$, $b_2 = \arg\max_{i \in D \setminus \{b_1\}} g_{\phi_i}$, etc., so B^k is the top k of the perturbed log-probabilities *in decreasing order*. The probability of obtaining B_k using this procedure is given by equation 49, so this provides an alternative sampling method which is effectively a (non-differentiable) reparameterization of sampling without replacement. For a differentiable reparameterization, see Grover et al. (2019).

It follows that taking the top k perturbed log-probabilities *without order*, we obtain the unordered sample set S^k. This way of sampling underlies the efficient computation of $p(S^k)$ in Appendix D.2.

6.4 METHODOLOGY

In this section, we derive the *unordered set policy gradient estimator*: a low-variance, unbiased estimator of $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[f(x)]$ based on an unordered sample without replacement S^k. First, we derive the generic (non-gradient) estimator for $\mathbb{E}[f(x)]$ as the Rao-Blackwellized version of a single sample Monte Carlo estimator (and two other estimators!). Then we combine this estimator with REINFORCE (Williams, 1992) and we show how to reduce its variance using a built-in baseline.

6.4.1 Rao-Blackwellization of the single sample estimator

A very crude but simple estimator for $\mathbb{E}[f(x)]$ based on the *ordered* sample B^k is to *only* use the first element b_1, which by definition is a sample from the distribution p. We define this estimator as the *single sample estimator*, which is unbiased, since

$$\mathbb{E}_{B^k \sim p(B^k)}[f(b_1)] = \mathbb{E}_{b_1 \sim p(b_1)}[f(b_1)] = \mathbb{E}_{x \sim p(x)}[f(x)]. \tag{53}$$

Discarding all but one sample, the single sample estimator is inefficient, but we can use Rao-Blackwellization (Casella and Robert, 1996) to signficantly improve it. To this end, we consider the distribution $B^k|S^k$, which is, knowing the unordered sample S^k, the conditional distribution over ordered samples $B^k \in \mathcal{B}(S^k)$ that could have generated S^k.[2] Using $B^k|S^k$, we rewrite $\mathbb{E}[f(b_1)]$ as

$$\mathbb{E}_{B^k \sim p(B^k)}[f(b_1)] = \mathbb{E}_{S^k \sim p(S^k)}\left[\mathbb{E}_{B^k \sim p(B^k|S^k)}\left[f(b_1)\right]\right] = \mathbb{E}_{S^k \sim p(S^k)}\left[\mathbb{E}_{b_1 \sim p(b_1|S^k)}\left[f(b_1)\right]\right].$$

2 Note that $B^k|S^k$ is *not* a Plackett-Luce distribution restricted to S^k!

The Rao-Blackwellized version of the single sample estimator computes the inner conditional expectation exactly. Since B^k is an ordering of S^k, we have $b_1 \in S^k$ and we can compute this as

$$\mathbb{E}_{b_1 \sim p(b_1|S^k)}[f(b_1)] = \sum_{s \in S^k} P(b_1 = s|S^k) f(s) \tag{54}$$

where, in a slight abuse of notation, $P(b_1 = s|S^k)$ is the probability that the first sampled element b_1 takes the value s, given that the complete set of k samples is S^k. Using Bayes' Theorem we find

$$P(b_1 = s|S^k) = \frac{p(S^k|b_1 = s)P(b_1 = s)}{p(S^k)} = \frac{p^{D\setminus\{s\}}(S^k \setminus \{s\})p(s)}{p(S^k)}. \tag{55}$$

The step $p(S^k|b_1 = s) = p^{D\setminus\{s\}}(S^k \setminus \{s\})$ comes from analyzing sequential sampling without replacement: given that the first element sampled is s, the remaining elements have a distribution restricted to $D \setminus \{s\}$, so sampling S^k (including s) given the first element s is equivalent to sampling the remainder $S^k \setminus \{s\}$ from the restricted distribution, which has probability $p^{D\setminus\{s\}}(S^k \setminus \{s\})$ (see equation 52).

THE UNORDERED SET ESTIMATOR For notational convenience, we introduce the *leave-one-out ratio*.

Definition 4. *The* leave-one-out ratio *of s w.r.t. the set S is given by*

$$R(S^k, s) = \frac{p^{D\setminus\{s\}}(S^k \setminus \{s\})}{p(S^k)}. \tag{56}$$

Rewriting equation 55 as $P(b_1 = s|S^k) = p(s)R(S^k, s)$ shows that the probability of sampling s first, given S^k, is simply the unconditional probability multiplied by the leave-one-out ratio. We now define the *unordered set estimator* as the Rao-Blackwellized version of the single sample estimator.

Theorem 5. *The* unordered set estimator, *given by*

$$e^{US}(S^k) = \sum_{s \in S^k} p(s)R(S^k, s)f(s) \tag{57}$$

is the Rao-Blackwellized version of the (unbiased!) single sample estimator.

Proof. Using $P(b_1 = s|S^k) = p(s)R(S^k, s)$ in equation 54 we have

$$\mathbb{E}_{b_1 \sim p(b_1|S^k)}[f(b_1)] = \sum_{s \in S^k} P(b_1 = s|S^k) f(s) = \sum_{s \in S^k} p(s)R(S^k, s)f(s). \tag{58}$$

□

The implication of this theorem is that the unordered set estimator, in explicit form given by equation 57, is an unbiased estimator of $\mathbb{E}[f(x)]$ since it is the Rao-

Blackwellized version of the unbiased single sample estimator. Also, as expected by taking multiple samples, it has variance equal or lower than the single sample estimator by the Rao-Blackwell theorem (Lehmann and Scheffé, 1950).

6.4.2 Rao-Blackwellization of other estimators

The unordered set estimator is also the result of Rao-Blackwellizing two other unbiased estimators: the *stochastic sum-and-sample* estimator and the *importance-weighted estimator*.

THE SUM-AND-SAMPLE ESTIMATOR We define as *sum-and-sample estimator* any estimator that relies on the identity that for any $C \subset D$

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \mathbb{E}_{x\sim p^{D\setminus C}(x)}\left[\sum_{c\in C} p(c)f(c) + \left(1 - \sum_{c\in C} p(c)\right) f(x)\right]. \tag{59}$$

For the derivation, see Appendix D.3.1 or Liang et al. (2018) and Liu et al. (2019). In general, a sum-and-sample estimator with a budget of $k > 1$ evaluations sums expectation terms for a set of categories C (s.t. $|C| < k$) explicitly (e.g. selected by their value f (Liang et al., 2018) or probability p (Liu et al., 2019)), and uses $k - |C|$ (down-weighted) samples from $D \setminus C$ to estimate the remaining terms. As is noted by Liu et al. (2019), selecting C such that $\frac{1-\sum_{c\in C} p(c)}{k-|C|}$ is minimized guarantees to reduce variance compared to a standard minibatch of k samples (which is equivalent to setting $C = \varnothing$). See also Fearnhead and Clifford (2003) for a discussion on selecting C optimally. The ability to optimize C depends on whether $p(c)$ can be computed efficiently a-priori (before sampling). This is difficult in high-dimensional settings, e.g. sequence models which compute the probability incrementally while ancestral sampling. An alternative is to select C stochastically (as equation 59 holds for any C), and we choose $C = B^{k-1}$ to define the *stochastic sum-and-sample* estimator:

$$e^{\text{SSAS}}(B^k) = \sum_{j=1}^{k-1} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-1} p(b_j)\right) f(b_k). \tag{60}$$

For simplicity, we consider the version that sums $k-1$ terms here, but the following results also hold for a version that sums $k-m$ terms and uses m samples (without replacement) (see Appendix D.3.3). Sampling without replacement, it holds that $b_k|B^{k-1} \sim p^{D\setminus B^{k-1}}$, so the unbiasedness follows from equation 59 by separating the expectation over B^k into expectations over B^{k-1} and $b_k|B^{k-1}$:

$$\mathbb{E}_{B^{k-1}\sim p(B^{k-1})}\left[\mathbb{E}_{b_k\sim p(b_k|B^{k-1})}\left[e^{\text{SSAS}}(B^k)\right]\right] = \mathbb{E}_{B^{k-1}\sim p(B^{k-1})}\left[\mathbb{E}[f(x)]\right] = \mathbb{E}[f(x)].$$

In general, a sum-and-sample estimator reduces variance if the probability mass is concentrated on the summed categories. As typically high probability categories are sampled first, the stochastic sum-and-sample estimator sums high probability categories, similar to the estimator by Liu et al. (2019) which we refer to as the *determin-*

istic sum-and-sample estimator. As we show in Appendix D.3.2, Rao-Blackwellizing the stochastic sum-and-sample estimator also results in the unordered set estimator. This even holds for a version that uses m samples and $k - m$ summed terms (see Appendix D.3.3), which means that the unordered set estimator has equal or lower variance than the optimal (in terms of m) stochastic sum-and-sample estimator, but conveniently does *not* need to choose m.

THE IMPORTANCE-WEIGHTED ESTIMATOR The importance-weighted estimator (Vieira, 2017) is

$$e^{\text{IW}}(S^k, \kappa) = \sum_{s \in S^k} \frac{p(s)}{q(s, \kappa)} f(s). \tag{61}$$

This estimator is based on the idea of priority sampling (Duffield et al., 2007). It does *not* use the order of the sample, but assumes sampling using Gumbel-top-k sampling and requires access to κ, the $(k+1)$-th largest perturbed log-probability, which can be seen as the 'threshold' since $g_{\phi_s} > \kappa \; \forall s \in S^k$. $q(s, a) = P(g_{\phi_s} > a)$ can be interpreted as the *inclusion probability* of $s \in S^k$ (assuming a fixed threshold a instead of a fixed sample size k). For details and a proof of unbiasedness, see Vieira (2017) and Kool et al. (2019c) or Appendix C.2. As the estimator has high variance, in Section 5.5.3, we *normalized* the importance weights, resulting in biased estimates. Instead, here we use Rao-Blackwellization to eliminate stochasticity by κ. Again, the result is the unordered set estimator (see Appendix D.4.1), which thus has equal or lower variance.

6.4.3 The unordered set policy gradient estimator

Writing $p_{\boldsymbol{\theta}}$ to indicate the dependency on the model parameters $\boldsymbol{\theta}$, we can combine the unordered set estimator with REINFORCE (Williams, 1992) to obtain the *unordered set policy gradient* estimator.

Corollary 6. *The* unordered set policy gradient estimator, *given by*

$$e^{USPG}(S^k) = \sum_{s \in S^k} p_{\boldsymbol{\theta}}(s) R(S^k, s) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(s) f(s) = \sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) R(S^k, s) f(s), \tag{62}$$

is an unbiased estimate of the policy gradient.

Proof. Using REINFORCE (Williams, 1992) combined with the unordered set estimator we find:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[f(x)] &= \mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(x) f(x)] \\ &= \mathbb{E}_{S^k \sim p_{\boldsymbol{\theta}}(S^k)} \left[\sum_{s \in S^k} p_{\boldsymbol{\theta}}(s) R(S^k, s) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(s) f(s) \right]. \end{aligned}$$

□

VARIANCE REDUCTION USING A BUILT-IN CONTROL VARIATE The variance of REINFORCE can be reduced by subtracting a baseline from f. When taking multiple samples (with replacement), a simple and effective baseline is to take the mean of other (independent!) samples (Mnih and Rezende, 2016). Sampling without replacement, we can use the same idea to construct a baseline based on the other samples, but we have to correct for the fact that the samples are *not* independent.

Theorem 7. *The* unordered set policy gradient estimator with baseline, *given by*

$$e^{USPGBL}(S^k) = \sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) R(S^k, s) \left(f(s) - \sum_{s' \in S^k} p_{\boldsymbol{\theta}}(s') R^{D \setminus \{s\}}(S^k, s') f(s') \right), \quad (63)$$

where

$$R^{D \setminus \{s\}}(S^k, s') = \frac{p_{\boldsymbol{\theta}}^{D \setminus \{s,s'\}}(S^k \setminus \{s, s'\})}{p_{\boldsymbol{\theta}}^{D \setminus \{s\}}(S^k \setminus \{s\})} \quad (64)$$

is the second order leave-one-out ratio, *is an unbiased estimate of the policy gradient.*

Proof. See Appendix D.5.1. □

This theorem shows how to include a built-in baseline based on *dependent* samples (without replacement), without introducing bias. By having a built-in baseline, the value $f(s)$ for a sample s is compared against an estimate of its expectation $\mathbb{E}[f(s)]$, based on the other samples. The difference is an estimate of the *advantage* (Sutton and Barto, 2018), which is positive if the sample s is 'better' than average, causing $p_{\boldsymbol{\theta}}(s)$ to be increased (reinforced) through the sign of the gradient, and vice versa. By sampling without replacement, the unordered set estimator forces the estimator to compare different alternatives, and reinforces the best among them.

INCLUDING THE PATHWISE DERIVATIVE So far, we have only considered the scenario where f does not depend on $\boldsymbol{\theta}$. If f does depend on $\boldsymbol{\theta}$, for example in a VAE (Kingma and Welling, 2014; Rezende et al., 2014), then we use the notation $f_{\boldsymbol{\theta}}$ and we can write the gradient (Schulman et al., 2015) as

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[f_{\boldsymbol{\theta}}(x)] = \mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(x) f_{\boldsymbol{\theta}}(x) + \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(x)]. \quad (65)$$

The additional second ('pathwise') term can be estimated (using the same samples) with the standard unordered set estimator. This results in the *full* unordered set policy gradient estimator:

$$\begin{aligned} e^{\text{FUSPG}}(S^k) &= \sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) R(S^k, s) f_{\boldsymbol{\theta}}(s) + \sum_{s \in S^k} p_{\boldsymbol{\theta}}(s) R(S^k, s) \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(s) \\ &= \sum_{s \in S^k} R(S^k, s) \nabla_{\boldsymbol{\theta}} \left(p_{\boldsymbol{\theta}}(s) f_{\boldsymbol{\theta}}(s) \right). \end{aligned} \quad (66)$$

Equation 66 is straightforward to implement using an automatic differentiation library. We can also include the baseline (as in equation 63) but we must make sure

to call STOP_GRADIENT (DETACH in PyTorch) on the baseline (but not on $f_\theta(s)$!). Importantly, we should *never* track gradients through the leave-one-out ratio $R(S^k, s)$ which means it can be efficiently computed in pure inference mode.

SCOPE & LIMITATIONS We can use the unordered set estimator for any discrete distribution from which we can sample without replacement, by treating it as a univariate categorical distribution over its domain. This includes all the distributions that can be sampled from using ancestral Gumbel-top-k sampling (Chapter 5): multivariate categorical distributions, sequence models or any other distributions that are represented as discrete-valued Bayesian networks (see Figure 11). In the presence of continuous variables or a stochastic function f, we may separate this stochasticity from the stochasticity over the discrete distribution, as in Lorberbom et al. (2020). The computation of the leave-one-out ratios adds some overhead, although they can be computed efficiently, even for large k (see Appendix D.2). For a moderately sized model, the costs of model evaluation and backpropagation dominate the cost of computing the estimator.

6.4.4 Relation to other multi-sample estimators

RELATION TO MURTHY'S ESTIMATOR We found out that the 'vanilla' unordered set estimator (equation 57) is actually a special case of the estimator by Murthy (1957), known in statistics literature for estimation of a population total $\Theta = \sum_{i \in D} y_i$. Using $y_i = p(i)f(i)$, we have $\Theta = \mathbb{E}[f(i)]$, so Murthy's estimator can be used to estimate expectations (see equation 57). Murthy derives the estimator by 'unordering' a convex combination of Raj (1956) estimators, which, using $y_i = p(i)f(i)$, are stochastic sum-and-sample estimators in our analogy.

Murthy (1957) also provides an unbiased estimator of the variance, which may be interesting for future applications. Since Murthy's estimator can be used with *arbitrary* sampling distribution, it is straightforward to derive importance-sampling versions of our estimators. In particular, we can sample S without replacement using $q(x) > 0, x \in D$, and use equations 57, 62, 63 and 66, as long as we compute the leave-one-out ratio $R(S^k, s)$ using q.

While part of our derivation coincides with Murthy (1957), we are not aware of previous work using this estimator to estimate expectations. Additionally, we discuss practical computation of $p(S)$ (Appendix D.2), we show the relation to the importance-weighted estimator, and we provide the extension to estimating policy gradients, especially including a built-in baseline without adding bias.

RELATION TO THE EMPIRICAL RISK ESTIMATOR The empirical risk loss (Edunov et al., 2018b) estimates the expectation in equation 46 by summing only a subset S

of the domain, using *normalized* probabilities $\hat{p}_{\boldsymbol{\theta}}(s) = \frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s' \in S} p_{\boldsymbol{\theta}}(s)}$. Using this loss, the (biased) estimate of the gradient is given by

$$e^{\text{RISK}}(S^k) = \sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} \left(\frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s' \in S^k} p_{\boldsymbol{\theta}}(s')} \right) f(s). \tag{67}$$

The risk estimator is similar to the unordered set policy gradient estimator, with two important differences: 1) the individual terms are normalized by the total probability mass rather than the leave-one-out ratio and 2) the gradient w.r.t. the normalization factor is taken into account. As a result, samples 'compete' for probability mass and only the best can be reinforced. This has the same effect as using a built-in baseline, which we prove in the following theorem.

Theorem 8. *By taking the gradient w.r.t. the normalization factor into account, the risk estimator has a built-in baseline, which means it can be written as*

$$e^{RISK}(S^k) = \sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) \frac{1}{\sum_{s'' \in S^k} p_{\boldsymbol{\theta}}(s'')} \left(f(s) - \sum_{s' \in S^k} p_{\boldsymbol{\theta}}(s') \frac{1}{\sum_{s'' \in S^k} p_{\boldsymbol{\theta}}(s'')} f(s') \right). \tag{68}$$

Proof. See Appendix D.6.1 □

This theorem highlights the similarity between the biased risk estimator and our unbiased estimator (equation 63), and suggests that their only difference is the weighting of terms. Unfortunately, the implementation by Edunov et al. (2018b) has more sources of bias (e.g. length normalization), which are not compatible with our estimator. However, we believe that our analysis helps analyze the bias of the risk estimator and is a step towards developing unbiased estimators for structured prediction.

RELATION TO VIMCO VIMCO (Mnih and Rezende, 2016) is an estimator that uses k samples (with replacement) to optimize an objective of the form $\log \frac{1}{k} \sum_i f(x_i)$, which is a multi-sample stochastic lower bound in the context of variational inference. VIMCO reduces the variance by using a *local* baseline for each of the k samples, based on the other $k-1$ samples. While we do not have a log term, as our goal is to optimize general $\mathbb{E}[f(x)]$, we adopt the idea of forming a baseline based on the other samples, and we define *REINFORCE with replacement* (with built-in baseline) as the estimator that computes the gradient estimate using samples with replacement $X^k = (x_1, ..., x_k)$ as

$$e^{\text{RFWR}}(X^k) = \frac{1}{k} \sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(x_i) \left(f(x_i) - \frac{1}{k-1} \sum_{j \neq i} f(x_j) \right). \tag{69}$$

This estimator is unbiased, as $\mathbb{E}_{x_i, x_j}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(x_i) f(x_j)] = 0$ for $i \neq j$ (see also Kool et al. (2019b)). We think of the unordered set estimator as the without-replacement version of this estimator, which weights terms by $p_{\boldsymbol{\theta}}(s) R(S^k, s)$ instead of $\frac{1}{k}$. This puts more weight on higher probability elements to compensate for sampling with-

out replacement. If probabilities are small and (close to) uniform, there are (almost) no duplicate samples and the weights will be close to $\frac{1}{k}$, so the gradient estimate of the with- and without-replacement versions are similar.

RELATION TO ARSM ARSM (Yin et al., 2019) also uses multiple evaluations ('pseudo-samples') of p_{θ} and f. This can be seen as similar to sampling without replacement, and the estimator also has a built-in control variate. Compared to ARSM, our estimator allows direct control over the computational cost (through the sample size k) and has wider applicability, for example it also applies to multivariate categorical variables with different numbers of categories per dimension.

RELATION TO STRATIFIED/SYSTEMATIC SAMPLING Our estimator aims to reduce variance by changing the sampling distribution for multiple samples by sampling without replacement. There are alternatives, such as using stratified or systematic sampling (see, e.g. Douc and Cappé (2005)). Both partition the domain D into k strata and take a single sample from each stratum, where systematic sampling uses common random numbers for each stratum. In applications involving high-dimensional or structured domains, it is unclear how to partition the domain and how to sample from each partition. Additionally, as samples are *not* independent, it is non-trivial to include a built-in baseline, which we find is a key component that makes our estimator perform well.

6.5 EXPERIMENTS

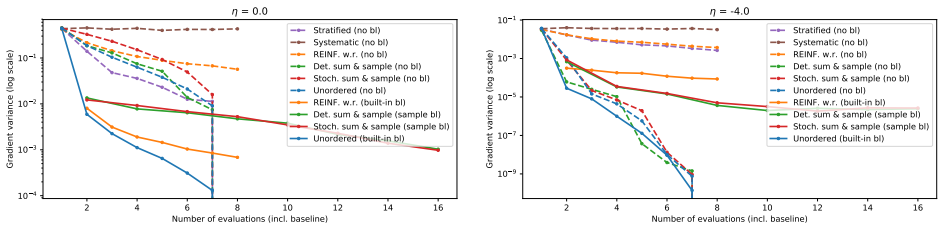
6.5.1 Bernoulli toy experiment

We use the code by Liu et al. (2019) to reproduce their Bernoulli toy experiment. Given a vector $\boldsymbol{p} = (0.6, 0.51, 0.48)$, the goal is to minimize the loss

$$\mathcal{L}(\eta) = \mathbb{E}_{x_1,x_2,x_3\sim\text{Bern}(\sigma(\eta))}\left[\sum_{i=1}^{3}(x_i - p_i)^2\right].$$

Here x_1, x_2, x_3 are i.i.d. from the Bernoulli$(\sigma(\eta))$ distribution, parameterized by a scalar $\eta \in \mathbb{R}$, where $\sigma(\eta) = (1 + \exp(-\eta))^{-1}$ is the sigmoid function. We compare different estimators, with and without baseline (either 'built-in' or using additional samples, referred to as REINFORCE+ in Liu et al. (2019)). We report the (log-)variance of the scalar gradient $\frac{\partial \mathcal{L}}{\partial \eta}$ as a function of the number of model evaluations, which is twice as high when using a sampled baseline (for each term).

As can be seen in Figure 24, the unordered set estimator is the only estimator that has consistently the lowest (or comparable) variance in both the high ($\eta = 0$) and low entropy ($\eta = -4$) regimes and for different number of samples/model evaluations. This suggests that it combines the advantages of the other estimators.



(a) High entropy ($\eta = 0$) **(b)** Low entropy ($\eta = -4$)

Figure 24: Bernoulli gradient variance (on log scale) as a function of the number of model evaluations (including baseline evaluations, so the sum-and-sample estimators with sampled baselines use twice as many evaluations). Note that for some estimators, the variance is 0 (log variance $-\infty$) for $k = 8$.

We also ran the actual optimization experiment, where with as few as $k = 3$ samples the trajectory was indistinguishable from using the exact gradient (see Liu et al. (2019)).

6.5.2 Categorical variational auto-encoder

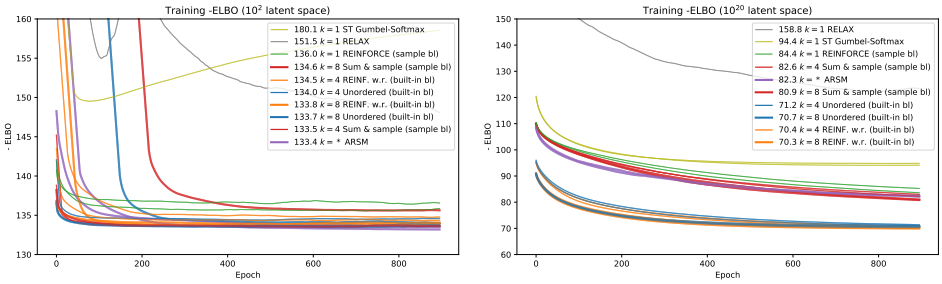
We use the code from Yin et al. (2019) to train a *categorical* or *discrete variational auto-encoder* (VAE) with 20 dimensional latent space, with 10 categories per dimension (details in Appendix D.7.1). To use our estimator, we treat this as a single factorized distribution with 10^{20} categories from which we can sample without replacement using ancestral Gumbel-top-k sampling (Chapter 5 and Kool et al. (2020a)). We also perform experiments with 10^2 latent space, which provides a lower entropy setting, to highlight the advantage of our estimator.

