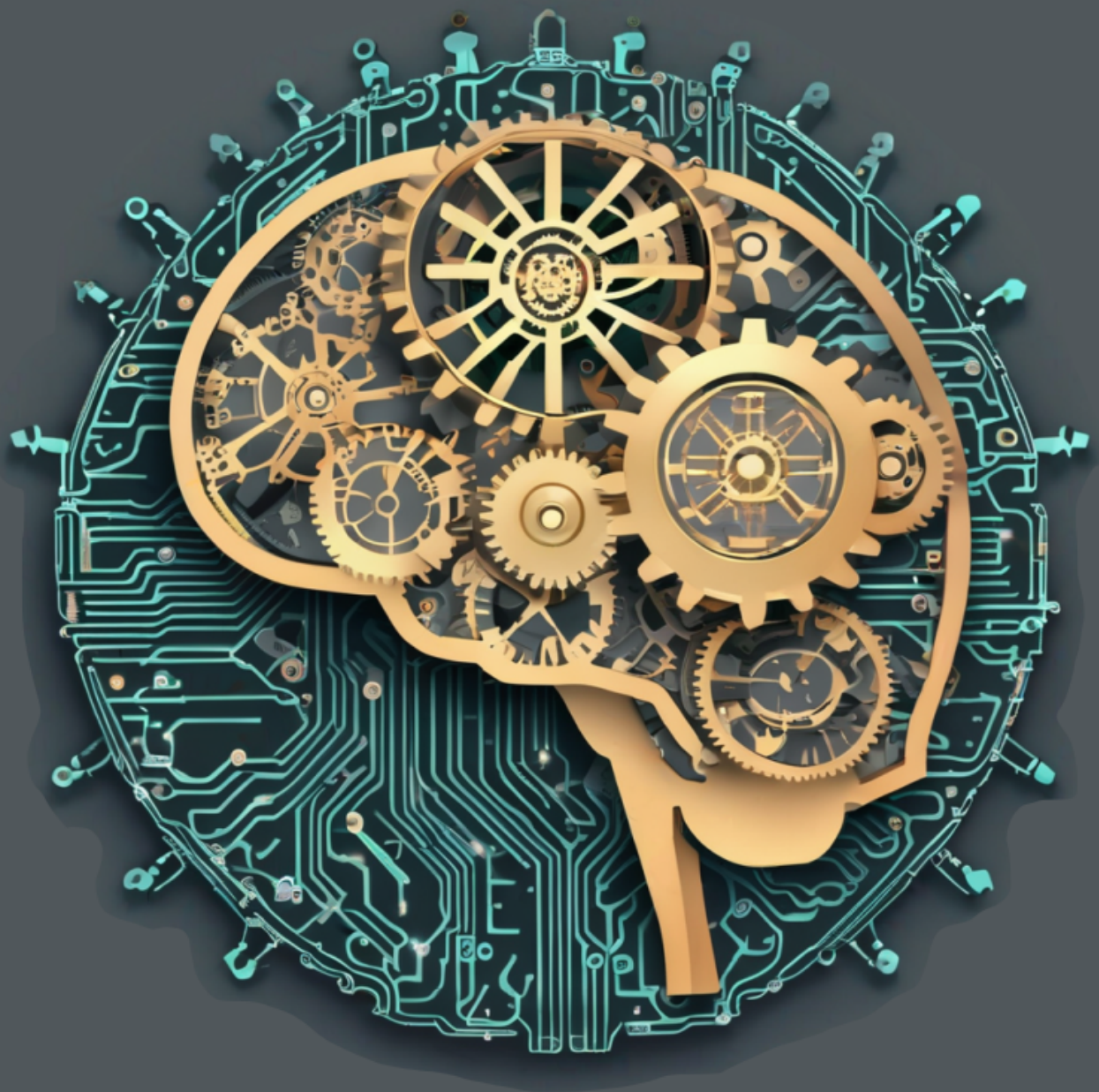
Max Benedikt Paulus

Learning with and for discrete optimization



Diss. ETH No. 29328

DISS. ETH NO. 29328

LEARNING WITH AND FOR DISCRETE OPTIMIZATION

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

MAX B. PAULUS
MSc ETH Statistics, ETH Zurich
born on 26 December 1993

accepted on the recommendation of

Prof. Dr. Andreas Krause
Prof. Dr. Daniel Tarlow
Prof. Dr. Tamir Hazan
Dr. Jan Poland