MEASURING THE VARIANCE In Table 5, we report the variance of different gradient estimators with $k = 4$ samples, evaluated on a trained model. The unordered set estimator has the lowest variance in both the small and large domain (low and high entropy) setting, being on-par with the best of the (stochastic[3]) sum-and-sample estimator and REINFORCE with replacement[4]. This confirms the toy experiment, suggesting that the unordered set estimator provides the best of both estimators. In Appendix D.7.2 we repeat the same experiment at different stages of training, with similar results.

3 We cannot use the deterministic version by Liu et al. (2019) since we cannot select the top k categories.
4 We cannot compare against VIMCO (Mnih and Rezende, 2016) as it optimizes a different objective.

Table 5: VAE gradient log-variance of different unbiased estimators with $k = 4$ samples.

Domain	ARSM	RELAX	REINFORCE (no bl)	REINFORCE (sample bl)	Sum & sample (no bl)	Sum & sample (sample bl)	REINF. w.r. (built-in bl)	Unordered (built-in bl)
Small 10^2	13.45	11.67	11.52	7.49	**6.29**	**6.29**	6.65	**6.29**
Large 10^{20}	15.55	15.86	13.81	8.48	13.77	8.44	**7.06**	**7.05**



(a) Small domain (latent space size 10^2) **(b)** Large domain (latent space size 10^{20})

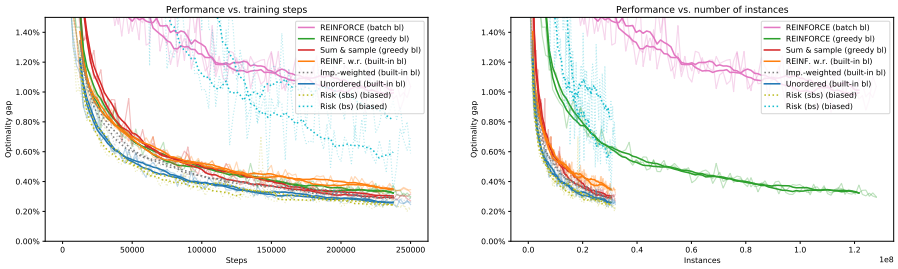
Figure 25: VAE smoothed training curves (-ELBO) of two independent runs when training with different estimators with $k = 1$, 4 or 8 (thicker lines) samples (ARSM has a variable number). Some lines coincide, so we sort the legend by the lowest -ELBO achieved and report this value.

ELBO OPTIMIZATION We use different estimators to optimize the ELBO (details in Appendix D.7.1). Additionally to the baselines by Yin et al. (2019) we compare against REINFORCE with replacement and the stochastic sum-and-sample estimator. In Figure 25 we observe that our estimator performs on par with REINFORCE with replacement (and built-in baseline, equation 69) and outperforms other estimators in at least one of the settings. There are a lot of other factors, e.g. exploration that may explain why we do not get a strictly better result despite the lower variance. We note some overfitting (see validation curves in Appendix D.7.2), but since our goal is to show improved optimization, and to keep results directly comparable to Yin et al. (2019), we consider regularization a separate issue outside the scope of this work. These results are using MNIST binarized using a threshold of 0.5. In Appendix D.7.2 we report results using the standard binarized MNIST dataset from Salakhutdinov and Murray (2008).

6.5.3 Structured prediction for the travelling salesman problem

To show the wide applicability of our estimator, we consider the structured prediction task of predicting routes (sequences) for the travelling salesman problem (TSP) (Vinyals et al., 2015a; Bello et al., 2016; Kool et al., 2019a). We build on the TSP experiment with 20 nodes from Chapter 3. For details, see Appendix D.8.

We implement REINFORCE with replacement (and built-in baseline) as well as the stochastic sum-and-sample estimator and our estimator, using ancestral Gumbel-top-k sampling (Chapter 5 and Kool et al. (2020a)) for sampling, specifically the fully-parallel version known as stochastic beam search (Kool et al., 2019c). Also, we include results using the biased normalized importance-weighted policy gradient estimator with built-in baseline (derived in Kool et al. (2019b), see Appendix D.4.2). Additionally, we compare against REINFORCE with greedy rollout baseline (Rennie et al., 2017; Kool et al., 2019a) used in Chapter 3 and a batch-average baseline. For reference, we also include the biased risk estimator, 'sampling' either using stochastic or deterministic beam search (as in Edunov et al. (2018b)).



(a) Performance vs. training steps

(b) Performance vs. number of instances

Figure 26: TSP validation set optimality gap measured during training. Raw results are light, smoothed results are darker (2 random seeds). We compare our estimator against different unbiased and biased (dotted) multi-sample estimators and against single sample REINFORCE, with batch-average or greedy rollout baseline.

In Figure 26a, we compare training progress (measured on the validation set) as a function of the number of training steps, where we divide the batch size by k to keep the total number of samples equal. Our estimator outperforms REINFORCE with replacement, the stochastic sum-and-sample estimator and the strong greedy rollout baseline (which uses additional baseline model evaluations) and performs on-par with the biased risk estimator. In Figure 26b, we plot the same results against the number of instances, which shows that, compared to the single sample estimators, we can train with less data and less computational cost (as we only need to run the encoder once for each instance).

6.6 DISCUSSION

We introduced the unordered set estimator, a low-variance, unbiased gradient estimator based on sampling without replacement, which can be used as an alternative to the popular biased Gumbel-Softmax estimator (Jang et al., 2016; Maddison et al., 2016). Our estimator is the result of Rao-Blackwellizing three existing estimators, which guarantees equal or lower variance, and is closely related to a number of other estimators. It has wide applicability, is parameter free (except for the sample size k) and has competitive performance to the best of alternatives in both high and low entropy regimes.

In our experiments, we also found that REINFORCE *with* replacement, with multiple samples and a built-in baseline as inspired by VIMCO (Mnih and Rezende, 2016), is a simple yet strong estimator which has performance similar to our estimator in the high entropy setting. Independently of our work, the same estimator has been presented by Luo (2020), but we are not aware of any other recent work on gradient estimators for discrete distributions that has considered this estimator, while it may be often preferred given its simplicity.

7 CONCLUSION

Machine learning has the exciting potential to revolutionarize the way we solve (combinatorial) optimization problems today. This thesis presented a set of ideas, techniques, experiments and results to inspire research in this direction, which has become known as *neural combinatorial optimization*. With a focus on solving vehicle routing problems, this thesis also presented methods for sampling from neural network models defined over combinatorial spaces, and optimizing their parameters.

Part i

In Part i of this thesis, we presented some of the first methods to use machine learning for combinatorial optimization, especially vehicle routing. While the proposed methods have not provided an immediate and unconditional improvement of the state-of-the-art for vehicle routing, they have shown impressive results obtained using a different paradigm for optimization: learning to optimize. Whereas traditional techniques rely on search to explicitly (or implicitly) consider many solutions, the methods proposed in Chapter 3 and 4 yield competitive solutions while considering a much smaller fraction of the search space.

The attention model (Chapter 3) is able to create a high quality tour in a single sequential construction, which is completely different from using search or local improvement. This has the potential to address new types of optimization problems as well, e.g. with stochasticity such as the PCTSP in Chapter 3. In Chapter 4, we have illustrated how machine learning is able to identify promising edges for vehicle routing, which is much better than using a naïve alternative such as using the cost (distance) of the edges. In a hybrid algorithm, combined with dynamic programming, this gives a significant improvement over the classic DP algorithm.

The methods presented in Part i provide answers to our research questions 1 and 2: Chapter 3 gives a clear example of how we can use reinforcement learning to solve vehicle routing problems and Chapter 4 addresses routing problems by integrating supervised machine learning and dynamic programming, as an example to combine learning and exact optimization. More importantly, the methods we proposed have sparked a large interest in learning based algorithms for vehicle routing and other optimization problems. For a review of these see e.g. Mazyavkina et al. (2020), Vesselinova et al. (2020), and Bai et al. (2021). Research in the area of *neural combinatorial optimization* happens at an increasingly fast pace: for example the work in Chapter 4 was based on the model developed by Joshi et al. (2019a), which itself was inspired by the attention model proposed in Chapter 3.

Part ii

In Part ii of this thesis, we presented various methods that were originally motivated by our search for better methods to optimize deep learning models for combinatorial optimization, especially the attention model in Chapter 3. However, as the methods are equally applicable to other machine learning tasks involving structured models over combinatorial spaces, they are described in general machine learning terminology and include results in different application domains, such as machine translation and latent variable modelling. We may say that Part i of this thesis used ideas from the machine learning community to solve optimization problems, whereas Part ii gives back the results of our effort to improve techniques to optimize models over combinatorial spaces.

In Chapter 5, we presented ancestral Gumbel-top-k sampling, which was the result of further developing stochastic beam search (Kool et al., 2019c). Ancestral Gumbel-top-k sampling is an efficient, but primarily very *elegant* method to draw multiple samples without replacement from a structured model over a combinatorial domain, and provides an answer to research question 3. In Chapter 6, we have illustrated how the resulting samples without replacement can be used to define a gradient estimator that improves training performance for various models, including the attention model from Chapter 3, providing an answer to research question 4. Whereas the methods presented in Part ii of this thesis are advanced methods that specifically address optimization of structured models using sampling without replacement, they have also contributed to popularizing the underlying ideas of Gumbel-top-k sampling (without a structured model), and the leave-one-out baseline for REINFORCE (when sampling *with* replacement).

Building bridges between communities

This thesis contains research on the intersection of machine learning (ML) and operations research (OR), with a focus on vehicle routing. Most of the research presented in the different chapters has been published in the ML community, but the work has reached an audience in the OR community as well. Both communities are now working on the integration of the two disciplines, although there is still a clear gap that can be observed: researchers from the ML community tend to disregard or misrepresent baselines or techniques from the OR community (Accorsi et al., 2021), whereas researchers from the OR community often use relatively simple ML techniques, leaving a lot of potential from deep learning underexplored. Various initiatives aim to bridge this gap, for example by providing the Ecole environment (Prouvost et al., 2021), releasing benchmark datasets (Queiroga et al., 2022) or organizing workshops or competitions (Gasse, 2021).

Over the past decades, the performance of OR algorithms has improved by over a million times, using a combination of algorithmic improvements and better hardware, primarily faster CPU cores. With an enormous effort, a similar improvement of neural network performance has been achieved in the ML community in

recent years, with both algorithmic improvements and massive scaling of computational resources through parallelization. As a result, we believe that new scientific breakthroughs are within reach when both communities cooperate in combining state-of-the-art techniques to solve problems at scales or complexities that were previously considered impossible. Scaling to larger instances, dealing with different constraints, parallelization of OR algorithms and combining them with neural networks effectively (see also Section 2.3) are just a few of the challenges to be overcome. When these are addressed successfully, we may expect significant improvements in the state-of-the-art of combinatorial optimization. Some of the effects of ongoing efforts are already visible, e.g. multiple improved solutions for well-known MIPLIB benchmark instances (Gleixner et al., 2021) have been found recently (Nair et al., 2020).

Optimizing the world

We can only imagine what the world will look like 10, 20 or 50 years from now. Inevitably, major parts of the infrastructure that enables our society will be controlled by 'artificial intelligence' (as that sounds fancier than 'algorithms'). This provides a tremendous opportunity for optimizing the way in which we use the scarce resources available on earth. Time, space and energy (to name just a few) should be allocated carefully to best cater to our (inflating) demands in a sustainable manner. *Optimizing the world* is not a luxury: it is a necessity for a sustainable future. Important decisions should be made with increasing frequency, and at an increasingly fast pace, not only taking into account available data, but also in the face of uncertainty; all of this in a continuously changing environment. As we simply cannot keep a human in the loop for every decision, machines must learn to make decisions: they must not only predict, but also *act* automatically. In our complex society, machines must learn *what* the problem is and *how* to optimize it (see also Section 2.3). If we continue on the road to success, machine learning will be the technology that enables *us* to learn what the optimized world of tomorrow will look like. *Let's Go!*

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support, in different forms, from a number of people. First of all, I'd like to thank **Joaquim Gromicho**, for sparking within me the idea to pursue a PhD in the first place, then helping me to make it possible, and finally supporting me in various ways to complete it. Without him, I would have never had the opportunity to start (and complete) this fantastic endeavour. I am extremely grateful to **Gerrit Timmer**, for providing me with the freedom and opportunity to spend the majority of my time over the past few years on this research adventure, from the comfort of my employment at ORTEC.

I also thank Gerrit for convincing **Max Welling** to take me as his PhD student at the University of Amsterdam (UvA), to become part of the excellent Amsterdam Machine Learning Lab (AMLab). Max has been an amazing advisor who, despite his busy agenda, was able to keep up and think along with my research and provide me with excellent feedback, pushing me to set ambitious goals. Continuing the chain of acknowledgement, I'd like to thank Max for quickly onboarding **Herke van Hoof** into my PhD journey. Herke has been an excellent supervisor whom I really enjoyed working with. I'm truly amazed by his capabilities to follow my line of thought, even when it's not clear to myself (yet) and outside of his direct expertise. I could not have wished for a better daily supervisor. I'd like to thank my many **colleagues at AMLab**, for everything I learned from our conversations, but primarily for sharing many memorable moments during a summer school, various conference visits, a fantastic California road trip and multiple AMLab retreats.

I'd like to thank **Andriy Mnih** and **Chris Maddison** for hosting me during my internship at DeepMind in London in 2021. It was truly an amazing experience to be part of one of the best research labs in artificial intelligence. I have learned a lot from incredibly smart people in a short period of time. I'd like to thank multiple **colleagues at DeepMind**, especially **Matthias Bauer**, for making me feel welcome despite the work-from-home restrictions. My experience would not have been the same without you. Back in the Netherlands, I'd like to thank **Iris Huijben** from TU Eindhoven for an enjoyable collaboration with many interesting conversations.

I am privileged to have been able to carry out this research with the loving support from my **friends and family**, especially my wife **Fenny Postma**. Thanks for encouraging me, believing in me, baking a cake when I submitted my first paper (in the middle of the night) and temporarily moving to London with me. I could not have done without your (virtually) endless support. Finally, thanks to my beautiful daughter **Floor Postma**, born on 25th of March 2022: you set a helpful (stochastic) deadline for the completion of this thesis. I'm looking very much forward to being your *papa with a PhD*.

BIBLIOGRAPHY

Abadi, Martin, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. (2016). "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." In: *arXiv preprint arXiv:1603.04467*.

Accorsi, Luca, Andrea Lodi, and Daniele Vigo (2021). "Guidelines for the Computational Testing of Machine Learning approaches to Vehicle Routing Problems." In: *arXiv preprint arXiv:2109.13983*.

Accorsi, Luca and Daniele Vigo (2021). "A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems." In: *Transportation Science* 55.4, pp. 832–856.

Applegate, David, Robert Bixby, Vasek Chvatal, and William Cook (2006). *Concorde TSP Solver*. URL: http://www.math.uwaterloo.ca/tsp/concorde.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton (2016). "Layer normalization." In: *Deep Learning Symposium at Neural Information Processing Systems (NeurIPS)*.

Bahdanau, Dzmitry, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio (2017). "An actor-critic algorithm for sequence prediction." In: *International Conference on Learning Representations (ICLR)*.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural machine translation by jointly learning to align and translate." In: *International Conference on Learning Representations (ICLR)*.

Bai, Ruibin, Xinan Chen, Zhi-Long Chen, Tianxiang Cui, Shuhui Gong, Wentao He, Xiaoping Jiang, Huan Jin, Jiahuan Jin, Graham Kendall, et al. (2021). "Analytics and Machine Learning in Vehicle Routing Research." In: *arXiv preprint arXiv:2102.10012*.

Balas, Egon (1989). "The prize collecting traveling salesman problem." In: *Networks* 19.6, pp. 621–636.

Balog, Matej, Nilesh Tripuraneni, Zoubin Ghahramani, and Adrian Weller (2017). "Lost Relatives of the Gumbel Trick." In: *International Conference on Machine Learning (ICML)*, pp. 371–379.

Barnhart, Cynthia, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance (1998). "Branch-and-price: Column generation for solving huge integer programs." In: *Operations research* 46.3, pp. 316–329.

Bellman, Richard (1952). "On the theory of dynamic programming." In: *Proceedings of the National Academy of Sciences of the United States of America* 38.8, p. 716.

Bellman, Richard (1962). "Dynamic programming treatment of the travelling salesman problem." In: *Journal of the ACM (JACM)* 9.1, pp. 61–63.

Bello, Irwan, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio (2016). "Neural combinatorial optimization with reinforcement learning." In: *arXiv preprint arXiv:1611.09940*.

Ben-Tal, Aharon and Arkadi Nemirovski (2002). "Robust optimization–methodology and applications." In: *Mathematical programming* 92.3, pp. 453–480.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville (2013). "Estimating or propagating gradients through stochastic neurons for conditional computation." In: *arXiv preprint arXiv:1308.3432*.

Bengio, Yoshua, Andrea Lodi, and Antoine Prouvost (2021). "Machine learning for combinatorial optimization: a methodological tour d'horizon." In: *European Journal of Operational Research* 290.2, pp. 405–421.

Bertsekas, Dimitri (2017). *Dynamic programming and optimal control: Volume I*. Vol. 1. Athena scientific.

Bertsimas, Dimitris, David B Brown, and Constantine Caramanis (2011). "Theory and applications of robust optimization." In: *SIAM review* 53.3, pp. 464–501.

Bishop, Christopher M (2006). "Pattern recognition." In: *Machine learning* 128.9.

Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. URL: http://github.com/google/jax.

Bräysy, Olli and Michel Gendreau (2005). "Vehicle routing problem with time windows, Part I: Route construction and local search algorithms." In: *Transportation science* 39.1, pp. 104–118.

Bronstein, Michael M, Joan Bruna, Taco Cohen, and Petar Veličković (2021). "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges." In: *arXiv preprint arXiv:2104.13478*.

Burke, Edmund K, Edmund K Burke, Graham Kendall, and Graham Kendall (2014). *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer.

Cappart, Quentin, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre Cire (2021). "Combining reinforcement learning and constraint programming for combinatorial optimization." In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Carlson, James A, Arthur Jaffe, and Andrew Wiles (2006). *The millennium prize problems*. Citeseer.

Casella, George and Christian P Robert (1996). "Rao-Blackwellisation of sampling schemes." In: *Biometrika* 83.1, pp. 81–94.

Chen, Xinyun and Yuandong Tian (2019). "Learning to Perform Local Rewriting for Combinatorial Optimization." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 6281–6292.

Chen, Yutian and Zoubin Ghahramani (2016). "Scalable discrete sampling as a multi-armed bandit problem." In: *International Conference on Machine Learning (ICML)*, pp. 2492–2501.

Cook, Stephen (2006). "The P versus NP problem." In: *The millennium prize problems*, pp. 87–104.

Cook, William and Paul Seymour (2003). "Tour merging via branch-decomposition." In: *INFORMS Journal on Computing* 15.3, pp. 233–248.

Costa, Paulo Roberto de O da, Jason Rhuggenaath, Yingqian Zhang, and Alp Akcay (2020). "Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning." In: *Asian Conference on Machine Learning (ACML)*.

Da Silva, Rodrigo Ferreira and Sebastián Urrutia (2010). "A General VNS heuristic for the traveling salesman problem with time windows." In: *Discrete Optimization* 7.4, pp. 203–211.

Dai, Hanjun, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song (2017). "Learning Combinatorial Optimization Algorithms over Graphs." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 6348–6358.

Daumé, Hal, John Langford, and Daniel Marcu (2009). "Search-based structured prediction." In: *Machine learning* 75.3, pp. 297–325.

Daumé III, Hal and Daniel Marcu (2005). "Learning as search optimization: Approximate large margin methods for structured prediction." In: *International Conference on Machine Learning (ICML)*, pp. 169–176.

Delarue, Arthur, Ross Anderson, and Christian Tjandraatmadja (2020). "Reinforcement Learning with Combinatorial Actions: An Application to Vehicle Routing." In: *Advances in Neural Information Processing Systems (NeurIPS)* 33.

Deudon, Michel, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau (2018). "Learning Heuristics for the TSP by Policy Gradient." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. Springer, pp. 170–181.

Dijkstra, Edsger W (1959). "A note on two problems in connexion with graphs." In: *Numerische mathematik* 1.1, pp. 269–271.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. (2021). "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." In: *International Conference on Learning Representations (ICLR)*.

Douc, Randal and Olivier Cappé (2005). "Comparison of resampling schemes for particle filtering." In: *International Symposium on Image and Signal Processing and Analysis (ISPA)*. IEEE, pp. 64–69.

Duffield, Nick, Carsten Lund, and Mikkel Thorup (2005). "Learn more, sample less: control of volume and variance in network measurement." In: *Transactions on Information Theory* 51.5, pp. 1756–1775.

– (2007). "Priority sampling for estimation of arbitrary subset sums." In: *Journal of the ACM (JACM)* 54.6, p. 32.

Dumas, Yvan, Jacques Desrosiers, Eric Gelinas, and Marius M Solomon (1995). "An optimal algorithm for the traveling salesman problem with time windows." In: *Operations Research* 43.2, pp. 367–371.

Edunov, Sergey, Myle Ott, Michael Auli, and David Grangier (2018a). "Understanding Back-Translation at Scale." In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 489–500.

Edunov, Sergey, Myle Ott, Michael Auli, David Grangier, and Marc'Aurelio Ranzato (2018b). "Classical Structured Prediction Losses for Sequence to Sequence Learning." In: *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 355–364.

Efraimidis, Pavlos S and Paul G Spirakis (2006). "Weighted random sampling with a reservoir." In: *Information Processing Letters* 97.5, pp. 181–185.

Elmachtoub, Adam N and Paul Grigas (2021). "Smart "predict, then optimize"." In: *Management Science*.

Ene, Alina, Viswanath Nagarajan, and Rishi Saket (2018). "Approximation Algorithms for Stochastic k-TSP." In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*.

Ermon, Stefano, Carla P Gomes, Ashish Sabharwal, and Bart Selman (2013). "Embed and project: Discrete sampling with universal hashing." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2085–2093.

Falkner, Jonas K and Lars Schmidt-Thieme (2020). "Learning to Solve Vehicle Routing Problems with Time Windows through Joint Attention." In: *arXiv preprint arXiv:2006.09100*.

Fearnhead, Paul and Peter Clifford (2003). "On-line inference for hidden Markov models via particle filters." In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 65.4, pp. 887–899.

Fischetti, Matteo, Juan Jose Salazar Gonzalez, and Paolo Toth (1998). "Solving the orienteering problem through branch-and-cut." In: *INFORMS Journal on Computing* 10.2, pp. 133–148.

Fu, Zhang-Hua, Kai-Bin Qiu, and Hongyuan Zha (2021). "Generalize a Small Pretrained Model to Arbitrarily Large TSP Instances." In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Gao, Lei, Mingxiang Chen, Qichang Chen, Ganzhong Luo, Nuoyi Zhu, and Zhixin Liu (2020). "Learn to design the heuristics for vehicle routing problem." In: *International Workshop on Heuristic Search in Industry (HSI) at the International Joint Conference on Artificial Intelligence (IJCAI)*.

Garey, Michael R and David S Johnson (1979). "Computers and intractability: A guide to the theory of NP-completeness (series of books in the mathematical sciences), ed." In: *Computers and Intractability*, p. 340.

Gasarch, William I (2012). "Guest column: The second p=? np poll." In: *ACM SIGACT News* 43.2, pp. 53–77.

Gasse, Maxime (2021). *Machine Learning for Combinatorial Optimization*. URL: https://www.ecole.ai/2021/ml4co-competition/.

Gasse, Maxime, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi (2019). "Exact Combinatorial Optimization with Graph Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Gehring, Jonas, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin (2017). "Convolutional Sequence to Sequence Learning." In: *International Conference on Machine Learning (ICML)*, pp. 1243–1252.

Glasserman, Paul and David D Yao (1992). "Some guidelines and guarantees for common random numbers." In: *Management Science* 38.6, pp. 884–908.

Gleixner, Ambros, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. (2021). "MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library." In: *Mathematical Programming Computation* 13.3, pp. 443–490.

Golden, Bruce L, Larry Levy, and Rakesh Vohra (1987). "The orienteering problem." In: *Naval Research Logistics (NRL)* 34.3, pp. 307–318.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Grathwohl, Will, Dami Choi, Yuhuai Wu, Geoffrey Roeder, and David Duvenaud (2018). "Backpropagation through the void: Optimizing control variates for black-box gradient estimation." In: *International Conference on Learning Representations (ICLR)*.

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton (2013). "Speech recognition with deep recurrent neural networks." In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6645–6649.

Gregor, Karol, Ivo Danihelka, Andriy Mnih, Charles Blundell, and Daan Wierstra (2014). "Deep AutoRegressive Networks." In: *International Conference on Machine Learning (ICML)*, pp. 1242–1250.

Gromicho, Joaquim, Jelke J van Hoorn, Adrianus Leendert Kok, and Johannes MJ Schutten (2012a). "Restricted dynamic programming: a flexible framework for solving realistic VRPs." In: *Computers & Operations Research* 39.5, pp. 902–909.