2023

Max B. Paulus: *Learning with and for discrete optimization,* © 2023

DOI: 10.3929/ethz-b-000629004

To Clara

ABSTRACT

Machine learning and discrete optimization are pillars of computer science and both are widely used tools for analysis, prediction and decision-making across business, science and technology. However, the premises on which machine learning and discrete optimization methods are developed differ fundamentally. Learning relies on data and often requires little if any manual design. Its strengths are generalization and near universal applicability, but many models cannot effectively integrate domain knowledge or specific constraints, lack interpretability and their predictions are uncertain which hinders adoption in practice. Conversely, algorithms for discrete optimization are usually tailored to specific applications, such as combinatorial problems. Their precise formulation affords insight and analysis and their outputs often come with performance guarantees. However, in contrast to machine learning, methods in discrete optimization do not generalize between instances, which is a deficiency in practical applications.

In light of the complementary strengths and weaknesses of machine learning and discrete optimization, it is natural to ask to what extent methods from these two areas can be fruitfully combined. This is the question we ask in this thesis and that we answer affirmatively by presenting methods for learning *with* and *for* discrete optimization.

In learning with discrete optimization, we focus on gradient estimation for models in non-supervised learning that involve discrete variables. Such models are widespread and provide benefits in terms of regularization, interpretability, model design and algorithmic integration. We rely on efficient methods from discrete optimization to design new gradient estimators for these models via relaxations and demonstrate experimentally that they make learning more performant, useful and efficient.

In learning for discrete optimization, we focus on improving the performance of branch and bound solvers for integer programming with machine learning. We replace existing subroutines for cutting plane selection and diving in these solvers with learnt models that are tailored to specific applications. Our methods draw on ideas from imitation learning and generative modeling, they are scalable and effective. In a range of experiments, our models outperform existing heuristics as well as competing machine learning approaches to facilitate overall improvements in solver performance.

ZUSAMMENFASSUNG

Maschinelles Lernen und diskrete Optimierung sind Grundpfeiler der Informatik. Beide werden zur Analyse und Entscheidungsfindung in Wirtschaft, Wissenschaft und Technik eingesetzt. Die Prämissen, auf denen ihre Methoden beruhen, unterscheiden sich jedoch grundlegend. Das maschinelle Lernen stützt sich hauptsächlich auf Daten, was Generalisierung auf unbekannte Instanzen und eine nahezu universellen Anwendbarkeit ermöglicht. Aber viele Modelle können spezifische Einschränkungen nicht effektiv integrieren, sind nicht interpretierbar und liefern unsichere Vorhersagen, was ihren praktischen Einsatz erschwert. Umgekehrt sind Algorithmen für die diskrete Optimierung in der Regel auf spezifische Anwendungen zugeschnitten, zum Beispiel auf kombinatorische Probleme. Ihre präzise Formulierung ermöglicht ein besseres Verständnis und ihre Ergebnisse sind oft mit Garantien verbunden. Im Gegensatz zum maschinellen Lernen behandelt die diskrete Optimierung aber neue Probleminstanzen isoliert, was in der Praxis ein grosses Manko darstellt.

Angesichts ihrer komplementären Stärken und Schwächen stellen wir in dieser Arbeit die Frage, ob sich Methoden des maschinellen Lernens und der diskreten Optimierung kombinieren lassen, um ihre jeweiligen Schwächen auszugleichen. Wir beantworten diese Frage positiv, indem wir Methoden des Lernens *mit* und *für* die diskrete Optimierung vorstellen.