Gromicho, Joaquim AS, Jelke J Van Hoorn, Francisco Saldanha-da-Gama, and Gerrit T Timmer (2012b). "Solving the job-shop scheduling problem optimally by dynamic programming." In: *Computers & Operations Research* 39.12, pp. 2968–2977.

Grover, Aditya, Eric Wang, Aaron Zweig, and Stefano Ermon (2019). "Stochastic Optimization of Sorting Networks via Continuous Relaxations." In: *International Conference on Learning Representations (ICLR)*.

Gu, Jiatao, Daniel Jiwoong Im, and Victor OK Li (2018). "Neural machine translation with Gumbel-greedy decoding." In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Gu, Shixiang, Sergey Levine, Ilya Sutskever, and Andriy Mnih (2016). "Muprop: Unbiased backpropagation for stochastic neural networks." In: *International Conference on Learning Representations (ICLR)*.

Gumbel, Emil Julius (1954). *Statistical theory of extreme values and some practical applications: a series of lectures*. 33. US Govt. Print. Office.

Gurobi Optimization, LLC (2022). *Gurobi Optimizer Reference Manual*. URL: https://www.gurobi.com.

Hazan, Tamir and Tommi Jaakkola (2012). "On the partition function and random maximum a-posteriori perturbations." In: *International Conference on Machine Learning (ICML)*. Omnipress, pp. 1667–1674.

Hazan, Tamir, Subhransu Maji, and Tommi Jaakkola (2013). "On sampling from the Gibbs distribution with random maximum a-posteriori perturbations." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1268–1276.

He, Di, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma (2016a). "Dual learning for machine translation." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 820–828.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016b). "Deep residual learning for image recognition." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Heeswijk, Wouter van and Han La Poutré (2019). "Approximate dynamic programming with neural networks in linear discrete action spaces." In: *arXiv preprint arXiv:1902.09855*.

Held, Michael and Richard M Karp (1962). "A dynamic programming approach to sequencing problems." In: *Journal of the Society for Industrial and Applied Mathematics* 10.1, pp. 196–210.

– (1971). "The traveling-salesman problem and minimum spanning trees: Part II." In: *Mathematical programming* 1.1, pp. 6–25.

Helsgaun, Keld (2017). "An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report." In:

Hesterberg, Timothy Classen (1988). "Advances in importance sampling." PhD thesis. Stanford University.

Heyman, Daniel P and Matthew J Sobel (2004). *Stochastic models in operations research: stochastic optimization*. Vol. 2. Courier Corporation.

Hinton, Geoffrey, Oriol Vinyals, and Jeffrey Dean (2015). "Distilling the Knowledge in a Neural Network." In: *Deep Learning and Representation Learning Workshop at Neural Information Processing Systems (NeurIPS)*. URL: http://arxiv.org/abs/1503.02531.

Hoorn, Jelke J. van (2016). "Dynamic Programming for Routing and Scheduling. Optimizing Sequences of Decisions." PhD thesis. VU University Amsterdam. ISBN: 978-94-6332-008-5.

Hopfield, John J and David W Tank (1985). ""Neural" computation of decisions in optimization problems." In: *Biological cybernetics* 52.3, pp. 141–152.

Horvitz, Daniel G and Donovan J Thompson (1952). "A generalization of sampling without replacement from a finite universe." In: *Journal of the American Statistical Association* 47.260, pp. 663–685.

Hottung, André, Bhanu Bhandari, and Kevin Tierney (2021). "Learning a Latent Search Space for Routing Problems using Variational Autoencoders." In: *International Conference on Learning Representations (ICML)*.

Hottung, André and Kevin Tierney (2020). "Neural large neighborhood search for the capacitated vehicle routing problem." In: *European Conference on Artificial Intelligence (ECAI)*.

Hromkovič, Juraj (2013). *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics*. Springer Science & Business Media.

Huijben, Iris AM, Wouter Kool, Max B Paulus, and Ruud JG van Sloun (2022). "A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning." In: *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*. To appear.

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *International Conference on Machine Learning (ICML)*, pp. 448–456.

Jang, Eric, Shixiang Gu, and Ben Poole (2016). "Categorical reparameterization with Gumbel-softmax." In: *International Conference on Learning Representations (ICLR)*.

Joshi, Chaitanya K, Thomas Laurent, and Xavier Bresson (2019a). "An efficient graph convolutional network technique for the travelling salesman problem." In: *INFORMS Annual Meeting*.

– (2019b). "On learning paradigms for the travelling salesman problem." In: *Graph Representation Learning Workshop at Neural Information Processing Systems (NeurIPS)*.

Kaempfer, Yoav and Lior Wolf (2018). "Learning the Multiple Traveling Salesmen Problem with Permutation Invariant Pooling Networks." In: *arXiv preprint arXiv:1803.09621*.

Kahn, Arthur B (1962). "Topological sorting of large networks." In: *Communications of the ACM* 5.11, pp. 558–562.

Kant, Goos (2019). *Top 10 Reasons to Optimize Transportation Planning*. Tech. rep. URL: https://ortec.com/assets/2019-05/ORTEC%5C%20White%5C%20paper%5C%20Transportation%5C%20Planning%5C%20Optimization.pdf.

Khalil, Elias, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina (2016). "Learning to branch in mixed integer programming." In: *AAAI Conference on Artificial Intelligence (AAAI)*. Vol. 30. 1.

Kim, Carolyn, Ashish Sabharwal, and Stefano Ermon (2016). "Exact sampling with integer linear programs and random perturbations." In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Kim, Minsu, Jinkyoo Park, and Joungho Kim (2021). "Learning Collaborative Policies to Solve NP-hard Routing Problems." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Kingma, Diederik P and Jimmy Ba (2015). "Adam: A method for stochastic optimization." In: *International Conference on Learning Representations (ICLR)*.

Kingma, Diederik P and Max Welling (2014). "Auto-encoding variational Bayes." In: *International Conference on Learning Representations (ICLR)*.

Kipf, Thomas N and Max Welling (2017). "Semi-supervised classification with graph convolutional networks." In: *International Conference on Learning Representations (ICLR)*.

Kobeaga, Gorka, Maria Merino, and Jose A Lozano (2018). "An efficient evolutionary algorithm for the orienteering problem." In: *Computers & Operations Research* 90, pp. 42–59.

Kok, AL, Elias W Hans, Johannes MJ Schutten, and Willem HM Zijm (2010). "A dynamic programming heuristic for vehicle routing with time-dependent travel times and required breaks." In: *Flexible services and manufacturing journal* 22.1-2, pp. 83–108.

Kool, Wouter, Herke van Hoof, Joaquim Gromicho, and Max Welling (2022a). "Deep Policy Dynamic Programming for Vehicle Routing Problems." In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*. First published in Lecture Notes in Computer Science, vol 13292 by Springer Nature. Reproduced with permission from Springer Nature.

Kool, Wouter, Herke van Hoof, and Max Welling (2019a). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations (ICLR)*.

– (2019b). "Buy 4 REINFORCE Samples, Get a Baseline for Free!" In: *Deep Reinforcement Learning Meets Structured Prediction Workshop at the International Conference on Learning Representations (ICLR)*.

– (2019c). "Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement." In: *International Conference on Machine Learning (ICML)*, pp. 3499–3508.

– (2020a). "Ancestral Gumbel-Top-k Sampling for Sampling Without Replacement." In: *Journal of Machine Learning Research (JMLR)* 21, pp. 1–36.

– (2020b). "Estimating Gradients for Discrete Random Variables by Sampling without Replacement." In: *International Conference on Learning Representations (ICLR)*.

Kool, Wouter, Joep Olde Juninck, Ernst Roos, Kamiel Cornelissen, Pim Agterberg, Jelke van Hoorn, and Thomas Visser (2022b). "Hybrid Genetic Search for the Vehicle Routing Problem with Time Windows: a High-Performance Implementation." In: *12th DIMACS Implementation Challenge Workshop*.

Kool, Wouter, Chris J. Maddison, and Andriy Mnih (2021). "Unbiased Gradient Estimation with Balanced Assignments for Mixtures of Experts." In: *I (Still) Can't Believe It's Not Better! Workshop at Neural Information Processing Systems (NeurIPS)*.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 25, pp. 1097–1105.

Kwon, Yeong-Dae, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min (2020). "POMO: Policy Optimization with Multiple Optima for Reinforcement Learning." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Laporte, Gilbert (1992). "The vehicle routing problem: An overview of exact and approximate algorithms." In: *European Journal of Operational Research (EJOR)* 59.3, pp. 345–358.

Larochelle, Hugo and Iain Murray (2011). "The neural autoregressive distribution estimator." In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 29–37.

Lawler, Eugene L and David E Wood (1966). "Branch-and-bound methods: A survey." In: *Operations research* 14.4, pp. 699–719.

Leblond, Rémi, Jean-Baptiste Alayrac, Anton Osokin, and Simon Lacoste-Julien (2018). "SeaRNN: Training RNNs with Global-Local Losses." In: *International Conference on Learning Representations (ICLR)*.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning." In: *Nature* 521.7553, pp. 436–444.

Lee, Juho, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh (2019). "Set transformer: A framework for attention-based permutation-invariant neural networks." In: *International Conference on Machine Learning (ICML)*. PMLR, pp. 3744–3753.

Lehmann, EL and Henry Scheffé (1950). "Completeness, Similar Regions, and Unbiased Estimation: Part I." In: *Sankhyā: The Indian Journal of Statistics*, pp. 305–340.

Lenstra, Jan Karel and AHG Rinnooy Kan (1981). "Complexity of vehicle routing and scheduling problems." In: *Networks* 11.2, pp. 221–227.

Li, Jiwei, Will Monroe, and Dan Jurafsky (2016). "A simple, fast diverse decoding algorithm for neural generation." In: *arXiv preprint arXiv:1611.08562*.

Li, Ke and Jitendra Malik (2016). "Learning to optimize." In: *arXiv preprint arXiv:1606.01885*.

Li, Sirui, Zhongxia Yan, and Cathy Wu (2021). "Learning to delegate for large-scale vehicle routing." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Li, Zhuwen, Qifeng Chen, and Vladlen Koltun (2018). "Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search." In: *Advances in Neural Information Processing Systems (NeurIPS)*, p. 539.

Liang, Chen, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao (2018). "Memory augmented policy optimization for program synthesis and semantic parsing." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9994–10006.

Liu, Runjing, Jeffrey Regier, Nilesh Tripuraneni, Michael Jordan, and Jon Mcauliffe (2019). "Rao-Blackwellized Stochastic Gradients for Discrete Distributions." In: *International Conference on Machine Learning (ICML)*, pp. 4023–4031.

Lorberbom, Guy, Chris J Maddison, Nicolas Heess, Tamir Hazan, and Daniel Tarlow (2020). "Direct Policy Gradients: Direct Optimization of Policies in Discrete Action Spaces." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 18076–18086.

Lu, Hao, Xingwen Zhang, and Shuang Yang (2020). "A learning-based iterative method for solving vehicle routing problems." In: *International Conference on Learning Representations*.

Luce, Robert Duncan (1959). "Individual choice behavior." In:

Luo, Ruotian (2020). "A better variant of self-critical sequence training." In: *arXiv preprint arXiv:2003.09971*.

Ma, Qiang, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori (2020). "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning." In: *AAAI International Workshop on Deep Learning on Graphs: Methodologies and Applications (DLGMA)*.

Ma, Yining, Jingwen Li, Zhiguang Cao, Wen Song, Le Zhang, Zhenghua Chen, and Jing Tang (2021). "Learning to Iteratively Solve Routing Problems with Dual-Aspect Collaborative Transformer." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Mächler, Martin (2012). *Accurately Computing* $\log(1 - \exp(-|a|))$ *Assessed by the Rmpfr package*. URL: https://cran.r-project.org/web/packages/Rmpfr/vignettes/log1mexp-note.pdf.

Maddison, Chris J, Andriy Mnih, and Yee Whye Teh (2016). "The concrete distribution: A continuous relaxation of discrete random variables." In: *International Conference on Learning Representations (ICLR)*.

Maddison, Chris J, Daniel Tarlow, and Tom Minka (2014). "A* sampling." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 3086–3094.

Malandraki, Chryssi and Robert B Dial (1996). "A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem." In: *European Journal of Operational Research (EJOR)* 90.1, pp. 45–55.

Marti, Rafael, Panos M. Pardalos, and Mauricio G. C. Resende, eds. (2018). *Handbook of Heuristics*. Springer.

Mazyavkina, Nina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev (2020). "Reinforcement learning for combinatorial optimization: A survey." In: *arXiv preprint arXiv:2003.03600*.

Michalewicz, Zbigniew and David B Fogel (2013). *How to solve it: modern heuristics*. Springer Science & Business Media.

Mingozzi, Aristide, Lucio Bianco, and Salvatore Ricciardelli (1997). "Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints." In: *Operations Research* 45.3, pp. 365–377.

Mnih, Andriy and Karol Gregor (2014). "Neural Variational Inference and Learning in Belief Networks." In: *International Conference on Machine Learning (ICML)*, pp. 1791–1799.

Mnih, Andriy and Danilo Rezende (2016). "Variational Inference for Monte Carlo Objectives." In: *International Conference on Machine Learning (ICML)*, pp. 2188–2196.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning." In: *Nature* 518.7540, p. 529.

Murthy, MN (1957). "Ordered and unordered estimators in sampling without replacement." In: *Sankhyā: The Indian Journal of Statistics* 18.3/4, pp. 379–390.

Nair, Vinod, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. (2020). "Solving Mixed Integer Programs Using Neural Networks." In: *arXiv preprint arXiv:2012.13349*.

Nazari, MohammadReza, Afshin Oroojlooy, Lawrence Snyder, and Martin Takac (2018). "Reinforcement Learning for Solving the Vehicle Routing Problem." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9860–9870.

Negrinho, Renato, Matthew Gormley, and Geoffrey J Gordon (2018). "Learning Beam Search Policies via Imitation Learning." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 10673–10682.

Norouzi, Mohammad, Samy Bengio, Navdeep Jaitly, Mike Schuster, Yonghui Wu, Dale Schuurmans, et al. (2016). "Reward augmented maximum likelihood for neural structured prediction." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1723–1731.

Novoa, Clara and Robert Storer (2009). "An approximate dynamic programming approach for the vehicle routing problem with stochastic demands." In: *European Journal of Operational Research (EJOR)* 196.2, pp. 509–515.

Nowak, Alex, Soledad Villar, Afonso S Bandeira, and Joan Bruna (2017). "A note on learning algorithms for quadratic assignment with graph neural networks." In: *Principled Approaches to Deep Learning Workshop at the International Conference on Machine Learning (ICML)*.

Ott, Myle, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli (2019). "fairseq: A Fast, Extensible Toolkit for Sequence Modeling." In: *Proceedings of NAACL-HLT 2019: Demonstrations*.

Paisley, John, David M Blei, and Michael I Jordan (2012). "Variational Bayesian inference with stochastic search." In: *International Conference on Machine Learning (ICML)*, pp. 1363–1370.

Papadimitriou, Christos H and Kenneth Steiglitz (1998). *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

Papandreou, George and Alan L Yuille (2011). "Perturb-and-map random fields: Using discrete optimization to learn and sample from energy models." In: *International Conference on Computer Vision (ICCV)*, pp. 193–200.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu (2002). "BLEU: a method for automatic evaluation of machine translation." In: *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 311–318.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. (2019). "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in Neural Information Processing Systems (NeurIPS)* 32, pp. 8026–8037.

Peng, Bo, Jiahai Wang, and Zizhen Zhang (2019). "A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems." In: *International Symposium on Intelligence Computation and Applications*. Springer, pp. 636–650.

Perron, Laurent and Vincent Furnon (2022). *OR-Tools*. Google. URL: https://developers.google.com/optimization/.

Pessoa, Artur, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck (2020). "A generic exact solver for vehicle routing and related problems." In: *Mathematical Programming* 183.1, pp. 483–523.

Pinedo, Michael (2012). *Scheduling*. Vol. 29. Springer.

Plackett, Robin L (1975). "The analysis of permutations." In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 24.2, pp. 193–202.

Prouvost, Antoine, Justin Dumouchelle, Maxime Gasse, Didier Chételat, and Andrea Lodi (2021). "Ecole: A Library for Learning Inside MILP Solvers." In: *arXiv preprint arXiv:2104.02828*.

Provost, Foster and Tom Fawcett (2013). "Data science and its relationship to big data and data-driven decision making." In: *Big data* 1.1, pp. 51–59.

Queiroga, Eduardo, Ruslan Sadykov, Eduardo Uchoa, and Thibaut Vidal (2022). "10,000 optimal CVRP solutions for testing machine learning based heuristics." In: *AAAI Workshop on Machine Learning for Operations Research (ML4OR)*.

Raj, Des (1956). "Some estimators in sampling with varying probabilities without replacement." In: *Journal of the American Statistical Association* 51.274, pp. 269–284.

Ranganath, Rajesh, Sean Gerrish, and David Blei (2014). "Black box variational inference." In: *Artificial Intelligence and Statistics*, pp. 814–822.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba (2016). "Sequence level training with recurrent neural networks." In: *International Conference on Learning Representations (ICLR)*.

Rennie, Steven J, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel (2017). "Self-Critical Sequence Training for Image Captioning." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7008–7024.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). "Stochastic Backpropagation and Approximate Inference in Deep Generative Models." In: *International Conference on Machine Learning (ICML)*, pp. 1278–1286.

Roeder, Geoffrey, Yuhuai Wu, and David K Duvenaud (2017). "Sticking the landing: Simple, lower-variance gradient estimators for variational inference." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 6925–6934.

Ropke, Stefan and David Pisinger (2006). "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows." In: *Transportation Science* 40.4, pp. 455–472.

Rosenkrantz, Daniel J, Richard E Stearns, and Philip M Lewis (2009). "An analysis of several heuristics for the traveling salesman problem." In: *Fundamental Problems in Computing*. Springer, pp. 45–69.

Salakhutdinov, Ruslan and Iain Murray (2008). "On the quantitative analysis of deep belief networks." In: *International Conference on Machine Learning (ICML)*, pp. 872–879.

Schrijver, Alexander (2003). *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. Springer Science & Business Media.

Schrimpf, Gerhard, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck (2000). "Record breaking optimization results using the ruin and recreate principle." In: *Journal of Computational Physics* 159.2, pp. 139–171.

Schulman, John, Nicolas Heess, Theophane Weber, and Pieter Abbeel (2015). "Gradient estimation using stochastic computation graphs." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 3528–3536.

Shao, Yuanlong, Stephan Gouws, Denny Britz, Anna Goldie, Brian Strope, and Ray Kurzweil (2017). "Generating High-Quality and Informative Conversation Responses with Sequence-to-Sequence Models." In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2210–2219.

Shen, Shiqi, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu (2016). "Minimum Risk Training for Neural Machine Translation." In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. Vol. 1, pp. 1683–1692.

Shi, Kensen, David Bieber, and Charles Sutton (2020). "Incremental sampling without replacement for sequence models." In: *International Conference on Machine Learning (ICML)*, pp. 8785–8795.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search." In: *nature* 529.7587, pp. 484–489.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." In: *Science* 362.6419, pp. 1140–1144.

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. (2017). "Mastering the game of go without human knowledge." In: *Nature* 550.7676, p. 354.

Smith, Kate A (1999). "Neural networks for combinatorial optimization: a review of more than a decade of research." In: *INFORMS Journal on Computing* 11.1, pp. 15–34.

Stöttner, Timo (2019). *Why data should be normalized before training a neural network*. Blog. URL: https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d.

Sun, Yuan, Andreas Ernst, Xiaodong Li, and Jake Weiner (2020). "Generalization of machine learning for problem reduction: a case study on travelling salesman problems." In: *OR Spectrum*, pp. 1–27.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 3104–3112.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Taha, Hamdy A (2011). *Operations research: an introduction*. Vol. 790. Pearson/Prentice Hall Upper Saddle River, NJ, USA.

Tarlow, Daniel, Ryan Adams, and Richard Zemel (2012). "Randomized optimum models for structured prediction." In: *Artificial Intelligence and Statistics*, pp. 1221–1229.

Ting, Daniel (2017). "Adaptive threshold sampling and estimation." In: *arXiv preprint arXiv:1708.04970*.

Titsias, Michalis K and Miguel Lázaro-Gredilla (2015). "Local expectation gradients for black box variational inference." In: *Advances in Neural Information Processing Systems-Volume 2*, pp. 2638–2646.

Toth, Paolo and Daniele Vigo (2014). *Vehicle routing: problems, methods, and applications*. SIAM.

Tsiligirides, Theodore (1984). "Heuristic methods applied to orienteering." In: *Journal of the Operational Research Society* 35.9, pp. 797–809.

Tucker, George, Andriy Mnih, Chris J Maddison, John Lawson, and Jascha Sohl-Dickstein (2017). "Rebar: Low-variance, unbiased gradient estimates for discrete

latent variable models." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2627–2636.

Uchoa, Eduardo, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian (2017). "New benchmark instances for the capacitated vehicle routing problem." In: *European Journal of Operational Research (EJOR)* 257.3, pp. 845–858.

Vansteenwegen, Pieter, Wouter Souffriau, and Dirk Van Oudheusden (2011). "The orienteering problem: A survey." In: *European Journal of Operational Research (EJOR)* 209.1, pp. 1–10.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5998–6008.

Velickovic, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio (2018). "Graph Attention Networks." In: *International Conference on Learning Representations (ICLR)*.

Vesselinova, Natalia, Rebecca Steinert, Daniel F Perez-Ramirez, and Magnus Boman (2020). "Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking." In: *IEEE Access* 8, pp. 120388–120416.

Vidal, Thibaut (2022). "Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood." In: *Computers & Operations Research* 140, p. 105643.

Vidal, Thibaut, Teodor Gabriel Crainic, Michel Gendreau, Nadia Lahrichi, and Walter Rei (2012). "A hybrid genetic algorithm for multidepot and periodic vehicle routing problems." In: *Operations Research* 60.3, pp. 611–624.

Vieira, Tim (2014). *Gumbel-max trick and weighted reservoir sampling*. Blog. URL: https://timvieira.github.io/blog/post/2014/08/01/gumbel-max-trick-and-weighted-reservoir-sampling/.

– (2017). *Estimating means in a finite universe*. Blog. URL: https://timvieira.github.io/blog/post/2017/07/03/estimating-means-in-a-finite-universe/.

Vijayakumar, Ashwin K, Michael Cogswell, Ramprasaath R Selvaraju, Qing Sun, Stefan Lee, David J Crandall, and Dhruv Batra (2018). "Diverse Beam Search for Improved Description of Complex Scenes." In: *AAAI Conference on Artificial Intelligence (AAAI)*.

Vinyals, Oriol, Samy Bengio, and Manjunath Kudlur (2016). "Order matters: Sequence to sequence for sets." In: *International Conference on Learning Representations (ICLR)*.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015a). "Pointer networks." In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 2692–2700.

Vinyals, Oriol, Alexander Toshev, Samy Bengio, and Dumitru Erhan (2015b). "Show and tell: A neural image caption generator." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3156–3164.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Machine learning* 8.3-4, pp. 229–256.

Wiseman, Sam and Alexander M Rush (2016). "Sequence-to-Sequence Learning as Beam-Search Optimization." In: *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1296–1306.

Woeginger, Gerard (2016). *The P-versus-NP page*. URL: https://www.win.tue.nl/~gwoegi/P-versus-NP.htm.

Wolpert, David H and William G Macready (1997). "No free lunch theorems for optimization." In: *IEEE Transactions on Evolutionary Computation* 1.1, pp. 67–82.

Wolsey, Laurence A and George L Nemhauser (1999). *Integer and combinatorial optimization*. Vol. 55. John Wiley & Sons.

Wong, Chak-Kuen and Malcolm C. Easton (1980). "An efficient method for weighted sampling without replacement." In: *SIAM Journal on Computing* 9.1, pp. 111–113.

Wu, Yaoxin, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim (2021). "Learning improvement heuristics for solving routing problems." In: *IEEE Transactions on Neural Networks and Learning Systems*.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. (2016). "Google's neural machine translation system: Bridging the gap between human and machine translation." In: *arXiv preprint arXiv:1609.08144*.

Xin, Liang, Wen Song, Zhiguang Cao, and Jie Zhang (2020). "Step-wise Deep Learning Models for Solving Routing Problems." In: *IEEE Transactions on Industrial Informatics*.

– (2021). "NeuroLKH: Combining Deep Learning Model with Lin-Kernighan-Helsgaun Heuristic for Solving the Traveling Salesman Problem." In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Xu, Shenghe, Shivendra S Panwar, Murali Kodialam, and TV Lakshman (2020). "Deep Neural Network Approximated Dynamic Programming for Combinatorial Optimization." In: *AAAI Conference on Artificial Intelligence (AAAI)*. Vol. 34. 02, pp. 1684–1691.

Yang, Feidiao, Tiancheng Jin, Tie-Yan Liu, Xiaoming Sun, and Jialin Zhang (2018). "Boosting dynamic programming with neural networks for solving np-hard problems." In: *Asian Conference on Machine Learning (ACML)*. PMLR, pp. 726–739.

Yellott, John I (1977). "The relationship between Luce's choice axiom, Thurstone's theory of comparative judgment, and the double exponential distribution." In: *Journal of Mathematical Psychology* 15.2, pp. 109–144.

Yin, Mingzhang, Yuguang Yue, and Mingyuan Zhou (2019). "ARSM: Augment-REINFORCE-Swap-Merge Estimator for Gradient Backpropagation Through Categorical Variables." In: *International Conference on Machine Learning (ICML)*, pp. 7095–7104.

APPENDIX

A ATTENTION, LEARN TO SOLVE ROUTING PROBLEMS!

A.1 ATTENTION MODEL DETAILS

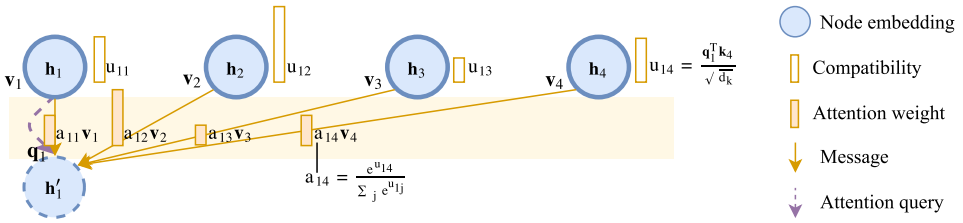


Figure 27: Illustration of weighted message passing using a dot-attention mechanism. Only computation of messages received by node 1 are shown for clarity. Best viewed in color.

ATTENTION MECHANISM We interpret the attention mechanism by Vaswani et al. (2017) as a weighted message passing algorithm between nodes in a graph. The weight of the message *value* that a node receives from a neighbor depends on the *compatibility* of its *query* with the *key* of the neighbor, as illustrated in Figure 27. Formally, we define dimensions d_k and d_v and compute the key $\mathbf{k}_i \in \mathbb{R}^{d_\mathrm{k}}$, value $\mathbf{v}_i \in \mathbb{R}^{d_\mathrm{v}}$ and query $\mathbf{q}_i \in \mathbb{R}^{d_\mathrm{k}}$ for each node by projecting the embedding $\mathbf{h}_i$:

$$\mathbf{q}_i = W^Q \mathbf{h}_i, \quad \mathbf{k}_i = W^K \mathbf{h}_i, \quad \mathbf{v}_i = W^V \mathbf{h}_i. \tag{70}$$

Here parameters W^Q and W^K are $(d_\mathrm{k} \times d_\mathrm{h})$ matrices and W^V has size $(d_\mathrm{v} \times d_\mathrm{h})$. From the queries and keys, we compute the compatibility $u_{ij} \in \mathbb{R}$ of the query $\mathbf{q}_i$ of node i with the key $\mathbf{k}_j$ of node j as the (scaled, see Vaswani et al. (2017)) dot-product:

$$u_{ij} = \begin{cases} \frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_\mathrm{k}}} & \text{if } i \text{ adjacent to } j \\ -\infty & \text{otherwise.} \end{cases} \tag{71}$$

In a general graph, defining the compatibility of non-adjacent nodes as $-\infty$ prevents message passing between these nodes. From the compatibilities u_{ij}, we compute the *attention weights* $a_{ij} \in [0, 1]$ using a softmax:

$$a_{ij} = \frac{e^{u_{ij}}}{\sum_{j'} e^{u_{ij'}}}. \tag{72}$$

Finally, the vector $\mathbf{h}'_i$ that is received by node i is the convex combination of messages $\mathbf{v}_j$:

$$\mathbf{h}'_i = \sum_j a_{ij}\mathbf{v}_j. \tag{73}$$

MULTI-HEAD ATTENTION As was noted by Vaswani et al. (2017) and Velickovic et al. (2018), it is beneficial to have multiple attention heads. This allows nodes to receive different types of messages from different neighbors. Especially, we compute the value in equation 73 $M = 8$ times with different parameters, using $d_\text{k} = d_\text{v} = \frac{d_\text{h}}{M} = 16$. We denote the result vectors by $\mathbf{h}'_{im}$ for $m \in 1, \dots, M$. These are projected back to a single d_h-dimensional vector using $(d_\text{h} \times d_\text{v})$ parameter matrices W^O_m. The final multi-head attention value for node i is a function of $\mathbf{h}_1, \dots, \mathbf{h}_n$ through $\mathbf{h}'_{im}$:

$$\text{MHA}_i(\mathbf{h}_1, \dots, \mathbf{h}_n) = \sum_{m=1}^{M} W^O_m \mathbf{h}'_{im}. \tag{74}$$

FEED-FORWARD SUBLAYER The feed-forward sublayer computes node-wise projections using a hidden (sub)sublayer with dimension $d_\text{ff} = 512$ and a ReLu activation:

$$\text{FF}(\hat{\mathbf{h}}_i) = W^{\text{ff},1} \cdot \text{ReLu}(W^{\text{ff},0}\hat{\mathbf{h}}_i + \boldsymbol{b}^{\text{ff},0}) + \boldsymbol{b}^{\text{ff},1}. \tag{75}$$

BATCH NORMALIZATION We use batch normalization with learnable d_h-dimensional affine parameters $\boldsymbol{w}^\text{bn}$ and $\boldsymbol{b}^\text{bn}$:

$$\text{BN}(\mathbf{h}_i) = \boldsymbol{w}^\text{bn} \odot \overline{\text{BN}}(\mathbf{h}_i) + \boldsymbol{b}^\text{bn}. \tag{76}$$

Here $\odot$ denotes the element-wise product and $\overline{\text{BN}}$ refers to batch normalization without affine transformation.

A.2 TRAVELLING SALESMAN PROBLEM

A.2.1 Critic architecture

The critic network architecture uses 3 attention layers similar to our encoder, after which the node embeddings are averaged and processed by an MLP with one hidden layer with 128 neurons and ReLu activation and a single output. We used the same learning rate as for the AM/PN in all experiments.

A.2.2 Instance generation

For all TSP instances, the n node locations are sampled uniformly at random in the unit square. This distribution is chosen to be neither easy nor artificially hard and to be able to compare to other learned heuristics.

A.2.3 Details of baselines

This section describes details of the heuristics implemented for the TSP. All of the heuristics construct a single tour in a single pass, by extending a partial solution one node at the time.

NEAREST NEIGHBOR The nearest neighbor heuristic represents the partial solution as a *path* with a *start* and *end* node. The initial path is formed by a single node, selected randomly, which becomes the start node but also the end node of the initial path. In each iteration, the next node is selected as the node nearest to the end node of the partial path. This node is added to the path and becomes the new end node. Finally, after all nodes are added this way, the end node is connected with the start node to form a tour. In our implementation, for deterministic results we always start with the first node in the input, which can be considered random as the instances are generated randomly.

FARTHEST/NEAREST/RANDOM INSERTION The insertion heuristics represent a partial solution as a *tour*, and extends it by *inserting* nodes one node at the time. In our implementation, we always insert the node using the *cheapest* insertion cost. This means that when node i is inserted, the place of insertion (between adjacent nodes j and k in the tour) is selected such that it minimizes the *insertion costs* $d_{ji} + d_{ik} - d_{jk}$, where d_{ji}, d_{ik} and d_{jk} represent the distances from node j to i, i to k and j to k, respectively.

The different variants of the insertion heuristic vary in the way in which the node which is inserted is selected. Let S be the set of nodes in the partial tour. *Nearest* insertion inserts the node i that is nearest to (any node in) the tour:

$$i^* = \arg\min_{i \notin S} \min_{j \in S} d_{ij}. \tag{77}$$

Farthest insertion inserts the node i such that the distance to the tour (i.e. the distance from i to the nearest node j in the tour) is maximized:

$$i^* = \arg\max_{i \notin S} \min_{j \in S} d_{ij}. \tag{78}$$

Random insertion inserts a random node. Similar to nearest neighbor, we consider the input order random so we simply insert the nodes in this order.

A.2.4 Comparison to concurrent work

Independently of our work, Deudon et al. (2018) also developed a model for TSP based on the Transformer (Vaswani et al., 2017). There are important differences to our approach:

- As 'context' for the decoder, Deudon et al. (2018) use the embeddings of the last $K = 3$ visited nodes. We use only the last (e.g. $K = 1$) node but add the *first* visited node (as well as the graph embedding), since the first node is important (it is the destination) while the order of the other nodes is irrelevant as we explain in Section 3.3.
- Deudon et al. (2018) use a critic as baseline (which also uses the Transformer architecture). We also experiment with using a critic (based on the Transformer architecture), but found that using a rollout baseline is much more effective (see Section 3.5).
- Deudon et al. (2018) report results with sampling 128 solutions, with and without 2OPT local search. We report results without 2OPT, using either a single greedy solution or sampling 1280 solutions and additionally show how this directly improves performance compared to Bello et al. (2016).
- By adding 2OPT on top of the best sampled solution, Deudon et al. (2018) show that the model does not produce a local optimum and results can improve by using a 'hybrid' approach of a learned algorithm with local search. This is a nice example of combining learned and traditional heuristics, but it is not compared against using the pointer network (Bello et al., 2016) with 2OPT.
- The model of Deudon et al. (2018) uses a higher dimensionality internally in the decoder (for details see their paper). Training is done with 20000 steps with a batch size of 256.
- Deudon et al. (2018) apply principal component analysis (PCA) on the input coordinates to eliminate rotation symmetry whereas we directly input node coordinates.
- Additionally to TSP, we also consider two variants of VRP, the OP with different prize distributions and the (stochastic) PCTSP.

We want to emphasize that this is independent work, but for completeness we include a full emperical comparison of performance. Since the results presented in the paper by Deudon et al. (2018) are not directly comparable, we ran their code[1] and report results under the same circumstances: using greedy decoding and sampling 1280 solutions on our test dataset (which has exactly the same generative procedure, e.g. uniform in the unit square). Additionally, we include results of their model with 2OPT, showing that (even without 2OPT) final performance of our model is better. We use the hyperparameters in their code, but increase the

1 https://github.com/MichelDeudon/encode-attend-navigate

Table 6: Epoch durations and results and with different seeds and learning rate schedules for TSP.

	EPOCH TIME	$\eta = 10^{-4}$ SEED = 1234	$\eta = 10^{-4}$ SEED = 1235	$\eta = 10^{-3} \times 0.96^{\text{EPOCH}}$ SEED = 1234	$\eta = 10^{-3} \times 0.96^{\text{EPOCH}}$ SEED = 1235
TSP20	5:30	3.85 (0.34%)	3.85 (0.29%)	3.85 (0.33%)	3.85 (0.32%)
TSP50	16:20	5.80 (1.76%)	5.79 (1.66%)	5.81 (2.02%)	5.81 (2.00%)
TSP100 (2GPUs)	27:30	8.12 (4.53%)	8.10 (4.34%)	-	-
N = 0	3:10	4.24 (10.50%)	4.26 (10.95%)	4.25 (10.79%)	4.24 (10.55%)
N = 1	3:50	3.87 (0.97%)	3.87 (1.01%)	3.87 (0.90%)	3.87 (0.89%)
N = 2	5:00	3.85 (0.40%)	3.85 (0.44%)	3.85 (0.38%)	3.85 (0.39%)
N = 3	5:30	3.85 (0.34%)	3.85 (0.29%)	3.85 (0.33%)	3.85 (0.32%)
N = 5	7:00	3.85 (0.25%)	3.85 (0.28%)	3.85 (0.30%)	10.43 (171.82%)
N = 8	10:10	3.85 (0.28%)	3.85 (0.33%)	10.43 (171.82%)	10.43 (171.82%)
AM / EXPONENTIAL	4:20	3.87 (0.95%)	3.87 (0.93%)	3.87 (0.90%)	3.87 (0.87%)
AM / CRITIC	6:10	3.87 (0.96%)	3.87 (0.97%)	3.87 (0.88%)	3.87 (0.88%)
AM / ROLLOUT	5:30	3.85 (0.34%)	3.85 (0.29%)	3.85 (0.33%)	3.85 (0.32%)
PN / EXPONENTIAL	5:10	3.95 (2.94%)	3.94 (2.80%)	3.92 (2.09%)	3.93 (2.37%)
PN / CRITIC	7:30	3.95 (3.00%)	3.95 (2.93%)	3.91 (2.01%)	3.94 (2.84%)
PN / ROLLOUT	6:40	3.93 (2.46%)	3.93 (2.36%)	3.90 (1.63%)	3.90 (1.78%)

batch size to 512 and number of training steps to $100 \times 2500 = 250000$ for a fair comparison (this increased the performance of their model). As training with $n = 100$ gave out-of-memory errors, we train only on $n = 20$ and $n = 50$ and (following Deudon et al. (2018)) report results for $n = 100$ using the model trained for $n = 50$. The training time as well as test run times are comparable.

A.2.5 Extended results

HYPERPARAMETERS We found in general that using a larger learning rate of 10^{-3} works better with decay but may be unstable in some cases. A smaller learning rate 10^{-4} is more stable and does not require decay. This is illustrated in Figure 29, which shows validation results over time using both 10^{-3} and 10^{-4} with and without decay for TSP20 and TSP50 (2 seeds). As can be seen, without decay the method has not yet fully converged after 100 epochs and results may improve even further with longer training.

Table 6 shows the results in absolute terms as well as the relative *optimality gap* compared to Gurobi, for all runs using seeds 1234 and 1235 with the two different learning rate schedules. We did not run final experiments for $n = 100$ with the larger learning rate as we found training with the smaller learning rate to be more stable. It can be seen that in most cases the end results with different learning rate schedules are similar, except for the larger models ($N = 5$, $N = 8$) where some of the runs diverged using the larger learning rate. Experiments with different number of layers N show that $N = 3$ and $N = 5$ achieve best performance, and we find $N = 3$ is a good trade-off between quality of the results and computational complexity (runtime) of the model.

GENERALIZATION We test generalization performance on different n than trained for, which we plot in Figure 28 in terms of the relative optimality gap compared to

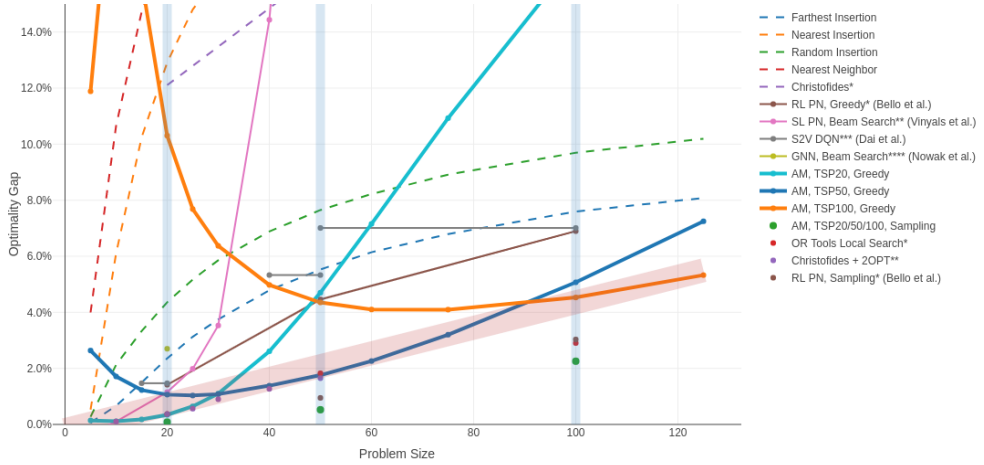
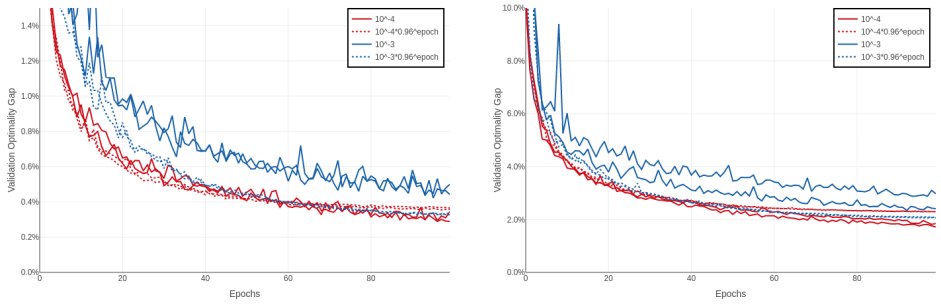


Figure 28: Optimality gap of different methods as a function of problem size $n \in \{5, 10, 15, 20, 25, 30, 40, 50, 60, 75, 100, 125\}$. General baselines are drawn using dashed lines while learned algorithms are drawn with a solid line. Algorithms (general and learned) that perform search or sampling are plotted without connecting lines for clarity. The *, **, *** and **** indicate that values are reported from Bello et al. (2016), Vinyals et al. (2015a), Dai et al. (2017) and Nowak et al. (2017) respectively. Best viewed in color.



(a) TSP20, four schedules for η (2 seeds) **(b)** TSP50, four schedules for η (2 seeds)

Figure 29: Validation set optimality gap as a function of the number of epochs for different η.

Gurobi. The train sizes are indicated with vertical marker bars. The models generalize when tested on different sizes, although quality degrades as the difference becomes bigger, which can be expected as there is *no free lunch* (Wolpert and Macready, 1997). Since the architectures are the same, these differences mean the models learn to specialize on the problem sizes trained for. We can make a strong overall algorithm by selecting the trained model with highest validation performance for each instance size n (marked in Figure 28 by the red bar). For reference, we also include the baselines, where for the methods that perform search or sampling we do not connect the dots to prevent cluttering and to make the distinction with methods that consider only a single solution clear.

A.3 VEHICLE ROUTING PROBLEM

The capacitated vehicle routing problem (CVRP) is a generalization of the TSP in which case there is a depot and multiple routes should be created, each starting and ending at the depot. In our graph based formulation, we add a special depot node with index o and coordinates $\mathbf{x}_0$. A vehicle (route) has capacity $D > 0$ and each (regular) node $i \in \{1, \dots n\}$ has a demand $0 < \delta_i \leq D$. Each route starts and ends at the depot and the total demand in each route should not exceed the capacity, so $\sum_{i \in R_j} \delta_i \leq D$, where R_j is the set of node indices assigned to route j. Without loss of generality, we assume a normalized $\hat{D} = 1$ as we can use normalized demands $\hat{\delta}_i = \frac{\delta_i}{D}$.

The split delivery VRP (SDVRP) is a generalization of CVRP in which every node can be visited multiple times, and only a subset of the demand has to be delivered at each visit. Instances for both CVRP and SDVRP are specified in the same way: an instance with size n as a depot location $\mathbf{x}_0$, n node locations $\mathbf{x}_i, i = 1 \dots n$ and (normalized) demands $0 < \hat{\delta}_i \leq 1, i = 1 \dots n$.

A.3.1 Instance generation

We follow Nazari et al. (2018) in the generation of instances for $n = 20, 50, 100$, but normalize the demands by the capacities. The depot location as well as n node locations are sampled uniformly at random in the unit square. The demands are defined as $\hat{\delta}_i = \frac{\delta_i}{D^n}$ where δ_i is discrete and sampled uniformly from $\{1, \dots, 9\}$ and $D^{20} = 30$, $D^{50} = 40$ and $D^{100} = 50$.

A.3.2 Attention model for the VRP

ENCODER In order to allow our attention model to distinguish the depot node from the regular nodes, we use separate parameters W_0^x and $\mathbf{b}_0^x$ to compute the initial embedding $\mathbf{h}_0^{(0)}$ of the depot node. Additionally, we provide the normalized demand δ_i as input feature (and adjust the size of parameter W^x accordingly):

$$\mathbf{h}_i^{(0)} = \begin{cases} W_0^x \mathbf{x}_i + \mathbf{b}_0^x & i = 0 \\ W^x \left[\mathbf{x}_i, \hat{\delta}_i\right] + \mathbf{b}^x & i = 1, \dots, n. \end{cases} \tag{79}$$

CAPACITY CONSTRAINTS To facilitate the capacity constraints, we keep track of the remaining demands $\hat{\delta}_{i,t}$ for the nodes $i \in \{1, \dots n\}$ and remaining vehicle capacity $\hat{D}_t$ at time t. At $t = 1$, these are initialized as $\hat{\delta}_{i,t} = \hat{\delta}_i$ and $\hat{D}_t = 1$, after

which they are updated as follows (recall that π_t is the index of the node selected at decoding step t):

$$\hat{\delta}_{i,t+1} = \begin{cases} \max(0, \hat{\delta}_{i,t} - \hat{D}_t) & \pi_t = i \\ \hat{\delta}_{i,t} & \pi_t \neq i \end{cases} \tag{80}$$

$$\hat{D}_{t+1} = \begin{cases} \max(\hat{D}_t - \hat{\delta}_{\pi_t,t}, 0) & \pi_t \neq 0 \\ 1 & \pi_t = 0. \end{cases} \tag{81}$$

If we do not allow split deliveries, $\hat{\delta}_{i,t}$ will be either 0 or $\hat{\delta}_i$ for all t.

DECODER CONTEXT The context for the decoder for the VRP at time t is the current/last location π_{t-1} and the remaining capacity $\hat{D}_t$. Compared to TSP, we do not need placeholders if $t = 1$ as the route starts at the depot and we do not need to provide information about the first node as the route should end at the depot:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, \hat{D}_t\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{0}^{(N)}, \hat{D}_t\right] & t = 1. \end{cases} \tag{82}$$

MASKING The depot can be visited multiple times, but we do not allow it to be visited at two subsequent timesteps. Therefore, in both layers of the decoder, we change the masking for the depot $j = 0$ and define $u_{(c)0} = -\infty$ if (and only if) $t = 1$ or $\pi_{t-1} = 0$. The masking for the nodes depends on whether we allow split deliveries. Without split deliveries, we do not allow nodes to be visited if their remaining demand is 0 (if the node was already visited) or exceeds the remaining capacity, so for $j \neq 0$ we define $u_{(c)j} = -\infty$ if (and only if) $\hat{\delta}_{i,t} = 0$ or $\hat{\delta}_{i,t} > \hat{D}_t$. With split deliveries, we only forbid delivery when the remaining demand is 0, so we define $u_{(c)j} = -\infty$ if (and only if) $\hat{\delta}_{i,t} = 0$.

SPLIT DELIVERIES Without split deliveries, the remaining demand $\hat{\delta}_{i,t}$ is either 0 or $\hat{\delta}_i$, corresponding to whether the node has been visited or not, and this information is conveyed to the model via the masking of the nodes already visited. However, when split deliveries are allowed, the remaining demand $\hat{\delta}_{i,t}$ can take any value $0 \leq \hat{\delta}_{i,t} \leq \hat{\delta}_i$. This information cannot be included in the context node as it corresponds to individual nodes. Therefore we include it in the computation of the keys and values in both the attention layer (glimpse) and the output layer of the decoder, such that we compute queries, keys and values using:

$$\mathbf{q}_{(c)} = W^Q \mathbf{h}_{(c)} \quad \mathbf{k}_i = W^K \mathbf{h}_i + W_d^K \hat{\delta}_{i,t}, \quad \mathbf{v}_i = W^V \mathbf{h}_i + W_d^V \hat{\delta}_{i,t}. \tag{83}$$

Here we W_d^K and W_d^V are $(d_k \times 1)$ parameter matrices and we define $\hat{\delta}_{i,t} = 0$ for the depot $i = 0$. Summing the projection of both $\mathbf{h}_i$ and $\hat{\delta}_{i,t}$ is equivalent to projecting the concatenation $[\mathbf{h}_i, \hat{\delta}_{i,t}]$ with a single $((d_h + 1) \times d_k)$ matrix W^K. However, using

this formulation we only need to compute the first term once (instead for every t) and by the weight initialization this puts more importance on $\hat{\delta}_{i,t}$ initially (which is otherwise just 1 of $d_h + 1 = 129$ input values).

TRAINING For the VRP, the length of the output of the model depends on the number of times the depot is visited. In general, the depot is visited multiple times, and in the case of SDVRP also some regular nodes are visited twice. Therefore the length of the solution is larger than n, which requires more memory such that we find it necessary to limit the batch size B to 256 for $n = 100$ (on 2 GPUs). To keep training times tractable and the total number of parameter updates equal, we still process 2500 batches per epoch, for a total of 0.64M training instances per epoch.