Beim Lernen mit diskreter Optimierung konzentrieren wir uns auf die Gradientenschätzung in Modellen für nicht-überwachtes Lernen mit diskreten Variablen. Solche Modelle bieten Vorteile in Bezug auf Regularisierung, Interpretierbarkeit, Modelldesign und algorithmische Integration. Wir stützen uns auf effiziente Methoden aus der diskreten Optimierung, um neue Gradientenschätzer für diese Modelle mit Hilfe von Relaxationen zu entwerfen, und zeigen experimentell, dass sie das Lernen besser, nützlicher und effizienter machen.

Beim Lernen für die diskrete Optimierung verbessern wir das Lösungsverfahren *branch and bound* für die ganzzahlige Programmierung mittels maschinellen Lernens. Wir ersetzen bestehende Subroutinen dieses Verfahrens für die Auswahl von Schnittebenen und Tauchheuristiken durch gelernte Modelle, die auf spezifischen Anwendungen trainiert werden. In einer Reihe von Experimenten zeigen wir, dass unsere Methoden skalierbar und effektiv sind.

ACKNOWLEDGEMENTS

I am grateful to my advisor, Andreas Krause, for providing me with the opportunity to pursue research in machine learning at ETH Zürich. Andreas' unconditional support and generosity have allowed me to pursue my research with confidence, enthusiasm and liberty. I am grateful for his patience, encouragement and guidance, which has helped me overcome the challenges of conducting scientific research.

I am immensely grateful to my advisor, Chris J. Maddison. Chris has been a magnificent mentor and close collaborator throughout my PhD studies. He has been a profound influence on my growth as both a researcher and an individual. His exceptional creativity, curiosity, relentlessness, and intellect have continuously inspired me. I am indebted to Chris for his trust in me as a young doctoral student and for the incredible opportunity to work with and learn from him. Visiting Chris in Oxford and Princeton for research were undoubtedly the highlights of my doctoral studies.

I am grateful to Tamir Hazan, Daniel Tarlow, and Jan Poland for serving on my doctoral committee and providing feedback on my work. Additionally, I am deeply grateful to all the individuals with whom I had the pleasure of collaborating on research over the past years. Among them, I am especially thankful to Dami Choi and Giulia Zarpellon for their close collaboration and invaluable contributions to the research presented in this thesis.

I also want to thank all the members of the Learning and Adaptive Systems group, the Institute for Machine Learning, ETH Zürich, the University of Toronto and the Max Planck Institute for Intelligent Systems with whom I have had the pleasure of interacting for work or leisure. I am especially grateful to Philippe Wenk, Andisheh Amrollahi, Aytunç Şahin, Djordje Miladinovic, Robin Geyer, Imant Daunhawer, Chris Wendler, Pashootan Vaezipoor, Tobias Wekhof, and Sascha Langenbach for becoming good friends over the years and sharing the joys and misery of graduate studentship.

Finally, I gratefully acknowledge support from the Max Planck ETH Center for Learning Systems and the Sustainable Chemical Processes through Catalysis (Suchcat) National Center of Competence in Research (NCCR) for funding part of my research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

NOTATION

GENERAL

We reserve calligraphic notation to denote sets, but for some sets including e.g., the real numbers $\mathbb{R}$ or a polyhedron $P \subset \mathbb{R}^n$ we adhere to standard notation. We denote matrices and vectors in bold font, e.g., $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ or $\boldsymbol{b} \in \mathbb{R}^n$, and scalar quantities in plain font, e.g., the temperature parameter τ in chapter 3. We always write random variables in upper-case letters, but upper-case letters may also denote functions, such as the loss function $L(\cdot)$, or matrices as above, depending on the context. We often use an upper-level

asterisk to denote the solution of a relaxation, e.g., $\boldsymbol{X}^*$ is the solution of the stochastic softmax relaxation in chapter 3 and z^* is the solution to the linear program relaxation in chapters 6 and 7. Similarly, we sometimes use an upper-level star to denote the optimal solution of a discrete optimization problem, e.g., $z^\star$ or $\boldsymbol{x}^\star$ for the solution of the integer program in chapter 5, but may omit the symbol in some cases to avoid notational clutter. We summarize the most important symbols in the table below.