A.3.3 Details of baselines

For LKH3[2] by Helsgaun (2017) we build and run their code with the `SPECIAL` parameter as specified in their CVRP runscript[3]. We perform 1 run with a maximum of 10000 trials, as we found performing 10 runs only marginally improves the quality of the results while taking much more time.

A.3.4 Example solutions

Figure 30 shows example solutions for the CVRP with $n = 100$ that were obtained by a single construction using the model with greedy decoding. These visualizations give insight in the heuristic that the model has learned. In general we see that the model constructs the routes from the bottom to the top, starting below the depot. Most routes are densely packed, except for the last route that has to serve some remaining (close to each other) customers. In most cases, the node in the route that is farthest from the depot is somewhere in the middle of the route, such that customers are served on the way to and from the farthest nodes. In some cases, we see that the order of stops within some individual routes is suboptimal, which means that the method will likely benefit from simple further optimizations on top, such as a beam search, a post-processing procedure based on local search (e.g. 2OPT) or solving the individual routes using a TSP solver.

A.4 ORIENTEERING PROBLEM

In the orienteering problem (OP) each node has a prize ρ_i and the goal is to *maximize* the total prize of nodes visited, while keeping the total length of the route below a maximum length T. This problem is different from the TSP and the VRP because

2 http://akira.ruc.dk/~keld/research/LKH-3/
3 run_CVRP in http://akira.ruc.dk/~keld/research/LKH-3/BENCHMARKS/CVRP.tgz

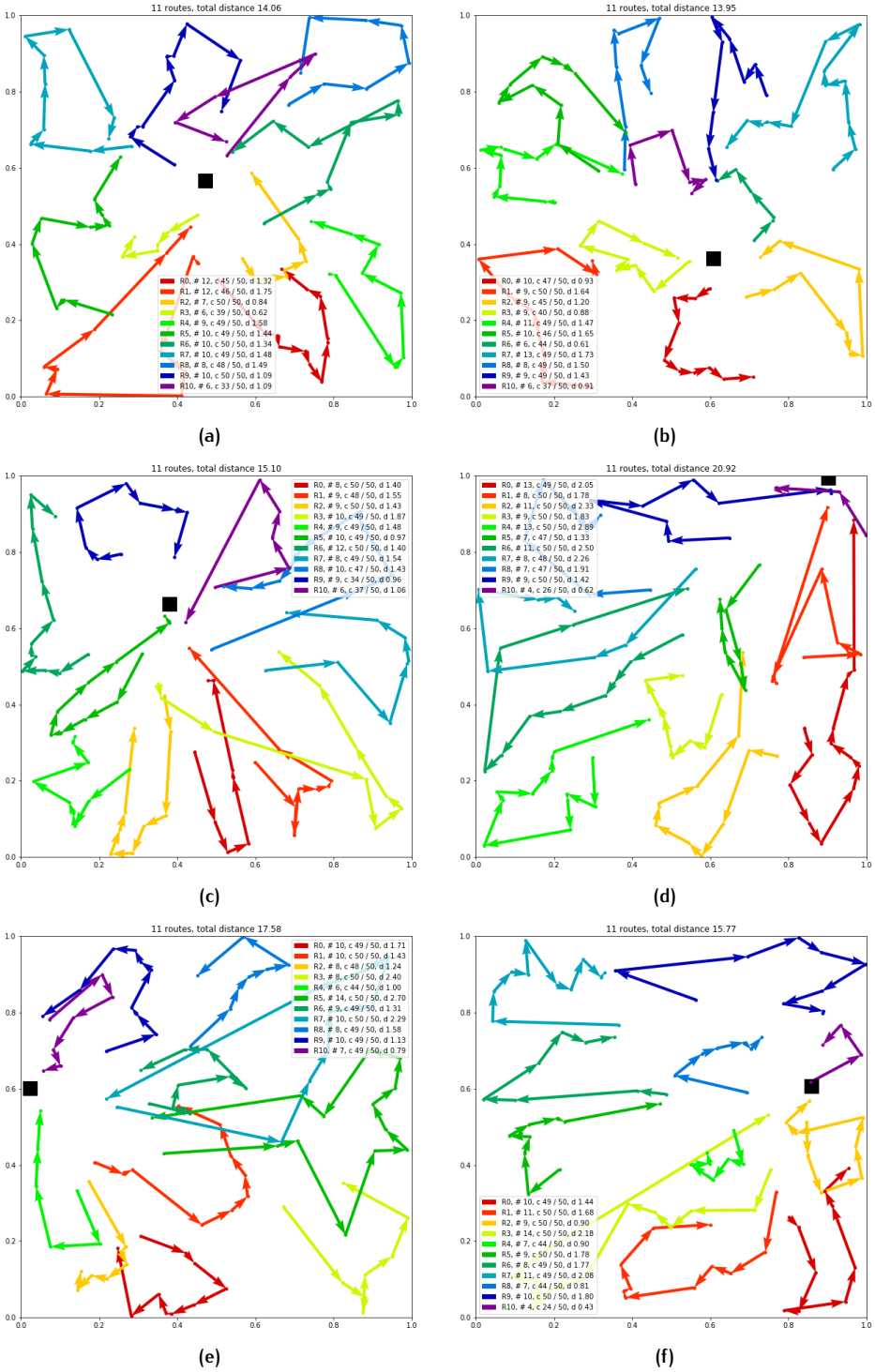


Figure 30: Example greedy solutions for the CVRP ($n = 100$). Edges from and to depot omitted for clarity. Legend order/coloring and arcs indicate the order in which the solution was generated. Legends indicate the number of stops, the used and available capacity and the distance per route.

visiting each node is optional. Similar to the VRP, we add a special depot node with index 0 and coordinates $\mathbf{x}_0$. If the model selects the depot, we consider the route to be finished. In order to prevent infeasible solutions, we only allow to visit a node if after visiting that node a return to the depot is still possible within the maximum length constraint. Note that it is always suboptimal to visit the depot if additional nodes can be visited, but we do not enforce this knowledge.

A.4.1 Instance generation

The depot location as well as n node locations are sampled uniformly at random in the unit square. For the distribution of the prizes, we consider three different variants described by Fischetti et al. (1998), but we normalize the prizes ρ_i such that the normalized prizes $\hat{\rho}_i$ are between 0 and 1.

CONSTANT $\rho_i = \hat{\rho}_i = 1$. Every node has the same prize so the goal becomes to visit as many nodes as possible within the length constraint.

UNIFORM $\rho_i \sim \text{DiscreteUniform}(1, 100), \hat{\rho}_i = \frac{\rho_i}{100}$. Every node has a prize that is (discretized) uniform.

DISTANCE $\rho_i = 1 + \left\lfloor 99 \cdot \frac{d_{0i}}{\max_{j=1}^{n} d_{0j}} \right\rfloor, \hat{\rho}_i = \frac{\rho_i}{100}$, where d_{0i} is the distance from the depot to node i. Every node has a (discretized) prize that is proportional to the distance to the depot. This is designed to be challenging as the largest prizes are furthest away from the depot (Fischetti et al., 1998).

The maximum length T^n for instances with n nodes (and a depot) is chosen to be (on average) approximately half of the length of the average TSP tour for uniform TSP instances with n nodes[4]. This idea is that this way approximately (a little more than) half of the nodes can be visited, which results in the most difficult problem instances (Vansteenwegen et al., 2011). This is because the number of possible node selections $\binom{n}{k}$ is maximized if $k = \frac{n}{2}$ and additionally determining the actual path is harder with more nodes selected. We set fixed maximum lengths $T^{20} = 2$, $T^{50} = 3$ and $T^{100} = 4$ instead of adjusting the constraint per instance, such that for some instances more or less nodes can be visited. Note that T^n has the same unit as the node coordinates $\mathbf{x}_i$, so we do not normalize them.

4 The average length of the optimal TSP tour is 3.84, 5.70 and 7.76 for $n = 20, 50, 100$.

A.4.2 Attention model for the OP

ENCODER Similar to the VRP, we use separate parameters for the depot node embedding. Additionally, we provide the node prize $\hat{\rho}_i$ as input feature:

$$\mathbf{h}_i^{(0)} = \begin{cases} W_0^x \mathbf{x}_i + \mathbf{b}_0^x & i = 0 \\ W^x [\mathbf{x}_i, \hat{\rho}_i] + \mathbf{b}^x & i = 1, \dots, n. \end{cases} \tag{84}$$

MAXIMUM LENGTH CONSTRAINT In order to satisfy the maximum length constraint, we keep track of the *remaining* maximum length T_t at time t. Starting at $t = 1$, $T_1 = T$. Then for $t > 0$, T is updated as

$$T_{t+1} = T_t - d_{\pi_{t-1},\pi_t}. \tag{85}$$

Here d_{π_{t-1},π_t} is the distance from node π_{t-1} to π_t and we conveniently define $\pi_0 =$ 0 as we start at the depot.

DECODER CONTEXT The context for the decoder for the OP at time t is the current/last location π_{t-1} and the remaining maximum length T_t. Similar to VRP, we do not need placeholders if $t = 1$ as the route starts at the depot and we do not need to provide information about the first node as the route should end at the depot. We do not need to provide information on the prizes gathered as this is irrelevant for the remaining decisions. The context is defined as:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, T_t\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_0^{(N)}, T_t\right] & t = 1. \end{cases} \tag{86}$$

MASKING In the OP, the depot node can always be visited so is never masked. Regular nodes are masked (i.e. cannot be visited) if either they are already visited or if they cannot be visited within the remaining length constraint:

$$u_{(c)j} = -\infty \Leftrightarrow \exists t' < t : \pi_{t'} = j \text{ or } d_{\pi_{t-1},j} + d_{j0} > T_t \tag{87}$$

A.4.3 Details of baselines

For Compass[5] by Kobeaga et al. (2018), we compile their code and run it with default parameters, only adding `-op -op-ea4op` to indicate that the Genetic Algorithm for the Orienteering Problem should be used. As Compass uses integer coordinates and prizes, we multiply all floats by 10^7 and round to integers. We run the Python Genetic Algorithm[6] with default parameters.

5 https://github.com/bcamath-ds/compass
6 https://github.com/mc-ride/orienteering

TSILIGIRIDES Tsiligirides (1984) describes a heuristic procedure for solving the OP. It consists of sampling 3000 tours through a randomized construction procedure and applies local search on top. The randomized construction part of the heuristic is structurally exactly the same as the heuristic learned by our model, but with a manually engineered function to define the node probabilities. We implement the construction part of the heuristic and compare it to our model (either greedy or sampling 1280 solutions), without the local search (as this can also be applied on top of our model). The final heuristic used by Tsiligirides (1984) uses a formula with multiple terms to define the probability that a node should be selected, but by tuning the weights the form with only one simple term works best, showing the difficulty of manually defining a good probability distribution. In our terms, the heuristic defines a score s_i for each node at time t as the prize divided by the distance from the current node π_{t-1}, raised to the 4th power:

$$s_i = \left(\frac{\hat{\rho}_i}{d_{\pi_{t-1},i}} \right)^4 . \tag{88}$$

Let S be the set with the $\min(4, n-(t-1))$ unvisited nodes with maximum score s_i. Then the node probabilities p_i at time t are defined as

$$p_i = p_\theta(\pi_t = i | s, \boldsymbol{\pi}_{1:t-1}) = \begin{cases} \frac{s_i}{\sum_{j \in S} s_j} & \text{if } i \in S \\ 0 & \text{otherwise.} \end{cases} \tag{89}$$

OR-TOOLS For the Google OR-Tools implementation, we modify the formulation for the CVRP[7]:

- We replace the Manhattan distance by the Euclidean distance.
- We set the number of vehicles to 1.
- For each individual node i, we add a *disjunction constraint* with $\{i\}$ as the set of nodes, and a penalty equal to the prize $\hat{\rho}_i$. This allows OR-Tools to skip node i at a cost $\hat{\rho}_i$.
- We replace the capacity constraint by a maximum distance constraint.
- We remove the objective to minimize the length.

We multiply all float inputs by 10^7 and round to integers. Note that OR-Tools computes penalties for skipped nodes rather than gains for nodes that are visited. The problem is equivalent, but in order to compare the objective value against our method, we need to add the constant sum of all penalties $\sum_i \hat{\rho}_i$ to the OR-Tools objective.

7 https://github.com/google/or-tools/blob/master/examples/python/cvrp.py

Table 7: Additional results for the OP

	Method	20		50		100	
OP (constant)	Gurobi	10.57	(4m)	-		-	
	Compass	10.56	(55s)	29.58	(3m)	59.35	(8m)
	Tsili (greedy)	8.82	(5s)	23.89	(4s)	47.65	(5s)
	AM (greedy)	**10.27**	(0s)	**28.31**	(2s)	**55.81**	(5s)
	GA (Python)	9.72	(10m)	18.52	(1h)	25.68	(5h)
	OR-Tools (10s)	8.54	(52m)	-		-	
	Tsili (sampling)	10.48	(28s)	28.26	(2m)	54.27	(6m)
	AM (sampling)	**10.49**	(4m)	**29.36**	(17m)	**58.33**	(56m)
OP (uniform)	Gurobi	5.85	(7m)	-		-	
	Compass	5.84	(1m)	16.46	(5m)	33.30	(14m)
	Tsili (greedy)	4.85	(4s)	12.80	(4s)	25.48	(5s)
	AM (greedy)	**5.60**	(0s)	**15.62**	(2s)	**31.03**	(5s)
	GA (Python)	5.53	(10m)	10.81	(1h)	14.89	(5h)
	OR-Tools (10s)	4.69	(52m)	-		-	
	Tsili (sampling)	5.70	(26s)	15.28	(2m)	29.54	(5m)
	AM (sampling)	**5.76**	(4m)	**16.25**	(16m)	**32.41**	(51m)

A.4.4 Extended results

Table 7 displays the results for the OP with constant and uniform prize distributions. The results are similar to the results for the prize distribution based on the distance to the depot, although by the calculation time for Gurobi it is confirmed that indeed constant and uniform prize distributions are easier.

A.5 PRIZE COLLECTING TSP

In the prize collecting TSP (PCTSP) each node has a prize ρ_i and an associated penalty β_i. The goal is to minimize the total length of the tour plus the sum of penalties for nodes which are not visited, while collecting at least a given minimum total prize. W.l.o.g. we assume the minimum total prize is equal to 1 (as prizes can be normalized). This problem is related to the OP but inverts the goal (minimizing tour length given a minimum total prize to collect instead of maximizing total prize given a maximum tour length) and additionally adds penalties. Again, we add a special depot node with index 0 and coordinates $\mathbf{x}_0$ and if the model selects the depot, the route is finished. In the PCTSP, it can be beneficial to visit additional nodes, even if the minimum total prize constraint is already satisfied, in order to avoid penalties.

A.5.1 Instance generation

The depot location as well as n node locations are sampled uniformly at random in the unit square. Similar to the OP, we select the distribution for the prizes and penalties with the idea that for difficult instances approximately half of the nodes should be visited. Additionally, neither the prize nor the penalty should dominate the node selection process.

PRIZES We consider uniformly distributed prizes. If we sample prizes $\rho_i \sim \text{Uniform}(0, 1)$, then $\mathbb{E}(\rho_i) = \frac{1}{2}$, and the expected total prize of any subset of $\frac{n}{2}$ nodes (i.e. half of the nodes) would be $\frac{n}{4}$. Therefore, if S is the set of nodes that is visited, we require that $\sum_{i \in S} \rho_i \geq \frac{n}{4}$, or equivalently $\sum_{i \in S} \hat{\rho}_i \geq 1$ where $\hat{\rho}_i = \rho_i \cdot \frac{4}{n}$ is the normalized prize. Note that it can be the case that $\sum_{i=1}^{n} \hat{\rho}_i < 1$, in which case the prize constraint may be violated but it is only allowed to return to the depot after all nodes have been visited.

PENALTIES If penalties are too small, then node selection is determined almost entirely by the minimum total prize constraint. If penalties are too large, we will always visit all nodes, making the minimum total prize constraint obsolete. We argue that in order for the penalties to be meaningful, they should contribute a term in the objective approximately equal to the total length of the tour. If L^n is the expected TSP tour length with n nodes, we try to achieve this by sampling $\beta_i \sim \text{Uniform}(0, 2 \cdot \frac{L^n}{n})$ such that $\mathbb{E}(\beta_i) = \frac{L^n}{n}$ and the expected total penalty for a subset of $\frac{n}{2}$ nodes is $\frac{L^n}{2}$. Following the numbers we use for the OP, we roughly define $\frac{L^n}{2} \approx K^n = 2, 3, 4$ for $n = 20, 50, 100$[8]. This means that we should sample $\beta_i \sim \text{Uniform}(0, 4 \cdot \frac{K^n}{n})$, but empirically we find that $\hat{\beta}_i \sim \text{Uniform}(0, 3 \cdot \frac{K^n}{n})$ works better, which means that the prizes and penalties are balanced as the minimum total prize constraint is sometimes binding and sometimes not.

A.5.2 Attention model for the PCTSP

ENCODER Again, we use separate parameters for the depot node embedding. Additionally, we provide the node prize $\hat{\rho}_i$ and the penalty $\hat{\beta}_i$ as input features:

$$\mathbf{h}_i^{(0)} = \begin{cases} W_0^x \mathbf{x}_i + \mathbf{b}_0^x & i = 0 \\ W^x \left[\mathbf{x}_i, \hat{\rho}_i, \hat{\beta}_i\right] + \mathbf{b}^x & i = 1, \ldots, n. \end{cases} \tag{90}$$

MINIMUM PRIZE CONSTRAINT In order to satisfy the minimum total prize constraint, we keep track of the *remaining* total prize P_t to collect at time t. At $t = 1$, $P_1 = 1$ (as we normalized prizes). Then for $t > 0$, P is updated as

$$P_{t+1} = \max(0, P_t - \hat{\rho}_{\pi_t}). \tag{91}$$

8 The average length of the optimal TSP tour is 3.84, 5.70 and 7.76 for $n = 20, 50, 100$.

If the constraint is satisfied after visiting π_t is visited at time t, then P_{t+1} will be 0.

DECODER CONTEXT The context for the decoder for the PCTSP at time t is the current/last location π_{t-1} and the remaining prize to collect P_t. Again, we do not need placeholders if $t = 1$ as the route starts at the depot and we do not need to provide information about the first node as the route should end at the depot. The information about the prizes collected is implicitly provided to the model in the form of P_t and we do not need to provide any information about the penalties as this is irrelevant for the remaining decisions:

$$\mathbf{h}_{(c)}^{(N)} = \begin{cases} \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{\pi_{t-1}}^{(N)}, P_t\right] & t > 1 \\ \left[\bar{\mathbf{h}}^{(N)}, \mathbf{h}_{0}^{(N)}, P_t\right] & t = 1. \end{cases} \tag{92}$$

MASKING In the PCTSP, the depot node cannot be visited if the remaining prize to collect P_t is larger than 0 and not yet all nodes have been visited (so $t \leq n$):

$$u_{(c)0} = -\infty \Leftrightarrow P_t > 0 \text{ and } t \leq n. \tag{93}$$

Regular nodes are masked (i.e. cannot be visited) only if they are already visited:

$$u_{(c)j} = -\infty \Leftrightarrow \exists t' < t : \pi_{t'} = j. \tag{94}$$

A.5.3 Details of baselines

For the C++ iterated local search (ILS) algorithm[9], we perform 1 run as this takes already 2 minutes per instance (single thread) on average. For the Python ILS algorithm[10] we perform 10 runs as this algorithm is fast. This improved results somewhat for $n = 20$.

OR-TOOLS For the Google OR-Tools implementation, we modify the formulation for the CVRP[11]:

- We replace the Manhattan distance by the Euclidean distance.
- We set the number of vehicles to 1.
- For each individual node i, we add a *disjunction constraint* with $\{i\}$ as the set of nodes, and a penalty equal to the penalty $\hat{\beta}_i$. This allows OR-Tools to skip node i at a cost $\hat{\beta}_i$.
- We replace the capacity constraint by a minimum total prize constraint by adding the prizes as a *dimension*.

9 https://github.com/jordanamecler/PCTSP
10 https://github.com/rafael2reis/salesman
11 https://github.com/google/or-tools/blob/master/examples/python/cvrp.py

We multiply all float inputs by 10^7 and round to integers. Note that we keep the total length objective from the CVRP and add the disjunction constraint with penalties to obtain the right objective.

A.6 STOCHASTIC PCTSP (SPCTSP)

For the SPCTSP, we assume that the real prize collected $\hat{\rho}_i^*$ at each node only becomes known when visiting the node, and $\hat{\rho}_i = \mathbb{E}\left[\hat{\rho}_i^*\right]$ is the expected prize. We assume the real prizes follow a uniform distribution, so $\hat{\rho}_i^* \sim \text{Uniform}(0, 2\hat{\rho}_i)$.

A.6.1 Attention model for the SPCTSP

In order to apply the attention model to the stochastic PCTSP, the only change we need is that we use the real $\hat{\rho}_i^*$ to update the remaining prize to collect P_t in equation 91:

$$P_{t+1} = \max(0, P_t - \hat{\rho}_{\pi_t}^*). \tag{95}$$

We could theoretically use the model trained for PCTSP without retraining, but we choose to retrain. This way the model could (for example) learn that *if* it needs to gather a remaining (normalized) prize of 0.1, it might prefer to visit a node with expected prize 0.2 over a node with expected prize 0.1 as the first real prize will be ≥ 0.1 with probability 75% (uniform prizes) whereas the latter only with 50% and thus has a probability of 50% to not satisfy the constraint.

A.6.2 Rollout baseline in the stochastic setting

Instead of sampling the real prizes online, we already sample them when creating the dataset but keep them hidden to the algorithm. This way, when using a rollout baseline, both the greedy rollout baseline as well as the sample (rollout) from the model use the same real prizes, such that any difference between the two is not a result of stochasticity. This can be seen as a variant of using common random numbers for variance reduction (Glasserman and Yao, 1992).

A.6.3 Details of baselines

For the SPCTSP, it is not possible to formulate an exact model that constructs a tour offline (as any tour can be infeasible with nonzero probability) and an algorithm that computes the optimal decision online should take into account an infinite number of scenarios. As a baseline we implement a strategy that:

1. Plans a tour using the expected prizes $\hat{\rho}_i$
2. Executes part of the tour (not returning to the depot), observing the real prizes $\hat{\rho}_i^*$
3. Computes the remaining total prize that needs to be collected
4. Computes a new tour (again using expected prizes $\hat{\rho}_i$), starting from the last node that was visited, through nodes that have not yet been visited and ending at the depot
5. Repeats the steps (2) - (4) above until the minimum total prize has been collected or all nodes have been visited
6. Returns to the depot

Planning of the tours using deterministic prizes means we need to solve a (deterministic) PCTSP, for which we use the ILS C++ algorithm as this was the strongest algorithm for PCTSP (for large n). Note that in (4), we have a variant of the PCTSP where we do not have a single depot, but rather separate start and end points, whereas the ILS C++ implementation assumes starting and ending at a single depot. However, as the ILS C++ implementation uses a distance matrix, we can effectively plan with a start and end node by defining the distance from the 'depot' to node j as the distance from the start node (the last visited node) to node j, whereas we leave the distance from node j to the depot/end node unchanged (so the distance matrix becomes asymmetrical). Additionally, we remove all nodes (rows/columns in the distance matrix) that have already been visited from the problem.

We consider three variants that differ in the number of nodes that are visited before replanning the tour, for a tradeoff between adaptivity and run time:

1. *All* nodes in the planned tour are visited (except the final return to the depot). We only need to replan and visit additional nodes if the constraint is not satisfied, otherwise we return to the depot.
2. *Half* of the nodes *in the planned tour* are visited, where we visit k nodes if there are $2k + 1$ nodes (excluding the return to the depot), so we round down if an odd number of visits is planned. This way, we will have $O(\log n)$ replanning iterations, while being more adaptive when we are closer to satisfying the total prize constraint. This is a trade-off of adaptivity vs computation time.
3. Only the *first* node is visited, after which we directly replan. This allows the algorithm to take new online information about the real prizes into account directly, but is very expensive to compute as it requires $O(n)$ iterations.

B DEEP POLICY DYNAMIC PROGRAMMING

B.1 THE GRAPH NEURAL NETWORK MODEL

For the TSP, we use the exact model from Joshi et al. (2019a), which we describe here for self-containment. The model uses node input features and edge input features, which get transformed into initial representations of the nodes and edges. These representations then get updated sequentially using a number of graph convolutional layers, which exchange information between nodes and edges, after which the final edge representation is used to predict whether the edge is part of the optimal solution.

INPUT FEATURES AND INITIAL REPRESENTATION The model uses input features for the nodes, consisting of the (x, y)-coordinates, which are then projected into H-dimensional initial embeddings $\mathbf{x}_i^0$ ($H = 300$). The initial edge features $\mathbf{e}_{ij}^0$ are a concatenation of a $\frac{H}{2}$-dimensional projection of the cost (Euclidean distance) c_{ij} from i to j, and a $\frac{H}{2}$-dimensional embedding of the edge type: 0 for normal edges, 1 for edges connecting K-nearest neighbors ($K = 20$) and 2 for *self-loop* edges connecting a node to itself (which are added for ease of implementation).

GRAPH CONVOLUTIONAL LAYERS In each of the $L = 30$ layers of the model, the node and edge representations $\mathbf{x}_i^\ell$ and $\mathbf{e}_{ij}^\ell$ get updated into $\mathbf{x}_i^{\ell+1}$ and $\mathbf{e}_{ij}^{\ell+1}$ (Joshi et al., 2019a):

$$\mathbf{x}_i^{\ell+1} = \mathbf{x}_i^\ell + \mathrm{ReLU}\left(\mathrm{BN}\left(W_1^\ell \mathbf{x}_i^\ell + \sum_{j \in \mathcal{N}(i)} \frac{\sigma(\mathbf{e}_{ij}^\ell)}{\sum_{j' \in \mathcal{N}(i)} \sigma(\mathbf{e}_{ij'}^\ell)} \odot W_2^\ell \mathbf{x}_j^\ell\right)\right) \tag{96}$$

$$\mathbf{e}_{ij}^{\ell+1} = \mathbf{e}_{ij}^\ell + \mathrm{ReLU}\left(\mathrm{BN}\left(W_3^\ell \mathbf{e}_{ij}^\ell + W_4^\ell \mathbf{x}_i^\ell + W_5^\ell \mathbf{x}_j^\ell\right)\right). \tag{97}$$

Here $\mathcal{N}(i)$ is the set of neighbors of node i (in our case all nodes, including i, as we use a fully connected input graph), $\odot$ is the element-wise product and σ is the sigmoid function, applied element-wise to the vector $\mathbf{e}_{ij}^\ell$. $\mathrm{ReLU}(\cdot) = \max(\cdot, 0)$ is the rectified linear unit and BN represents batch normalization (Ioffe and Szegedy, 2015). W_1, W_2, W_3, W_4 and W_5 are trainable parameter matrices, where we fix $W_4 = W_5$ for the symmetric TSP.

OUTPUT PREDICTION After L layers, the final prediction $h_{ij} \in (0, 1)$ is made independently for each edge (i, j) using a multi-layer perceptron (MLP), which takes $\mathbf{e}_{ij}^L$ as input and has two H-dimensional hidden layers with ReLU activation and a 1-

dimensional output layer, with sigmoid activation. We interpret h_{ij} as the predicted probability that the edge (i, j) is part of the optimal solution, which indicates how promising this edge is when searching for the optimal solution.

TRAINING For TSP, the model is trained on a dataset of 1 million optimal solutions, found using Concorde (Applegate et al., 2006), for randomly generated TSP instances. The training loss is a weighted binary cross-entropy loss, that maximizes the prediction quality when h_{ij} is compared to the ground-truth optimal solution. Generating the dataset takes between half a day and a few days (depending on number of CPU cores), and training the model takes a few days on one or multiple GPUs, but both are only required once given a desired data distribution.

B.1.1 Predicting directed edges for the TSPTW

The TSP is an undirected problem, so the neural network implementation[1] by Joshi et al. (2019a) shares the parameters W_4^l and W_5^l in equation 97, i.e. $W_4^l = W_5^l$, resulting in $\mathbf{e}_{ij}^l = \mathbf{e}_{ji}^l$ for all layers l, as for $l = 0$ both directions are initialized the same. While the VRP also is an undirected problem, the TSPTW is directed as the direction of the route determines the times of arrival at different nodes. To allow the model to make different predictions for different directions, we implement W_5^l as a separate parameter, such that the model can have different representations for edges (i, j) and (j, i). We define the training labels accordingly for directed edges, so if edge (i, j) is in the directed solution, it will have a label 1 whereas the edge (j, i) will not (for the undirected TSP and VRP, both labels are 1).

B.1.2 Dataset generation for the TSPTW

We found that using our DP formulation for TSPTW, the instances by Cappart et al. (2021) were all solved optimally, even with a very small beam size (around 10). This is because there is very little overlap in the time windows as a result of the way they are generated, and therefore very few actions are feasible as most of the actions would 'skip over other time windows' (advance the time so much that other nodes can no longer be served)[2]. We conducted some quick experiments with a weaker DP formulation, that only checks if actions *directly* violate time windows, but does not check if an action causes other nodes to be no longer reachable in their time windows. Using this formulation, the DP algorithm can run into many dead ends if just a single node gets skipped, and using the GNN policy (compared to a cost based policy as in Section 4.4.4) made the difference between good solutions and no solution at all being found.

1 https://github.com/chaitjo/graph-convnet-tsp/blob/master/models/gcn_layers.py
2 If all time windows are disjoint, there is only one feasible solution. Therefore, the amount of overlap in time windows determines to some extent the 'branching factor' of the problem and the difficulty.

We made two changes to the data generation procedure by Cappart et al. (2021) to increase the difficulty and make it similar to Da Silva and Urrutia (2010), defining the 'large time window' dataset. First, we sample the time windows around arrival times when visiting nodes in a random order without any waiting time, which is different from Cappart et al. (2021) who 'propagate' the waiting time (as a result of time windows sampled). Our modification causes a tighter schedule with more overlap in time windows, and is similar to Da Silva and Urrutia (2010). Secondly, we increase the maximum time window size from 100 to 1000, which makes that the time windows are in the order of 10% of the horizon[3]. This doubles the maximum time window size of 500 used by Da Silva and Urrutia (2010) for instances with 200 nodes, to compensate for half the number of nodes that can possibly overlap the time window.

To generate the training data, for practical reasons we used DP with the heuristic 'cost heat + potential' strategy and a large beam size (1M), which in many cases results in optimal solutions being found.

B.2 IMPLEMENTATION

We implement the dynamic programming algorithm on the GPU using PyTorch (Paszke et al., 2019). While mostly used as a Deep Learning framework, it can be used to speed up generic (vectorized) computations.

B.2.1 Beam variables

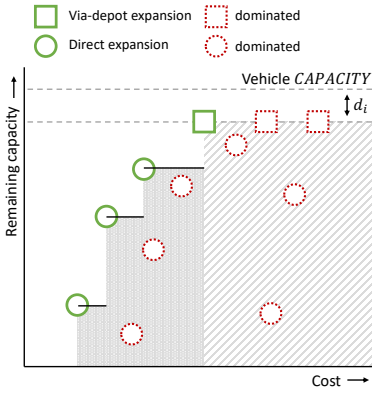
For each solution in the beam, we keep track of the following variables (storing them for all solutions in the beam as a vector): the cost, current node, visited nodes and (for VRP) the remaining capacity or (for TSPTW) the current time. As explained, these variables can be computed incrementally when generating expansions. Additionally, we keep a variable vector *parent*, which, for each solution in the current beam, tracks the index of the solution in the previous beam that generated the expanded solution. To compute the score of the policy for expansions efficiently, we also keep track of the score for each solution and the potential for each node for each solution incrementally.

We do not keep past beams in memory, but at the end of each iteration, we store the vectors containing the parents as well as last actions for each solution on the *trace*. As the solution is completely defined by the sequence of actions, this allows to backtrack the solution after the algorithm has finished. To save GPU memory (especially for larger beam sizes), we store the $O(Bn)$ sized trace on the CPU memory.

3 Serving 100 customers in a 100x100 grid, empirically we find the total schedule duration including waiting (the makespan) is around 5000.

Cost	Capacity	Visited	Current	Direct 0	1	2	3	4	Via-depot 0	1	2	3	4
10	5	01101	1	1	0	0	0	0	1	0	0	1	0
12	8	01101	1	1	0	0	1	0	1	0	0	1	0
13	7	01101	2	1	0	0	1	0	0	0	0	0	0
8	3	01101	4	0	0	0	0	0	1	0	0	1	0
11	7	10101	0	0	1	0	1	0	0	0	0	1	0
12	6	10101	2	0	0	0	1	0	0	0	0	1	0
13	7	10101	2	0	0	0	1	0	0	0	0	1	0

(a) Example beam for VRP with variables, grouped by set of visited nodes (left) and feasible, non-dominated expansions (right), with $2n$ columns corresponding to n direct expansions and n via-depot expansions. Some expansions to unvisited nodes are infeasible, e.g. due to the capacity constraint or a sparse adjacency graph. The shaded areas indicate groups of candidate expansions among which dominances should be checked: for each set of visited nodes there is only one non-dominated via-depot expansion (indicated by solid green square), which must necessarily be an expansion of the solution that has the lowest cost to return to the depot (indicated by the dashed green rectangle ; note that the cost displayed excludes the cost to return to the depot). Direct expansions can be dominated (indicated by red dotted circles) by the single non-dominated via-depot expansion or other direct expansions with the same DP state (set of visited nodes and expanded node, as indicated by the shaded areas). See also Figure 31b for (non-)dominated expansions corresponding to the same DP state.



(b) Example of a set of dominated and non-dominated expansions (direct and via-depot) corresponding to the same DP state (set of visited nodes and expanded node i) for VRP. Non-dominated expansions have lower cost or higher remaining capacity compared to all other expansions. The right striped area indicates expansions dominated by the (single) non-dominated via-depot expansion. The left (darker) areas are dominated by individual direct expansions. Dominated expansions in this area have remaining capacity lower than the cumulative maximum remaining capacity when going from left to right (i.e. in sorted order of increasing cost), indicated by the black horizontal lines.

Figure 31: Implementation of DPDP for VRP

For efficiency, we keep the set of visited nodes as a bitmask, packed into 64-bit long integers (2 for 100 nodes). Using bitwise operations with the packed adjacency matrix, this allows to quickly check feasible expansions (but we need to *unpack* the mask into boolean vectors to find all feasible expansions explicitly). Figure 31a shows an example of the beam (with variables related to the policy and backtracking omitted) for the VRP.

B.2.2 Generating non-dominated expansions

A solution $\boldsymbol{a}$ can only dominate a solution $\boldsymbol{a}'$ if $\text{visited}(\boldsymbol{a}) = \text{visited}(\boldsymbol{a}')$ and $\text{current}(\boldsymbol{a}) = \text{current}(\boldsymbol{a}')$, i.e. if they correspond to the same *DP state*. If this is the case, then, if we denote by $\text{parent}(\boldsymbol{a})$ the parent solution from which $\boldsymbol{a}$ was expanded, it holds that

$$\begin{aligned}\text{visited}(\text{parent}(\boldsymbol{a})) &= \text{visited}(\boldsymbol{a}) \setminus \{\text{current}(\boldsymbol{a})\} \\ &= \text{visited}(\boldsymbol{a}') \setminus \{\text{current}(\boldsymbol{a}')\} \\ &= \text{visited}(\text{parent}(\boldsymbol{a}')).\end{aligned}$$

This means that only expansions from solutions with the same set of visited nodes can dominate each other, so we only need to check for dominated solutions among groups of expansions originating from parent solutions with the same set of visited nodes. Therefore, before generating the expansions, we group the current beam (the parents of the expansions) by the set of visited nodes (see Figure 31a). This can be done efficiently, e.g. using a lexicographic sort of the packed bitmask representing the sets of visited nodes[4].

TRAVELLING SALESMAN PROBLEM For TSP, we can generate (using boolean operations) the $B \times n$ matrix with boolean entries indicating feasible expansions (with n action columns corresponding to n nodes, similar to the $B \times 2n$ matrix for VRP in Figure 31a), i.e. nodes that are unvisited and adjacent to the current node. If we find positive entries sequentially for each column (e.g. by calling TORCH.NONZERO on the transposed matrix), we get all expansions grouped by the combination of action (new current node) and parent set of visited nodes, i.e. grouped by the DP state. We can then trivially find the segments of consecutive expansions corresponding to the same DP state, and we can efficiently find the minimum cost solution for each segment, e.g. using TORCH_SCATTER [5].

VEHICLE ROUTING PROBLEM For VRP, the dominance check has two dimensions (cost *and* remaining capacity) and additionally we need to consider $2n$ actions: n direct and n via the depot (see Figure 31a). Therefore, as we will explain, we check dominances in two stages: first we find (for each DP state) the *single* non-dominated 'via-depot' expansion, after which we find all non-dominated 'direct' expansions (see Figure 31b).

The DP state of each expansion is defined by the expanded node (the new current node) and the set of visited nodes. For each DP state, there can be only *one*[6] non-dominated expansion where the last action was via the depot, since all expansions resulting from 'via-depot actions' have the same remaining capacity as visiting the depot resets the capacity (see Figure 31b). To find this expansion, we first find, for each unique set of visited nodes in the current beam, the solution that can return to the depot with lowest total cost (thus including the cost to return to the depot, indicated by a dashed green rectangle in Figure 31a). The single non-dominated 'via-depot expansion' for each DP state must necessarily be an expansion of this solution. Also observe that this via-depot solution cannot be dominated by a solution expanded using a direct action, which will always have a lower remaining vehicle capacity (assuming positive demands) as can bee seen in Figure 31b. We can thus generate the non-dominated via-depot expansion for each DP state efficiently and independently from the direct expansions.

For each DP state, all *direct* expansions with cost higher (or equal) than the via-depot expansion can directly be removed since they are dominated by the via-depot

4 For efficiency, we use a custom function similar to TORCH.UNIQUE, and argsort the returned inverse after which the resulting permutation is applied to all variables in the beam.
5 https://github.com/rusty1s/pytorch_scatter
6 Unless we have multiple expansions with the same costs, in which case can pick one arbitrarily.

expansion (having higher cost and lower remaining capacity, see Figure 31b). After that, we sort the remaining (if any) direct expansions for each DP state based on the cost (using a segmented sort as the expansions are already grouped if we generate them similarly to TSP, i.e. per column in Figure 31a). For each DP state, the lowest cost solution is never dominated. The other solutions should be kept only if their remaining capacity is strictly larger than the largest remaining capacity of all lower-cost solutions corresponding to the same DP state, which can be computed using a (segmented) cumulative maximum computation (see Figure 31b).

TSP WITH TIME WINDOWS For the TSPTW, the dominance check has two dimensions: cost and time. Therefore, it is similar to the check for non-dominated direct expansions for the VRP (see Figure 31b), but replacing remaining capacity (which should be maximized) by current time (to be minimized). In fact, we could reuse the implementation, if we replace remaining capacity by time multiplied by -1 (as this should be minimized). This means that we sort all expansions for each DP state based on the cost, keep the first solution and keep other solutions only if the time is strictly lower than the lowest current time for all lower-cost solutions, which can be computed using a cumulative minimum computation.

B.2.3 Finding the top B solutions

We may generate all 'candidate' non-dominated expansions and then select the top B using the score function. Alternatively, we can generate expansions in batches, and keep a streaming top B using a priority queue. We use the latter implementation, where we can also derive a bound for the score as soon as we have B candidate expansions. Using this bound, we can already remove solutions before checking dominances, to achieve some speedup in the algorithm.[7]

B.2.4 Performance improvements

There are many possibilities for improving the speed of the algorithm. For example, PyTorch lacks a segmented sort so we use a much slower lexicographic sort instead. Also an efficient GPU priority queue would allow much speedup, as we currently use sorting as PyTorch' top-k function is rather slow for large k. In some cases, a binary search for the k-th largest value can be faster, but this introduces undesired CUDA synchronisation points.

7 This may give slightly different results if the scoring function is inconsistent with the domination rules, i.e. if a better scoring solution would be dominated by a worse scoring solution but is not since that solution is removed using the score bound before checking the dominances.

C ANCESTRAL GUMBEL-TOP-k SAMPLING

C.1 SAMPLING A SET OF GUMBELS WITH MAXIMUM T

As explained in Section 5.3.5, to sample a set of Gumbels with maximum T, let $G_{\phi_i} \sim \text{Gumbel}(\phi_i)$, let $Z = \max_i G_{\phi_i}$ and define

$$\tilde{G}_{\phi_i} = F^{-1}_{\phi_i, T}(F_{\phi_i, Z}(G_{\phi_i})) = -\log(\exp(-T) - \exp(-Z) + \exp(-G_{\phi_i})). \tag{98}$$

Direct computation of equation 98 can be unstable as large terms need to be exponentiated. Instead, we compute

$$\begin{aligned} v_i &= T - G_{\phi_i} + \text{log1mexp}(G_{\phi_i} - Z), \\ \tilde{G}_{\phi_i} &= T - \max(0, v_i) - \text{log1pexp}(-|v_i|) \end{aligned}$$

where we have defined

$$\begin{aligned} \text{log1mexp}(a) &= \log(1 - \exp(a)), \quad a \leq 0 \\ \text{log1pexp}(a) &= \log(1 + \exp(a)). \end{aligned}$$

This is equivalent as

$$\begin{aligned} & T - \max(0, v_i) - \log(1 + \exp(-|v_i|)) \\ &= T - \log(1 + \exp(v_i)) \\ &= T - \log\left(1 + \exp\left(T - G_{\phi_i} + \log\left(1 - \exp\left(G_{\phi_i} - Z\right)\right)\right)\right) \\ &= T - \log\left(1 + \exp\left(T - G_{\phi_i}\right)\left(1 - \exp\left(G_{\phi_i} - Z\right)\right)\right) \\ &= T - \log\left(1 + \exp\left(T - G_{\phi_i}\right) - \exp\left(T - Z\right)\right) \\ &= -\log\left(\exp(-T) + \exp(-G_{\phi_i}) - \exp(-Z)\right) \\ &= \tilde{G}_{\phi_i} \end{aligned}$$

The first step can be easily verified by considering the cases $v_i < 0$ and $v_i \geq 0$. log1mexp and log1pexp can be computed accurately using $\text{log1p}(a) = \log(1 + a)$ and $\text{expm1}(a) = \exp(a) - 1$ (Mächler, 2012):

$$\text{log1mexp}(a) = \begin{cases} \log(-\text{expm1}(a)) & a > -0.693 \\ \text{log1p}(-\exp(a)) & \text{otherwise,} \end{cases}$$

$$\text{log1pexp}(a) = \begin{cases} \text{log1p}(\exp(a)) & a < 18 \\ a + \exp(-a) & \text{otherwise.} \end{cases}$$

C.2 UNBIASEDNESS OF THE IMPORTANCE WEIGHTED ESTIMATOR

We give a proof of unbiasedness of the importance weighted estimator, which is adapted from the proofs by Duffield et al. (2007) and Vieira (2017). For generality of the proof, we enumerate categories in the domain D by $i = 1, ..., n$ and we consider *general* random keys h_i for $i = 1, ..., n$ (not necessarily Gumbel perturbations). As was noted by Vieira (2017), the actual distribution of the keys does *not* influence the unbiasedness of the estimator, but does determine the effective sampling scheme. Using Gumbel perturbed log-probabilities as keys (e.g. $h_i = G_{\phi_i}$) is equivalent to the PPSWOR scheme described by Vieira (2017). For simplicity, we write $p_i = p_{\boldsymbol{\theta}}(i)$ and $q_i(a) = q_{\boldsymbol{\theta},a}(i) = P(h_i > a)$ (see Equation 44), and we define $h_{1:n} = \{h_1, ..., h_n\}$ and $h_{-i} = \{h_1, ..., h_{i-1}, h_{i+1}, ..., h_n\} = h_{1:n} \setminus \{h_i\}$.

Given a *fixed threshold* a, it holds that $q_i(a) = P(i \in S)$ is the probability that category i is included in the sample S, so it can be thought of as the *inclusion probability* of i. Given a *fixed sample size* k, let κ be the $(k+1)$-th largest element of $h_{1:n}$, so κ is the *empirical threshold*. Let κ_i' be the k-th largest element of h_{-i} (the k-th largest of all *other* elements). We first prove Lemma 9, which is then used to prove Theorem 10, which states that the importance weighted estimator is an unbiased estimator of $\mathbb{E}[f(i)]$, for a given function $f(i)$.

Lemma 9. *The expected weight of each term in the importance weighted estimator is 1:*

$$\mathbb{E}_{h_{1:n}}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\right] = 1.$$

Proof. We make use of the observation (slightly rephrased) by Duffield et al. (2007) that conditioning on h_{-i}, we know κ_i', and the event $i \in S$ implies that $\kappa = \kappa_i'$ since i will only be in the sample if $h_i > \kappa_i'$, which means that κ_i' is the $(k+1)$-th largest value of $h_{-i} \cup \{h_i\} = h_{1:n}$. The reverse is also true: if $\kappa = \kappa_i'$ then h_i must be larger than κ_i' since otherwise the $(k+1)$-th largest value of $h_{1:n}$ will be smaller than κ_i'. By separating the expectation over $h_{1:n}$ it follows that

$$\begin{aligned}
&\mathbb{E}_{h_{1:n}}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\right] \\
&= \mathbb{E}_{h_{-i}}\left[\mathbb{E}_{h_i}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\middle| h_i\right]\right] \\
&= \mathbb{E}_{h_{-i}}\left[\mathbb{E}_{h_i}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\middle| h_{-i}, i \in S\right] P(i \in S|h_{-i}) + \mathbb{E}_{h_i}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\middle| h_{-i}, i \notin S\right] P(i \notin S|h_{-i})\right] \\
&= \mathbb{E}_{h_{-i}}\left[\mathbb{E}_{h_i}\left[\frac{1}{q_i(\kappa)}\middle| h_{-i}, i \in S\right] q_i(\kappa_i') + 0\right] \\
&= \mathbb{E}_{h_{-i}}\left[\mathbb{E}_{h_i}\left[\frac{1}{q_i(\kappa)}\middle| \kappa = \kappa_i'\right] q_i(\kappa_i')\right] \\
&= \mathbb{E}_{h_{-i}}\left[\frac{1}{q_i(\kappa_i')} q_i(\kappa_i')\right] = \mathbb{E}_{h_{-i}}[1] = 1.
\end{aligned}$$

□

Theorem 10. *The importance weighted estimator is an unbiased estimator of* $\mathbb{E}[f(i)]$*:*

$$\mathbb{E}_{h_{1:n}}\left[\sum_{i\in S}\frac{p_i}{q_i(\kappa)}f(i)\right] = \mathbb{E}[f(i)].$$

Proof.

$$\begin{aligned}
\mathbb{E}_{h_{1:n}}\left[\sum_{i\in S}\frac{p_i}{q_i(\kappa)}f(i)\right] &= \mathbb{E}_{h_{1:n}}\left[\sum_{i=1}^{n}\frac{p_i}{q_i(\kappa)}f(i)\mathbb{1}_{\{i\in S\}}\right] \\
&= \sum_{i=1}^{n} p_i f(i)\cdot \mathbb{E}_{h_{1:n}}\left[\frac{\mathbb{1}_{\{i\in S\}}}{q_i(\kappa)}\right] \\
&= \sum_{i=1}^{n} p_i f(i)\cdot 1 \\
&= \mathbb{E}[f(i)].
\end{aligned}$$

□

C.3 NUMERICAL STABILITY OF IMPORTANCE WEIGHTS

We have to take care computing the importance weights as, depending on the entropy, the terms in the quotient $\frac{p_i}{q_i(\kappa)}$ (using notation from Appendix C.2) can become very small, and the computation of $q_i(\kappa) = 1-\exp(-\exp(\phi_i-\kappa))$ (equation 44) can suffer from catastrophic cancellation. We can rewrite this expression using the more numerically stable implementation $\mathrm{expm1}(x) = \exp(x)-1$ as $q_i(\kappa) = -\mathrm{expm1}(-\exp(\phi_i-\kappa))$ but this may still suffer from instability as $\exp(\phi_i-\kappa)$ can underflow if $\phi_i-\kappa$ is small. Instead, for $\phi_i-\kappa<-10$ we use

$$\log(1-\exp(-z)) = \log(z)-\frac{z}{2}+\frac{z^2}{24}-\frac{z^4}{2880}+\mathcal{O}(z^6)$$

to directly compute the log importance weight using $z=\exp(\phi_i-\kappa)$ and $\phi_i=\log p_i$ (we assume ϕ_i is normalized). Since $q_i(\kappa)=1-\exp(-z)$, we have

$$\begin{aligned}
\log\left(\frac{p_i}{q_i(\kappa)}\right) &= \log p_i - \log q_i(\kappa) \\
&= \log p_i - \log\left(1-\exp(-z)\right) \\
&= \log p_i - \left(\log(z)-\frac{z}{2}+\frac{z^2}{24}-\frac{z^4}{2880}+\mathcal{O}(z^6)\right) \\
&= \log p_i - \left(\phi_i-\kappa-\frac{z}{2}+\frac{z^2}{24}-\frac{z^4}{2880}+\mathcal{O}(z^6)\right) \\
&= \kappa+\frac{z}{2}-\frac{z^2}{24}+\frac{z^4}{2880}+\mathcal{O}(z^6).
\end{aligned}$$

If $\phi_i-\kappa<-10$ then $0<z<10^{-6}$ so the result will not lose significant digits.