FREQUENTLY USED SYMBOLS

SYMBOL	MEANING
$\mathcal{X}$	a discrete set containing the embeddings of a finite collection of objects or the feasible solutions of an integer program
$\boldsymbol{X} \sim q_{\boldsymbol{\theta}}$	a discrete random variable with distribution $q_{\boldsymbol{\theta}}$ whose domain is $\mathcal{X}$ and whose parameters are θ
$\boldsymbol{x}^\star \in \mathcal{X}$	an optimal solution to a discrete optimization problem with domain $\mathcal{X}$
$P \subset \mathbb{R}^n$	a continuous set which may be the convex hull of $\mathcal{X}$ or the feasible solutions of a linear program
$\boldsymbol{X}^* \sim q^*_{\boldsymbol{\theta}}$	a continuous random variable with distribution $q^*_{\boldsymbol{\theta}}$ whose domain is P and whose parameters are θ
$\boldsymbol{x}^* \in P$	an optimal solution to the relaxed optimization problem with domain P
$\boldsymbol{\theta}$	the learnable parameters of the distribution $q_{\boldsymbol{\theta}}$ for which no direct supervision is available
$\boldsymbol{\eta}$	the gradient of the learnable parameters $\boldsymbol{\theta}$ as defined in equation (2.3)
$\hat{\boldsymbol{\eta}}$	a gradient estimator, i.e, an estimator for $\boldsymbol{\eta}$

1

INTRODUCTION

Machine learning and discrete optimization are pillars of computer science. Both are widely used tools for analysis, prediction and decision-making across business, technology and science. The goal of machine learning is to develop methods that can independently identify patterns in data and make accurate predictions. Over the last decade machine learning has progressed tremendously. Today, modern systems are capable of learning from large amounts of data and successfully accomplish many diverse tasks. For example, machine learning is routinely used to process visual data, for example to reliably detect objects [1–3], for visual tracking [4, 5], semantic segmentation [6, 7] or image restoration [8]. Generative models for vision are capable of creating realistic and artistic images from few visual cues [9] or free text descriptions [10, 11]. Large models for natural language have recently demonstrated unprecedented capability. Apart from accomplishing canonical task, such as sentiment prediction [12], natural language inference [13], or reading comprehension [14], these models successfully learn to answer questions [15], generate summaries [16] or are capable of free dialogue [17]. Beyond, machine learning has aided the development of systems that defeat professional players in strategic games, such as Go [18] and poker [19], predict protein structures with atomic accuracy [20] or even assist humans in formulating and proving mathematical conjectures [21].

The goal of discrete optimization is to develop efficient methods for solving problems that involve discrete decisions. Such problems are ubiquitous. They arise in the design of reliable networks, for example in communication, transportation or energy, the allocation of resources and scheduling of tasks in various production processes and in a wide variety of combinatorial problems. Applications of discrete optimization are countless and methods for discrete optimization are widely used, from staffing airline personnel [22], to balancing server loads in distributed computing or to laying out printed circuit boards in electronics [23], to understanding gene expressions in computational biology [24]. Curiously, there are some direct applications of discrete optimization in machine learning, too, including in probabilistic inference [25, 26] or the verification of neural networks [27, 28].

The propositions on which methods in machine learning and discrete optimization are developed differ fundamentally. Methods in machine

learning rely on data and require little if any manual design to create powerful predictive models that generalize well to unseen instances and sometimes even across tasks [29]. Most methods in machine learning are widely applicable or can be easily adapted to a large number of diverse problems. In fact, the dominant method, i.e., learning deep neural networks with variants of gradient descent, is used universally (and nearly exclusively) across computer vision, natural language processing and many other areas of application in machine learning. However, the versatility and performance of these methods comes at a cost. Many methods, in particular neural networks, are opaque and it is difficult to interpret their predictions. The outputs of these models lack guarantees and probabilistic calibration can be brittle. In cases, where it is desirable, it is difficult to effectively integrate domain knowledge. To illustrate, consider training a convolutional neural network with stochastic gradient descent to predict sentiment ratings from free text. This machine learning method is versatile; it could easily be used in another domain or on other data, for example to train models that make diagnostic predictions from medical records [30] or with simple modifications from magnetic resonance imaging [31]. However, in either case the model is black-box which hinders adoption in practice. A user will find it more difficult to base decisions on the model predictions, when he cannot understand what the prediction was based on, for example which parts of the medical record or magnetic resonance image. In addition, it is not clear how to best integrate domain knowledge, for example it may be desirable to base predictions on key contiguous chunks of test or on lesions of certain shape. This is essential in many real-world applications where obtaining data for learning may be expensive.

In contrast, methods in discrete optimization are typically algorithms tailored to specific applications. Their design is often entirely manual and relies on human ingenuity to derive abstract formulations of a given problem. For example, a number of algorithms have been designed to solve combinatorial problems, such as Kruskal's algorithm to find the best spanning tree in a graph [32], or the Hungarian algorithm to find optimal matchings [33]. Even general-purpose methods for discrete optimization, such as branch and bound search [34] can often significantly be improved, if they are tailored to specific applications (Table 7.4 in chapter 7). Algorithms, by definition, prescribe exact and exhaustive rules for computation. These permit rigorous analysis and afford interpretability, unlike most models in machine learning. Methods in discrete optimization are known to produce exact or approximate solutions and guarantees on their performance, runtime or

memory requirements are usually known. For example, Kruskal's algorithm is known to find the optimal solution, greedy algorithms for submodular minimization are guaranteed to approximate the optimal solution well and scale gracefully [35]. Branch and bound search is an exact method, but can be terminated prematurely to yield lower and upper bounds on the optimal solution. In contrast to machine learning, problem instances in discrete optimization are traditionally treated in isolation. When given a new problem instance, algorithms in discrete optimization typically cannot leverage any information from previous solves, and instead follow the exact same set of rules as on previous instances. This is unlike to machine learning, where the model uses information acquired during training to make predictions at test time. It poses a serious deficiency in practical applications, where often many similar problem instances must be solved and structural commonality exists between them. For example, consider using discrete optimization to optimize server loads in a distributed computing cluster at an hourly rate. It is reasonable to expect that problem instances from different days or times are similar, because they share the same servers and certain compute loads may regularly occur. Thus, plausibly information from solving previous instances may be used to improve any discrete optimization method for solving new server load balancing problems. But methods such as branch and bound will build a new search tree from scratch for every new problem instance and disregard any previous solves.

In light of these observations and the complementary strengths and weaknesses of machine learning and discrete optimization, it is natural to ask to what extent methods from these two areas can be fruitfully combined. Can we use discrete optimization to design machine learning models that are more interpretable or to integrate domain knowledge? Can we use machine learning models to improve discrete optimization in practical application without sacrificing optimality guarantees? And above all, how do we learn such models? These are the question we ask in this thesis and that we answer affirmatively by presenting methods for *learning with and for discrete optimization.*

LEARNING WITH DISCRETE OPTIMIZATION The first part of this thesis concentrates on learning *with* discrete optimization. Our focus is on non-supervised machine learning which includes problems from unsupervised, self-supervised and reinforcement learning. In non-supervised learning, supervision is limited or may not exist at all and therefore the design of probabilistic models is common. In chapter 2 we review the use