D ESTIMATING GRADIENTS WITH SAMPLES WITHOUT REPLACEMENT

D.1 NOTATION

Throughout Appendix D we will use the following notation from Maddison et al. (2014):

$$
\begin{aligned}
e_\phi(g) &= \exp(-g+\phi) \\
F_\phi(g) &= \exp(-\exp(-g+\phi)) \\
f_\phi(g) &= e_\phi(g)F_\phi(g).
\end{aligned}
$$

This means that $F_\phi(g)$ is the CDF and $f_\phi(g)$ the PDF of the Gumbel(ϕ) distribution. Additionally we will use the identities by Maddison et al. (2014):

$$F_\phi(g)F_\gamma(g) = F_{\log(\exp(\phi)+\exp(\gamma))}(g) \tag{99}$$

$$\int_{g=a}^{b} e_\gamma(g)F_\phi(g)\partial g = (F_\phi(b) - F_\phi(a))\frac{\exp(\gamma)}{\exp(\phi)}. \tag{100}$$

Also, we will use the following notation, definitions and identities (see Kool et al. (2019c)):

$$\phi_i = \log p(i) \tag{101}$$

$$\phi_S = \log \sum_{i\in S} p(i) = \log \sum_{i\in S} \exp \phi_i \tag{102}$$

$$\phi_{D\setminus S} = \log \sum_{i\in D\setminus S} p(i) = \log\left(1 - \sum_{i\in S} p(i)\right) = \log(1-\exp(\phi_S)) \tag{103}$$

$$G_{\phi_i} \sim \text{Gumbel}(\phi_i) \tag{104}$$

$$G_{\phi_S} = \max_{i\in S} G_{\phi_i} \sim \text{Gumbel}(\phi_S) \tag{105}$$

For a proof of equation 105, see Maddison et al. (2014).

D.2 COMPUTATION OF $p(S^k)$, $p^{D\setminus C}(S\setminus C)$ AND $R(S^k, s)$

We can sample the set S^k from the Plackett-Luce distribution using the Gumbel-top-k trick by drawing Gumbel variables $G_{\phi_i} \sim \text{Gumbel}(\phi_i)$ for each element and returning the indices of the k largest Gumbels. If we ignore the ordering, this means we will obtain the set S^k if $\min_{i\in S^k} G_{\phi_i} > \max_{i\in D\setminus S^k} G_{\phi_i}$. Omitting the superscript

k for clarity, we can use the Gumbel-max trick, i.e. that $G_{\phi_{D\setminus S}} = \max_{i\notin S} G_{\phi_i} \sim \text{Gumbel}(\phi_{D\setminus S})$ (equation 105) and marginalize over $G_{\phi_{D\setminus S}}$:

$$\begin{aligned}
p(S) &= P(\min_{i\in S} G_{\phi_i} > G_{\phi_{D\setminus S}}) \\
&= P(G_{\phi_i} > G_{\phi_{D\setminus S}}, i \in S) \\
&= \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) P(G_{\phi_i} > g_{\phi_{D\setminus S}}, i \in S) \partial g_{\phi_{D\setminus S}} \\
&= \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) \prod_{i\in S} \left(1 - F_{\phi_i}(g_{\phi_{D\setminus S}})\right) \partial g_{\phi_{D\setminus S}} \qquad (106) \\
&= \int_{u=0}^{1} \prod_{i\in S} \left(1 - F_{\phi_i}\left(F^{-1}_{\phi_{D\setminus S}}(u)\right)\right) \partial u \qquad (107)
\end{aligned}$$

Here we have used a change of variables $u = F_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}})$. This expression can be efficiently numerically integrated (although another change of variables may be required for numerical stability depending on the values of $\boldsymbol{\phi}$).

EXACT COMPUTATION IN $O(2^k)$ The integral in equation 106 can be computed exactly using the identity

$$\prod_{i\in S}(a_i - b_i) = \sum_{C\subseteq S} (-1)^{|C|} \prod_{i\in C} b_i \prod_{i\in S\setminus C} a_i$$

which gives

$$\begin{aligned}
p(S) &= \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) \prod_{i\in S} \left(1 - F_{\phi_i}(g_{\phi_{D\setminus S}})\right) \partial g_{\phi_{D\setminus S}} \\
&= \sum_{C\subseteq S} (-1)^{|C|} \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) \prod_{i\in C} F_{\phi_i}(g_{\phi_{D\setminus S}}) \prod_{i\in S\setminus C} 1 \partial g_{\phi_{D\setminus S}} \\
&= \sum_{C\subseteq S} (-1)^{|C|} \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} e_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) F_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) F_{\phi_C}(g_{\phi_{D\setminus S}}) \partial g_{\phi_{D\setminus S}} \\
&= \sum_{C\subseteq S} (-1)^{|C|} \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} e_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) F_{\phi_{(D\setminus S)\cup C}}(g_{\phi_{D\setminus S}}) \partial g_{\phi_{D\setminus S}} \\
&= \sum_{C\subseteq S} (-1)^{|C|} (1 - 0) \frac{\exp(\phi_{D\setminus S})}{\exp(\phi_{(D\setminus S)\cup C})} \\
&= \sum_{C\subseteq S} (-1)^{|C|} \frac{1 - \sum_{i\in S} p(i)}{1 - \sum_{i\in S\setminus C} p(i)}. \qquad (108)
\end{aligned}$$

COMPUTATION OF $p^{D\setminus C}(S \setminus C)$ When using the Gumbel-top-k trick over the restricted domain $D \setminus C$, we do *not* need to renormalize the log-probabilities $\phi_s, s \in D \setminus C$ since the Gumbel-top-k trick applies to unnormalized log-probabilities. Also,

assuming $C \subseteq S^k$, it holds that $(D \setminus C) \setminus (S \setminus C) = D \setminus S$. This means that we can compute $p^{D\setminus C}(S \setminus C)$ similar to equation 106:

$$\begin{aligned} p^{D\setminus C}(S \setminus C) &= P(\min_{i\in S\setminus C} G_{\phi_i} > G_{\phi_{(D\setminus C)\setminus(S\setminus C)}}) \\ &= P(\min_{i\in S\setminus C} G_{\phi_i} > G_{\phi_{D\setminus S}}) \\ &= \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) \prod_{i\in S\setminus C} \left(1 - F_{\phi_i}(g_{\phi_{D\setminus S}})\right) \partial g_{\phi_{D\setminus S}}. \end{aligned} \tag{109}$$

COMPUTATION OF $R(S^k,s)$ Note that, using equation 55, it holds that

$$\sum_{s\in S^k} \frac{p^{D\setminus\{s\}}(S^k \setminus \{s\})p(s)}{p(S^k)} = \sum_{s\in S^k} P(b_1 = s|S^k) = 1$$

from which it follows that

$$p(S^k) = \sum_{s\in S^k} p^{D\setminus\{s\}}(S^k \setminus \{s\})p(s)$$

such that

$$R(S^k,s) = \frac{p^{D\setminus\{s\}}(S^k \setminus \{s\})}{p(S^k)} = \frac{p^{D\setminus\{s\}}(S^k \setminus \{s\})}{\sum_{s'\in S^k} p^{D\setminus\{s'\}}(S^k \setminus \{s'\})p(s')}. \tag{110}$$

This means that, to compute the leave-one-out ratio for all $s \in S^k$, we only need to compute $p^{D\setminus\{s\}}(S^k \setminus \{s\})$ for $s \in S^k$. When using the numerical integration or summation in $O(2^k)$, we can reuse computation, whereas using the naive method, the cost is $O(k \cdot (k-1)!) = O(k!)$, making the total computational cost comparable to computing just $p(S^k)$, and the same holds when computing the 'second-order' leave one out ratios for the built-in baseline (equation 63).

DETAILS OF NUMERICAL INTEGRATION For computation of the leave-one-out ratio (equation 110) for large k we can use the numerical integration, where we need to compute equation 109 with $C = \{s\}$. For this purpose, we rewrite the integral as

$$\begin{aligned} p^{D\setminus C}(S \setminus C) &= \int_{g_{\phi_{D\setminus S}}=-\infty}^{\infty} f_{\phi_{D\setminus S}}(g_{\phi_{D\setminus S}}) \prod_{i\in S\setminus C} \left(1 - F_{\phi_i}(g_{\phi_{D\setminus S}})\right) \partial g_{\phi_{D\setminus S}} \\ &= \int_{u=0}^{1} \prod_{i\in S\setminus C} \left(1 - F_{\phi_i}\left(F^{-1}_{\phi_{D\setminus S}}(u)\right)\right) \partial u \\ &= \int_{u=0}^{1} \prod_{i\in S\setminus C} \left(1 - u^{\exp(\phi_i - \phi_{D\setminus S})}\right) \partial u \\ &= \exp(b) \cdot \int_{v=0}^{1} v^{\exp(b)-1} \prod_{i\in S\setminus C} \left(1 - v^{\exp(\phi_i - \phi_{D\setminus S} + b)}\right) \partial v \\ &= \exp(a + \phi_{D\setminus S}) \cdot \int_{v=0}^{1} v^{\exp(a+\phi_{D\setminus S})-1} \prod_{i\in S\setminus C} \left(1 - v^{\exp(\phi_i + a)}\right) \partial v. \end{aligned}$$

Here we have used change of variables $v = u^{exp(-b)}$ and $a = b - \phi_{D\setminus S}$. This form allows to compute the integrands efficiently, as

$$\prod_{i\in S\setminus C}\left(1 - v^{\exp(\phi_i + a)}\right) = \frac{\prod_{i\in S}\left(1 - v^{\exp(\phi_i+a)}\right)}{\prod_{i\in C}\left(1 - v^{\exp(\phi_i+a)}\right)}$$

where the numerator only needs to computed once, and, since $C = \{s\}$ when computing equation 110, the denominator only consists of a single term.

The choice of a may depend on the setting, but we found that $a = 5$ is a good default option which leads to an integral that is generally smooth and can be accurately approximated using the trapezoid rule. We compute the integrands in logarithmic space and sum the terms using the stable LOGSUMEXP trick. In our code we provide an implementation which also computes all second-order leave-one-out ratios efficiently.

D.3 THE SUM-AND-SAMPLE ESTIMATOR

D.3.1 Unbiasedness of the sum-and-sample estimator

We show that the sum-and-sample estimator is unbiased for any set $C \subset D$ (see also Liang et al. (2018) and Liu et al. (2019)):

$$\begin{aligned}
&\mathbb{E}_{x\sim p^{D\setminus C}(x)}\left[\sum_{c\in C} p(c)f(c) + \left(1 - \sum_{x\in C} p(c)\right) f(x)\right] \\
&= \sum_{c\in C} p(c)f(c) + \left(1 - \sum_{c\in C} p(c)\right) \mathbb{E}_{x\sim p^{D\setminus C}(x)}[f(x)] \\
&= \sum_{c\in C} p(c)f(c) + \left(1 - \sum_{c\in C} p(c)\right) \sum_{x\in D\setminus C} \frac{p(x)}{1 - \sum_{c\in C} p(c)} f(x) \\
&= \sum_{c\in C} p(c)f(c) + \sum_{x\in D\setminus C} p(x)f(x) \\
&= \sum_{x\in D} p(x)f(x) \\
&= \mathbb{E}_{x\sim p(x)}[f(x)]
\end{aligned}$$

D.3.2 Rao-Blackwellization of the stochastic sum-and-sample estimator

In this section we give the proof that Rao-Blackwellizing the stochastic sum-and-sample estimator results in the unordered set estimator.

Theorem 11. *Rao-Blackwellizing the stochastic sum-and-sample estimator results in the unordered set estimator, i.e.*

$$\mathbb{E}_{B^k \sim p(B^k|S^k)} \left[\sum_{j=1}^{k-1} p(b_j) f(b_j) + \left(1 - \sum_{j=1}^{k-1} p(b_j) \right) f(b_k) \right] = \sum_{s \in S^k} p(s) R(S^k, s) f(s). \quad (111)$$

Proof. To give the proof, we first prove three Lemmas.

Lemma 12.

$$P(b_k = s|S^k) = \frac{p(S^k \setminus \{s\})}{p(S^k)} \frac{p(s)}{1 - \sum_{s' \in S^k \setminus \{s\}} p(s')} \quad (112)$$

Proof. Similar to the derivation of $P(b_1 = s|S^k)$ (equation 55), we can write:

$$\begin{aligned} P(b_k = s|S^k) &= \frac{P(S^k \cap b_k = s)}{p(S^k)} \\ &= \frac{p(S^k \setminus \{s\}) p^{D \setminus (S^k \setminus \{s\})}(s)}{p(S^k)} \\ &= \frac{p(S^k \setminus \{s\})}{p(S^k)} \frac{p(s)}{1 - \sum_{s' \in S^k \setminus \{s\}} p(s')}. \end{aligned}$$

The step from the first to the second row comes from analyzing the event $S^k \cap b_k = s$ using sequential sampling: to sample S^k (including s) with s being the k-th element means that we should first sample $S^k \setminus \{s\}$ (in any order), and then sample s from the distribution restricted to $D \setminus (S^k \setminus \{s\})$. □

Lemma 13.

$$p(S) + p(S \setminus \{s\}) \frac{1 - \sum_{s' \in S} p(s')}{1 - \sum_{s' \in S \setminus \{s\}} p(s')} = p^{D \setminus \{s\}}(S \setminus \{s\}) \quad (113)$$

Dividing equation 108 by $1 - \sum_{s' \in S} p(s')$ on both sides, we obtain

Proof.

$$
\begin{aligned}
&\frac{p(S)}{1-\sum_{s'\in S} p(s')} \\
=&\sum_{C\subseteq S}(-1)^{|C|}\frac{1}{1-\sum_{s'\in S\setminus C} p(s')} \\
=&\sum_{C\subseteq S\setminus\{s\}}\left((-1)^{|C|}\frac{1}{1-\sum_{s'\in S\setminus C} p(s')}+(-1)^{|C\cup\{s\}|}\frac{1}{1-\sum_{s'\in S\setminus(C\cup\{s\})} p(s')}\right) \\
=&\sum_{C\subseteq S\setminus\{s\}}(-1)^{|C|}\frac{1}{1-\sum_{s'\in S\setminus C} p(s')}+\sum_{C\subseteq S\setminus\{s\}}(-1)^{|C\cup\{s\}|}\frac{1}{1-\sum_{s'\in S\setminus(C\cup\{s\})} p(s')} \\
=&\sum_{C\subseteq S\setminus\{s\}}(-1)^{|C|}\frac{1}{1-p(s)-\sum_{s'\in(S\setminus\{s\})\setminus C} p(s')}-\sum_{C\subseteq S\setminus\{s\}}(-1)^{|C|}\frac{1}{1-\sum_{s'\in(S\setminus\{s\})\setminus C} p(s')} \\
=&\frac{1}{1-p(s)}\sum_{C\subseteq S\setminus\{s\}}(-1)^{|C|}\frac{1}{1-\sum_{s'\in(S\setminus\{s\})\setminus C}\frac{p(s')}{1-p(s)}}-\frac{p(S\setminus\{s\})}{1-\sum_{s'\in S\setminus\{s\}} p(s')} \\
=&\frac{1}{1-p(s)}\frac{p^{D\setminus\{s\}}(S\setminus\{s\})}{1-\sum_{s'\in S\setminus\{s\}}\frac{p(s')}{1-p(s)}}-\frac{p(S\setminus\{s\})}{1-\sum_{s'\in S\setminus\{s\}} p(s')} \\
=&\frac{p^{D\setminus\{s\}}(S\setminus\{s\})}{1-p(s)-\sum_{s'\in S\setminus\{s\}} p(s')}-\frac{p(S\setminus\{s\})}{1-\sum_{s'\in S\setminus\{s\}} p(s')} \\
=&\frac{p^{D\setminus\{s\}}(S\setminus\{s\})}{1-\sum_{s'\in S} p(s')}-\frac{p(S\setminus\{s\})}{1-\sum_{s'\in S\setminus\{s\}} p(s')}.
\end{aligned}
$$

Multiplying by $1-\sum_{s'\in S} p(s')$ and rearranging terms proves Lemma 13. □

Lemma 14.

$$
p(s)+\left(1-\sum_{s'\in S^k} p(s')\right)P(b_k=s|S^k)=p(s)R(S^k,s) \tag{114}
$$

Proof. First using Lemma 12 and then Lemma 13 we find

$$
\begin{aligned}
&p(s)+\left(1-\sum_{s'\in S^k} p(s')\right)P(b_k=s|S^k) \\
=&p(s)+\left(1-\sum_{s'\in S^k} p(s')\right)\frac{p(S^k\setminus\{s\})}{p(S^k)}\frac{p(s)}{1-\sum_{s'\in S^k\setminus\{s\}} p(s')} \\
=&\frac{p(s)}{p(S^k)}\left(p(S^k)+\frac{1-\sum_{s'\in S^k} p(s')}{1-\sum_{s'\in S^k\setminus\{s\}} p(s')}p(S^k\setminus\{s\})\right) \\
=&\frac{p(s)}{p(S^k)}p^{D\setminus\{s\}}(S^k\setminus\{s\}) \\
=&p(s)R(S^k,s).
\end{aligned}
$$

□

Now we can complete the proof of Theorem 11 by adding $p(b_k)f(b_k) - p(b_k)f(b_k) = 0$ to the estimator, moving the terms independent of B^k outside the expectation and using Lemma 14:

$$
\begin{aligned}
&\mathbb{E}_{B^k \sim p(B^k|S^k)} \left[\sum_{j=1}^{k-1} p(b_j) f(b_j) + \left(1 - \sum_{j=1}^{k-1} p(b_j)\right) f(b_k) \right] \\
=&\mathbb{E}_{B^k \sim p(B^k|S^k)} \left[\sum_{j=1}^{k} p(b_j) f(b_j) + \left(1 - \sum_{j=1}^{k} p(b_j)\right) f(b_k) \right] \\
=& \sum_{s \in S^k} p(s) f(s) + \mathbb{E}_{B^k \sim p(B^k|S^k)} \left[\left(1 - \sum_{s' \in S^k} p(s')\right) f(b_k) \right] \\
=& \sum_{s \in S^k} p(s) f(s) + \sum_{s \in S^k} \left(1 - \sum_{s' \in S^k} p(s')\right) P(b_k = s|S^k) f(s) \\
=& \sum_{s \in S^k} \left(p(s) + \left(1 - \sum_{s' \in S^k} p(s')\right) P(b_k = s|S^k) \right) f(s) \\
=& \sum_{s \in S^k} p(s) R(S^k, s) f(s).
\end{aligned}
$$

□

D.3.3 The stochastic sum-and-sample estimator with multiple samples

As was discussed in Liu et al. (2019), one can trade off the number of summed terms and number of sampled terms to maximize the achieved variance reduction. As a generalization of Theorem 11 (the stochastic sum-and-sample estimator with $k-1$ summed terms), we introduce here the stochastic sum-and-sample estimator that sums $k-m$ terms and samples $m > 1$ terms *without replacement*. To estimate the sampled term, we use the unordered set estimator on the m samples without replacement, on the domain restricted to $D \setminus B^{k-m}$. In general, we denote the *unordered set estimator* restricted to the domain $D \setminus C$ by

$$
e^{\mathrm{US}, D \setminus C}(S^k) = \sum_{s \in S^k \setminus C} p(s) R^{D \setminus C}(S^k, s) f(s) \tag{115}
$$

where $R^{D \setminus C}(S^k, s)$ is the *leave-one-out ratio* restricted to the domain $D \setminus C$, similar to the second order leave-one-out ratio in equation 64:

$$
R^{D \setminus C}(S^k, s) = \frac{p_{\boldsymbol{\theta}}^{(D \setminus C) \setminus \{s\}}((S^k \setminus C) \setminus \{s\})}{p_{\boldsymbol{\theta}}^{D \setminus C}(S^k \setminus C)}. \tag{116}
$$

While we can also constrain $S^k \subseteq (D \setminus C)$, this definition is consistent with equation 64 and allows simplified notation.

Theorem 15. *Rao-Blackwellizing the stochastic sum-and-sample estimator with $m > 1$ samples results in the unordered set estimator, i.e.*

$$\mathbb{E}_{B^k \sim p(B^k|S^k)}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) e^{US, D\setminus B^{k-m}}(S^k)\right] = \sum_{s \in S^k} p(s)R(S^k, s)f(s). \quad (117)$$

Proof. Recall that for the unordered set estimator, it holds that

$$e^{\mathrm{US}}(S^k) = \mathbb{E}_{b_1 \sim p(b_1|S^k)}\left[f(b_1)\right] = \mathbb{E}_{x \sim p(x)}\left[f(x)\middle| x \in S^k\right] \quad (118)$$

□

which for the restricted equivalent (with restricted distribution $p^{D\setminus C}$) translates into

$$e^{\mathrm{US}, D\setminus C}(S^k) = \mathbb{E}_{x \sim p^{D\setminus C}(x)}\left[f(x)\middle| x \in S^k\right] = \mathbb{E}_{x \sim p(x)}\left[f(x)\middle| x \in S^k, x \notin C\right]. \quad (119)$$

Now we consider the distribution $b_{k-m+1}|S^k, B^{k-m}$: the distribution of the first element sampled (without replacement) after sampling B^{k-m}, given (conditionally on the event) that the set of k samples is S^k, so we have $b_{k-m+1} \in S^k$ and $b_{k-m+1} \notin B^{k-m}$. This means that its conditional expectation of $f(b_{k-m+1})$ is the restricted unordered set estimator for $C = B^{k-m}$ since

$$\begin{aligned} e^{\mathrm{US}, D\setminus B^{k-m}}(S^k) &= \mathbb{E}_{x \sim p(x)}\left[f(x)\middle| x \in S^k, x \notin B^{k-m}\right] \\ &= \mathbb{E}_{b_{k-m+1} \sim p(b_{k-m+1}|S^k, B^{k-m})}\left[f(b_{k-m+1})\right]. \end{aligned} \quad (120)$$

Observing that the definition (equation 117) of the stochastic sum-and-sample estimator does not depend on the actual order of the m samples, and using equation 120, we can reduce the multi-sample estimator to the stochastic sum-and-sample estimator with $k' = k - m + 1$, such that the result follows from equation 111:

$$\begin{aligned} &\mathbb{E}_{B^k \sim p(B^k|S^k)}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) e^{\mathrm{US}, D\setminus B^{k-m}}(S^k)\right] \\ =&\mathbb{E}_{B^{k-m} \sim p(B^{k-m}|S^k)}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) e^{\mathrm{US}, D\setminus B^{k-m}}(S^k)\right] \\ =&\mathbb{E}_{B^{k-m} \sim p(B^{k-m}|S^k)}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) \mathbb{E}_{b_{k-m+1} \sim p(b_{k-m+1}|S^k, B^{k-m})}\left[f(b_{k-m+1})\right]\right] \\ =&\mathbb{E}_{B^{k-m+1} \sim p(B^{k-m+1}|S^k)}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) f(b_{k-m+1})\right] \\ =&\mathbb{E}_{S^{k-m+1}|S^k}\left[\mathbb{E}_{B^{k-m+1} \sim p(B^{k-m+1}|S^{k-m+1})}\left[\sum_{j=1}^{k-m} p(b_j)f(b_j) + \left(1 - \sum_{j=1}^{k-m} p(b_j)\right) f(b_{k-m+1})\right]\right] \\ =&\mathbb{E}_{S^{k-m+1}|S^k}\left[\sum_{s \in S^k} p(s)R(S^k, s)f(s)\right] \\ =&\sum_{s \in S^k} p(s)R(S^k, s)f(s). \end{aligned} \quad (121)$$

D.4 THE IMPORTANCE-WEIGHTED ESTIMATOR

D.4.1 Rao-Blackwellization of the importance-weighted estimator

In this section we give the proof that Rao-Blackwellizing the importance-weighted estimator results in the unordered set estimator.

Theorem 16. *Rao-Blackwellizing the importance-weighted estimator results in the unordered set estimator, i.e.:*

$$\mathbb{E}_{\kappa\sim p(\kappa|S^k)}\left[\sum_{s\in S^k}\frac{p(s)}{1-F_{\phi_s}(\kappa)}f(s)\right]=\sum_{s\in S^k}p(s)R(S^k,s)f(s). \tag{122}$$

Here we have slightly rewritten the definition of the importance-weighted estimator, using that $q(s,a)=P(g_{\phi_s}>a)=1-F_{\phi_s}(a)$, where F_{ϕ_s} is the CDF of the Gumbel distribution (see Appendix D.1).