of discrete variables in these models and argue that they can facilitate regularization, render models more interpretable and present opportunities for model design and algorithmic integration of domain knowledge. The main difficulty in learning these models is the estimation of gradients and unfortunately only few estimators for models with discrete variables are known. A key contribution of this thesis is the design of new gradient estimators for discrete variables in non-supervised machine learning. Naturally, discrete variables are closely tied to methods of discrete optimization where combinatorial objects appear frequently. In chapter 3, we use this to pose distributions over discrete variables using the perturbation model framework [36]. These variables can be efficiently sampled with the aid of discrete optimization routines and naturally facilitate gradient estimation via continuous relaxations. Our work offers a unified perspective on existing relaxed gradient estimators and contains many new gradient estimators. In chapter 4 we extend this work by introducing another class of gradient estimators that addresses some shortcomings of relaxed gradient estimation and offers improved estimation properties. Experimentally, we demonstrate that our gradient estimators facilitate learning models with discrete variables variables that are more performant, useful and efficient.

LEARNING FOR DISCRETE OPTIMIZATION The second part of this thesis addresses learning *for* discrete optimization. Our focus is on branch and bound search. This is a general-purpose algorithm for integer programming and can be used to solve a wide variety of discrete optimization problems. Modern implementations of branch and bound search involve many heuristic subroutines that guide the search and have a significant influence on the overall performance of the solver. In chapter 5 we argue that this presents an opportunity for machine learning. Existing heuristics designed by domain experts for universal use can be replaced by learnt models that are tailored to specific applications with the goal of improving solver performance. We identify two key subroutines in branch and bound search, the selection of cutting planes and the use of primal heuristics, and address both with machine learning. In chapter 6 we present *NeuralCut*, a method to learn models for cutting plane selection via imitation learning. Our models can be used inside branch and bound to select better cuts than existing heuristics and competing approaches based on reinforcement learning, they facilitate improvements in branch and bound search. In chapter 7 we present *L2Dive* to learn application-specific diving heuristics for branch and bound. *L2Dive* learns a generative model for the solution of a given integer program, the

predictions of the model can be used to conduct guided dives that reliably yield better solutions than existing diving heuristics. *L2Dive* improves overall solver performance in real-world applications and even outperforms a tuned ensemble of existing divers.

1.1 THESIS ORGANIZATION

The remainder of this thesis is organized into the following chapters.

- *Chapter 2* introduces non-supervised machine learning, highlights the opportunities that the use of discrete variables in non-supervised models affords and reviews existing methods to estimate gradients in these models.
- *Chapter 3* presents gradient estimation with stochastic softmax tricks, a unified framework to design gradient estimators for structured discrete variables.
- *Chapter 4* presents a simple but effective method to improve gradient estimation properties for straight-through variants of relaxed gradient estimators.
- *Chapter 5* reviews integer programming, branch and bound search and the role of cutting planes, primal heuristics and machine learning.
- *Chapter 6* presents *NeuralCut*, a framework to learn models for cutting plane selection that can be used to select better cuts for branch and bound search.
- *Chapter 7* presents *L2Dive*, a framework to learn application-specific diving heuristics that improve the performance branch and bound solvers.
- *Chapter 8* concludes the thesis with a summary of the findings and outlines several directions for future work.

1.2 LIST OF PUBLICATIONS

The thesis is based on the following publications.

1. Paulus, M. B. *et al. Gradient Estimation with Stochastic Softmax Tricks* in *Advances in Neural Information Processing Systems* (2020)

2. Paulus, M. B., Maddison, C. J. & Krause, A. *Rao-Blackwellizing the Straight-Through Gumbel-Softmax Gradient Estimator* in *International Conference on Learning Representations* (2021)

3. Paulus, M. B. *et al. Learning to Cut by Looking Ahead: Cutting Plane Selection via Imitation Learning* in *Proceedings of the 39th International Conference on Machine Learning* (2022)

4. Paulus, M. B. & Krause, A. *Learning To Dive In Branch and Bound* in *Under submission.* (2023)

The following work is relevant and related, but it was not directly included into the thesis.