Proof. We first prove the following Lemma:

Lemma 17.

$$\mathbb{E}_{\kappa\sim p(\kappa|S^k)}\left[\frac{1}{1-F_{\phi_s}(\kappa)}\right]=R(S^k,s) \tag{123}$$

Proof. Conditioning on S^k, we know that the elements in S^k have the k largest perturbed log-probabilities, so κ, the $(k+1)$-th largest perturbed log-probability is the largest perturbed log-probability in $D\setminus S^k$, and satisfies $\kappa=\max_{s\in D\setminus S^k}g_{\phi_s}=g_{\phi_{D\setminus S^k}}\sim\text{Gumbel}(\phi_{D\setminus S^k})$. Computing $p(\kappa|S^k)$ using Bayes' Theorem, we have

$$p(\kappa|S^k)=\frac{p(S^k|\kappa)p(\kappa)}{p(S^k)}=\frac{\prod_{s\in S^k}(1-F_{\phi_s}(\kappa))f_{\phi_{D\setminus S^k}}(\kappa)}{p(S^k)} \tag{124}$$

which allows us to compute (using equation 109 with $C=\{s\}$ and $g_{\phi_{D\setminus S}}=\kappa$)

$$\begin{aligned}
&\mathbb{E}_{\kappa\sim p(\kappa|S^k)}\left[\frac{1}{1-F_{\phi_s}(\kappa)}\right]\\
=&\int_{\kappa=-\infty}^{\infty}p(\kappa|S^k)\frac{1}{1-F_{\phi_s}(\kappa)}\partial\kappa\\
=&\int_{\kappa=-\infty}^{\infty}\frac{\prod_{s\in S^k}(1-F_{\phi_s}(\kappa))f_{\phi_{D\setminus S^k}}(\kappa)}{p(S^k)}\frac{1}{1-F_{\phi_s}(\kappa)}\partial\kappa\\
=&\frac{1}{p(S^k)}\int_{\kappa=-\infty}^{\infty}\prod_{s\in S^k\setminus\{s\}}(1-F_{\phi_s}(\kappa))f_{\phi_{D\setminus S^k}}(\kappa)\partial\kappa\\
=&\frac{1}{p(S^k)}p^{D\setminus\{s\}}(S\setminus\{s\})\\
=&R(S^k,s).
\end{aligned}$$

□

Using Lemma 17 we find

$$\begin{aligned}
&\mathbb{E}_{\kappa\sim p(\kappa|S^k)}\left[\sum_{s\in S^k}\frac{p(s)}{1-F_{\phi_s}(\kappa)}f(s)\right]\\
&=\sum_{s\in S^k}p(s)\mathbb{E}_{\kappa\sim p(\kappa|S^k)}\left[\frac{1}{1-F_{\phi_s}(\kappa)}\right]f(s)\\
&=\sum_{s\in S^k}p(s)R(S^k,s)f(s).
\end{aligned}$$

□

D.4.2 The importance-weighted policy gradient estimator with built-in baseline

For self-containment we include this section, which is adapted from Kool et al. (2019b). The importance-weighted policy gradient estimator combines REINFORCE (Williams, 1992) with the importance-weighted estimator (Duffield et al., 2007; Vieira, 2017) in equation 61 which results in an unbiased estimator of the policy gradient $\nabla_{\boldsymbol{\theta}}\mathbb{E}_{p_{\boldsymbol{\theta}}(x)}[f_{\boldsymbol{\theta}}(x)]$:

$$e^{\text{IWPG}}(S^k,\kappa)=\sum_{s\in S^k}\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(s)f(s)=\sum_{s\in S^k}\frac{\nabla_{\boldsymbol{\theta}}p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}f(s). \tag{125}$$

Recall that κ is the $(k+1)$-th largest perturbed log-probability (see Section 6.4.2). We compute a lower variance but biased variant by normalizing the importance weights using the normalization $W(S^k)=\sum_{s\in S^k}\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}$.

As we show in Kool et al. (2019b), we can include a 'baseline' $B(S^k)=\sum_{s\in S^k}\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}f(s)$ and correct for the bias (since it depends on the complete sample S^k) by weighting individual terms of the estimator by $1-p_{\boldsymbol{\theta}}(s)+\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}$:

$$e^{\text{IWPGBL}}(S^k,\kappa)=\sum_{s\in S^k}\frac{\nabla_{\boldsymbol{\theta}}p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}\left(f(s)\left(1-p_{\boldsymbol{\theta}}(s)+\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}\right)-B(S^k)\right). \tag{126}$$

For the normalized version, we use the normalization $W(S^k)=\sum_{s\in S^k}\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}$ for the baseline, and $W_i(S^k)=W(S^k)-\frac{p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}+p_{\boldsymbol{\theta}}(s)$ to normalize the individual terms:

$$\nabla_{\boldsymbol{\theta}}\mathbb{E}_{y\sim p_{\boldsymbol{\theta}}(y)}\left[f(y)\right]\approx\sum_{s\in S^k}\frac{1}{W_i(S^k)}\cdot\frac{\nabla_{\boldsymbol{\theta}}p_{\boldsymbol{\theta}}(s)}{q_{\boldsymbol{\theta},\kappa}(s)}\left(f(s)-\frac{B(S^k)}{W(S^k)}\right). \tag{127}$$

It seems odd to normalize the terms in the outer sum by $\frac{1}{W_i(S^k)}$ instead of $\frac{1}{W(S^k)}$, but equation 127 can be rewritten into a form similar to equation 63, i.e. with a different baseline for each sample, but this form is more convenient for implementation (Kool et al., 2019b).

D.5 THE UNORDERED SET POLICY GRADIENT ESTIMATOR

D.5.1 Proof of unbiasedness of the unordered set policy gradient estimator with baseline

To prove the unbiasedness of result we need to prove that the control variate has expectation 0:

Lemma 18.

$$\mathbb{E}_{S^k \sim p_{\boldsymbol{\theta}}(S^k)} \left[\sum_{s \in S^k} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) R(S^k, s) \sum_{s' \in S^k} p_{\boldsymbol{\theta}}(s') R^{D \setminus \{s\}}(S^k, s') f(s') \right] = 0. \tag{128}$$

Proof. Similar to equation 55, we apply Bayes' Theorem conditionally on $b_1 = s$ to derive for $s' \neq s$

$$\begin{aligned} P(b_2 = s' | S^k, b_1 = s) &= \frac{P(S^k | b_2 = s', b_1 = s) P(b_2 = s' | b_1 = s')}{P(S^k | b_1 = s)} \\ &= \frac{p_{\boldsymbol{\theta}}^{D \setminus \{s, s'\}}(S^k \setminus \{s, s'\}) p_{\boldsymbol{\theta}}^{D \setminus \{s\}}(s')}{p_{\boldsymbol{\theta}}^{D \setminus \{s\}}(S^k \setminus \{s\})} \\ &= \frac{p_{\boldsymbol{\theta}}(s')}{1 - p_{\boldsymbol{\theta}}(s)} R^{D \setminus \{s\}}(S^k, s'). \end{aligned} \tag{129}$$

For $s' = s$ we have $R^{D \setminus \{s\}}(S^k, s') = 1$ by definition, so using equation 129 we can show that

$$\begin{aligned} &\sum_{s' \in S^k} p_{\boldsymbol{\theta}}(s') R^{D \setminus \{s\}}(S^k, s') f(s') \\ &= p_{\boldsymbol{\theta}}(s) f(s) + \sum_{s' \in S^k \setminus \{s\}} p_{\boldsymbol{\theta}}(s') R^{D \setminus \{s\}}(S^k, s') f(s') \\ &= p_{\boldsymbol{\theta}}(s) f(s) + (1 - p_{\boldsymbol{\theta}}(s)) \sum_{s' \in S^k \setminus \{s\}} \frac{p_{\boldsymbol{\theta}}(s')}{1 - p_{\boldsymbol{\theta}}(s)} R^{D \setminus \{s\}}(S^k, s') f(s') \\ &= p_{\boldsymbol{\theta}}(s) f(s) + (1 - p_{\boldsymbol{\theta}}(s)) \sum_{s' \in S^k \setminus \{s\}} P(b_2 = s' | S^k, b_1 = s) f(s') \\ &= p_{\boldsymbol{\theta}}(s) f(s) + (1 - p_{\boldsymbol{\theta}}(s)) \mathbb{E}_{b_2 \sim p_{\boldsymbol{\theta}}(b_2 | S^k, b_1 = s)} [f(b_2)] \\ &= \mathbb{E}_{b_2 \sim p_{\boldsymbol{\theta}}(b_2 | S^k, b_1 = s)} [p_{\boldsymbol{\theta}}(b_1) f(b_1) + (1 - p_{\boldsymbol{\theta}}(b_1)) f(b_2)]. \end{aligned}$$

Now we can show that the control variate is actually the result of Rao-Blackwellization:

$$
\begin{aligned}
&\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\sum_{s\in S^k}\nabla_{\boldsymbol{\theta}}p_{\boldsymbol{\theta}}(s)R(S^k,s)\sum_{s'\in S^k}p_{\boldsymbol{\theta}}(s')R^{D\setminus\{s\}}(S^k,s')f(s')\right]\\
&=\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\sum_{s\in S^k}p_{\boldsymbol{\theta}}(s)R(S^k,s)\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(s)\sum_{s'\in S^k}p_{\boldsymbol{\theta}}(s')R^{D\setminus\{s\}}(S^k,s')f(s')\right]\\
&=\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\sum_{s\in S^k}P(b_1=s|S^k)\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(s)\sum_{s'\in S^k}p_{\boldsymbol{\theta}}(s')R^{D\setminus\{s\}}(S^k,s')f(s')\right]\\
&=\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1|S^k)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\sum_{s'\in S^k}p_{\boldsymbol{\theta}}(s')R^{D\setminus\{b_1\}}(S^k,s')f(s')\right]\right]\\
&=\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1|S^k)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\mathbb{E}_{b_2\sim p_{\boldsymbol{\theta}}(b_2|S^k,b_1)}\left[p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right]\right]\right]\\
&=\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\mathbb{E}_{B^k\sim p_{\boldsymbol{\theta}}(B^k|S^k)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\left(p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right)\right]\right]\\
&=\mathbb{E}_{B^k\sim p_{\boldsymbol{\theta}}(B^k)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\left(p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right)\right]
\end{aligned}
$$

This expression depends only on b_1 and b_2 and we recognize the stochastic sum-and-sample estimator for $k=2$ used as 'baseline'. As a special case of equation 59 for $C=\{b_1\}$, we have

$$
\mathbb{E}_{b_2\sim p_{\boldsymbol{\theta}}(b_2|b_1)}\left[\left(p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right)\right]=\mathbb{E}_{i\sim p_{\boldsymbol{\theta}}(i)}\left[f(i)\right]. \tag{130}
$$

Using this, and the fact that $\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\right]=\nabla_{\boldsymbol{\theta}}\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1)}\left[1\right]=\nabla_{\boldsymbol{\theta}}1=0$ we find

$$
\begin{aligned}
&\mathbb{E}_{S^k\sim p_{\boldsymbol{\theta}}(S^k)}\left[\sum_{s\in S^k}\nabla_{\boldsymbol{\theta}}p_{\boldsymbol{\theta}}(s)R(S^k,s)\sum_{s'\in S^k}p_{\boldsymbol{\theta}}(s')R^{D\setminus\{s\}}(S^k,s')f(s')\right]\\
&=\mathbb{E}_{B^k\sim p_{\boldsymbol{\theta}}(B^k)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\left(p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right)\right]\\
&=\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\mathbb{E}_{b_2\sim p_{\boldsymbol{\theta}}(b_2|b_1)}\left[\left(p_{\boldsymbol{\theta}}(b_1)f(b_1)+(1-p_{\boldsymbol{\theta}}(b_1))f(b_2)\right)\right]\right]\\
&=\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\mathbb{E}_{x\sim p_{\boldsymbol{\theta}}(x)}\left[f(x)\right]\right]\\
&=\mathbb{E}_{b_1\sim p_{\boldsymbol{\theta}}(b_1)}\left[\nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(b_1)\right]\mathbb{E}_{x\sim p_{\boldsymbol{\theta}}(x)}\left[f(x)\right]\\
&=0\cdot\mathbb{E}_{x\sim p_{\boldsymbol{\theta}}(x)}\left[f(x)\right]\\
&=0
\end{aligned}
$$

□

D.6 THE RISK ESTIMATOR

D.6.1 Proof of built-in baseline

We show that the RISK estimator, taking gradients through the normalization factor actually has a built-in baseline. We first use the log-derivative trick to rewrite the gradient of the ratio as the ratio times the logarithm of the gradient, and then swap the summation variables in the double sum that arises:

$$
\begin{aligned}
e^{\text{RISK}}(S) &= \sum_{s\in S} \nabla_{\boldsymbol{\theta}} \left(\frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \right) f(s) \\
&= \sum_{s\in S} \frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \nabla_{\boldsymbol{\theta}} \log \left(\frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \right) f(s) \\
&= \sum_{s\in S} \frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \left(\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(s) - \nabla_{\boldsymbol{\theta}} \log \sum_{s'\in S} p_{\boldsymbol{\theta}}(s') \right) f(s) \\
&= \sum_{s\in S} \frac{p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \left(\frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s)}{p_{\boldsymbol{\theta}}(s)} - \frac{\sum_{s'\in S} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s')}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \right) f(s) \\
&= \sum_{s\in S} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) f(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} - \frac{\sum_{s,s'\in S} p_{\boldsymbol{\theta}}(s) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s') f(s)}{\left(\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')\right)^2} \\
&= \sum_{s\in S} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) f(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} - \frac{\sum_{s,s'\in S} p_{\boldsymbol{\theta}}(s') \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s) f(s')}{\left(\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')\right)^2} \\
&= \sum_{s\in S} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s)}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \left(f(s) - \frac{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s') f(s')}{\sum_{s'\in S} p_{\boldsymbol{\theta}}(s')} \right) \\
&= \sum_{s\in S} \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(s)}{\sum_{s''\in S} p_{\boldsymbol{\theta}}(s'')} \left(f(s) - \sum_{s'\in S} \frac{p_{\boldsymbol{\theta}}(s')}{\sum_{s''\in S} p_{\boldsymbol{\theta}}(s'')} f(s') \right).
\end{aligned}
$$

D.7 CATEGORICAL VARIATIONAL AUTO-ENCODER

D.7.1 Experimental details

We use the code[1] by Yin et al. (2019) to reproduce their categorical VAE experiment, of which we include details here for self-containment. The dataset is MNIST, statically binarized by thresholding at 0.5 (although we include results using the standard binarized dataset by Salakhutdinov and Murray (2008) and Larochelle and Murray (2011) in Section D.7.2). The latent representation $\boldsymbol{z}$ is $K = 20$ dimensional with $C = 10$ categories per dimension with a uniform prior $p(z_k = c) = 1/C, k = 1, ..., K$. The encoder is parameterized by $\boldsymbol{\phi}$ as $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \prod_k q_{\boldsymbol{\phi}}(z_k|\boldsymbol{x})$ and has two fully connected hidden layers with 512 and 256 hidden nodes respectively, with LeakyReLU ($\alpha = 0.1$) activations. The decoder, parameterized by $\boldsymbol{\theta}$, is given by $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = \prod_i p_{\boldsymbol{\theta}}(x_i|\boldsymbol{z})$, where $x_i \in \{0, 1\}$ are the pixel values, and has fully connected hidden layers with 256 and 512 nodes and LeakyReLU activation.

1 https://github.com/ARM-gradient/ARSM

ELBO OPTIMIZATION The evidence lower bound (ELBO) that we optimize is given by

$$\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) + \ln p(z) - \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) \right] \tag{131}$$

$$= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) \right] - KL(q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) || p(z)) . \tag{132}$$

For the decoder parameters $\boldsymbol{\theta}$, since $q_{\boldsymbol{\phi}}(z|\boldsymbol{x})$ does not depend on $\boldsymbol{\theta}$, it follows that

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) \right] . \tag{133}$$

For the encoder parameters $\boldsymbol{\phi}$, we can write $\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta})$ using equation 132 and equation 65 as

$$\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) \ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) \right] - \nabla_{\boldsymbol{\phi}} KL(q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) || p(z)) . \tag{134}$$

This assumes we can compute the KL divergence analytically. Alternatively, we can use a sample estimate for the KL divergence, and use equation 131 with equation 65 to obtain

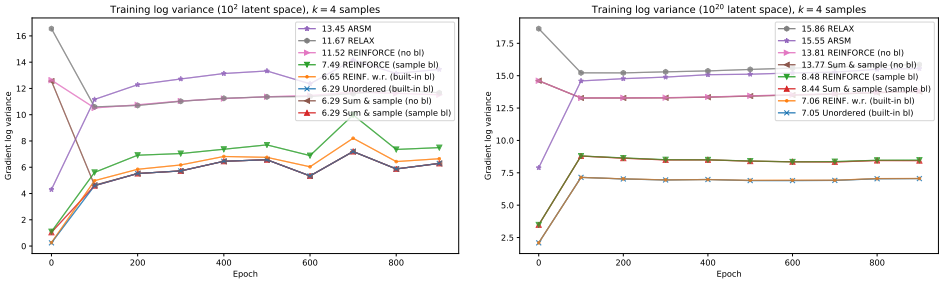
$$\nabla_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) (\ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) + \ln p(z) - \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x})) + \nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) \right] \tag{135}$$

$$= \mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) (\ln p_{\boldsymbol{\theta}}(\boldsymbol{x}|z) - \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x})) \right] . \tag{136}$$

Here we have left out the term $\mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) \right] = 0$, similar to Roeder et al. (2017), and, assuming a uniform (i.e. constant) prior $\ln p(z)$, the term $\mathbb{E}_{z \sim q_{\boldsymbol{\phi}}(z|\boldsymbol{x})} \left[\nabla_{\boldsymbol{\phi}} \ln q_{\boldsymbol{\phi}}(z|\boldsymbol{x}) \ln p(z) \right] = 0$. With a built-in baseline, this second term cancels out automatically, even if it is implemented. Despite the similarity of the equation 131 and equation 132, their gradient estimates (equation 135 and equation 134) are structurally dissimilar and care should be taken to implement the REINFORCE estimator (or related estimators such as ARSM and the unordered set estimator) correctly using automatic differentiation software. Using Gumbel-Softmax and RELAX, we take gradients 'directly' through the objective in equation 132.

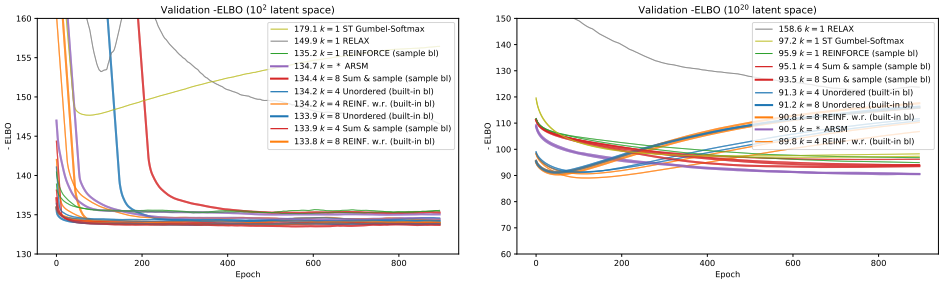
We optimize the ELBO using the analytic KL for 1000 epochs using the Adam (Kingma and Ba, 2015) optimizer. We use a learning rate of 10^{-3} for all estimators except Gumbel-Softmax and RELAX, which use a learning rate of 10^{-4} as we found they diverged with a higher learning rate. For ARSM, as an exception we use the sample KL, and a learning rate of $3 \cdot 10^{-4}$, as suggested by the authors. All reported ELBO values are computed using the analytic KL. Our code is publicly available[2].

2 https://github.com/wouterkool/estimating-gradients-without-replacement



(a) Small domain (latent space size 10^2) **(b)** Large domain (latent space size 10^{20})

Figure 32: Gradient log variance of different unbiased estimators with $k = 4$ samples, estimated every 100 (out of 1000) epochs while training using REINFORCE with replacement. Each estimator is computed 1000 times with different latent samples for a fixed minibatch (the first 100 records of training data). We report (the logarithm of) the sum of the variances per parameter (trace of the covariance matrix). Some lines coincide, so we sort the legend by the last measurement and report its value.



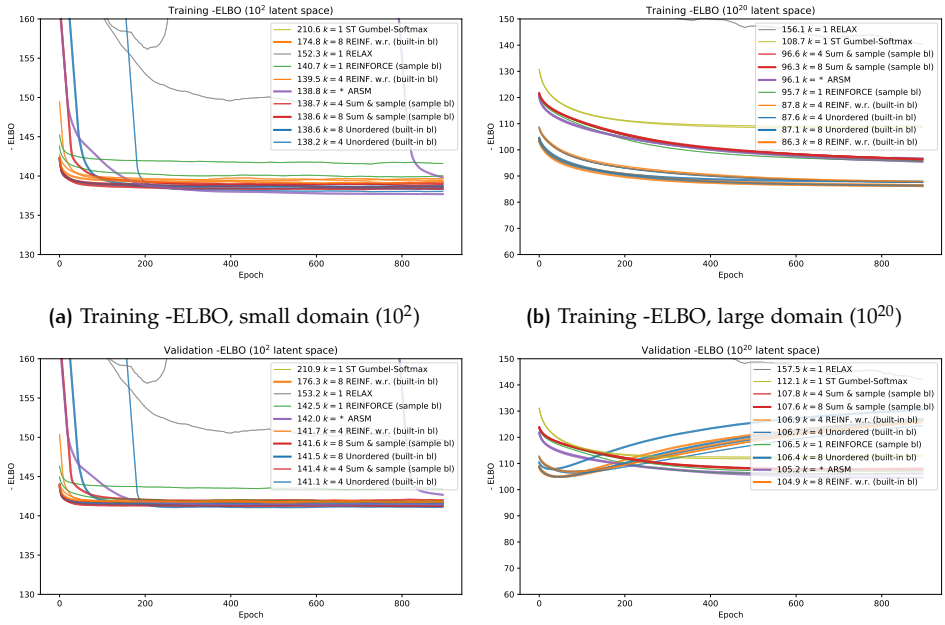
(a) Small domain (latent space size 10^2) **(b)** Large domain (latent space size 10^{20})

Figure 33: Smoothed validation -ELBO curves during training of two independent runs when with different estimators with $k = 1$, 4 or 8 (thicker lines) samples (ARSM has a variable number). Some lines coincide, so we sort the legend by the lowest -ELBO achieved and report this value.

D.7.2 Additional results

GRADIENT VARIANCE DURING TRAINING We also evaluate gradient variance of different estimators during different stages of training. We measure the variance of different estimators with $k = 4$ samples during training with REINFORCE with replacement, such that all estimators are computed for the same model parameters. The results during training, given in Figure 32, are similar to the results for the trained model in Table 5, except for at the beginning of training, although the rankings of different estimator are mostly the same.

NEGATIVE ELBO ON VALIDATION SET Figure 33 shows the -ELBO evaluated during training on the validation set. For the large latent space, we see validation error quickly increase (after reaching a minimum) which is likely because of overfitting (due to improved optimization), a phenomenon observed before (Tucker et al., 2017; Grathwohl et al., 2018). Note that before the overfitting starts, both REINFORCE without replacement and the unordered set estimator achieve a validation error similar to the other estimators, such that in a practical setting, one can use early stopping.



(a) Training -ELBO, small domain (10^2)

(b) Training -ELBO, large domain (10^{20})

(c) Validation -ELBO, small domain (10^2)

(d) Validation -ELBO, large domain (10^{20})

Figure 34: Smoothed training and validation -ELBO curves during training on the standard binarized MNIST dataset (Salakhutdinov and Murray, 2008; Larochelle and Murray, 2011) of two independent runs when with different estimators with $k = 1, 4$ or 8 (thicker lines) samples (ARSM has a variable number). Some lines coincide, so we sort the legend by the lowest -ELBO achieved and report this value.

RESULTS USING STANDARD BINARIZED MNIST DATASET Instead of using the MNIST dataset binarized by thresholding values at 0.5 (as in the code and paper by Yin et al. (2019)) we also experiment with the standard (fixed) binarized dataset by Salakhutdinov and Murray (2008) and Larochelle and Murray (2011), for which we plot train and validation curves for two runs on the small and large domain in Figure 34. This gives more realistic (higher) -ELBO scores, although we still observe the effect of overfitting. As this is a bit more unstable setting, one of the runs using REINFORCE with replacement diverged, but in general the relative performance of estimators is similar to using the dataset with 0.5 threshold.

D.8 TRAVELLING SALESMAN PROBLEM

We train the exact same attention model as in Chapter 3, and minimize the expected length of a tour predicted by the model, using different gradient estimators. We did not do any hyperparameter optimization and used the exact same training details, using the Adam optimizer (Kingma and Ba, 2015) with a learning rate of 10^{-4} (no decay) for 100 epochs for all estimators. For the baselines, we used the same batch size of 512, but for estimators that use $k = 4$ samples, we used a batch size of $\frac{512}{4} = 128$ to compensate for the additional samples (this makes multi-sample methods actually faster since the encoder still needs to be evaluated only once).