5. Valentin, R., Ferrari, C., Scheurer, J., Amrollahi, A., Wendler, C. & Paulus, M. B. Instance-wise algorithm configuration with graph neural networks. *NeurIPS Machine Learning for Combinatorial Optimization Competition* (2021)

6. Huijben, I. A., Kool, W., Paulus, M. B. & Van Sloun, R. J. *A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning* in (2022)

7. Miladinović, Đ., Shridhar, K., Jain, K., Paulus, M. B., Buhmann, J. M. & Allen, C. *Learning to Drop Out: An Adversarial Approach to Training Sequence VAEs* in *Advances in Neural Information Processing Systems* (2022)

8. Duan, H., Vaezipoor, P., Paulus, M. B., Ruan, Y. & Maddison, C. J. *Augment with Care: Contrastive Learning for Combinatorial Problems* in *Proceedings of the 39th International Conference on Machine Learning* (2022)

1.3 COLLABORATORS

For the better, research is often a collaborative effort. I had the pleasure to work closely with numerous colleagues over the last years. Some have contributed to the work I present in this thesis. Andreas Krause has encouraged me to pursue research at the intersection of machine learning and discrete optimization. His unconditional support, generosity and valuable feedback facilitated this research. I am grateful to him. Chris Maddison contributed several key ideas to our work on gradient estimation. He has

been a close collaborator and exceptional mentor to me during my PhD studies. Dami Choi contributed important experimental results to our work on gradient estimation with stochastic softmax tricks. Giulia Zarpellon greatly helped to conceptualize and conduct machine learning research for integer programming. Daniel Tarlow and Laurent Charlin provided valuable feedback. It was a pleasure to work with all of them.

Part I

LEARNING WITH DISCRETE OPTIMIZATION

2

BACKGROUND

2.1 NON-SUPERVISED MACHINE LEARNING

We distinguish between *supervised* and *non-supervised* machine learning. Supervised machine learning traditionally relies on labelled examples of the form $(\boldsymbol{x}_{\text{sup}}, y)$. The vector $\boldsymbol{x}_{\text{sup}}$ contains features or covariates that describe the example and y is a corresponding label. The supervised learning problem is to find good predictive models of the label y. Many learning algorithms parameterize the predictive models and cast learning as an optimization program,

$$\min_{\boldsymbol{\theta}_{\mathbf{sup}}} F_{\text{sup}}(\boldsymbol{\theta}_{\mathbf{sup}}) := L\left(y, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\mathbf{sup}}\right) \tag{2.1}$$

where $\boldsymbol{\theta}_{\mathbf{sup}}$ are the learnable parameters, $L(\cdot)$ is a loss function and $F_{\text{sup}}(\cdot)$ is the supervised objective. This framework covers many learning problems in classification and regression. For example, in logistic regression the label designates a binary class $y \in \{0, 1\}$. The model predicts a conditional probability from a linear combination of the features and a composition with the sigmoid function $\hat{y} = \text{sigmoid}\left(\boldsymbol{\theta}_{\mathbf{sup}}^{\mathsf{T}} \boldsymbol{x}_{\text{sup}}\right)$. The loss function is the binary cross-entropy loss $L(\boldsymbol{x}_{\text{sup}}, y, \boldsymbol{\theta}_{\mathbf{sup}}) = -\left(y \log \hat{y} + (1 - y) \log(1 - \hat{y})\right)$. Typically, models are learnt over a collection of examples instead of a single one and equation (2.1) can easily be modified to accommodate this case. For simple models, the optimization problem in (2.1) may admit a solution in closed form. However, most learning algorithms, including those for logistic regression or neural networks, will rely on gradient methods. These methods differentiate the objective $F_{\text{sup}}(\cdot)$ with respect to the model parameters $\boldsymbol{\theta}_{\mathbf{sup}}$ and use the gradient to guide the optimization of (2.1). Over the last decades supervised learning and in particular deep learning has enjoyed great success. For example, supervised learning facilitates state-of-the-art performance in computer vision [e.g., 45, 46]and natural language processing [e.g., 47, 48].

Despite this success, many important problems in machine learning cannot be addressed with the methods of supervised learning. Such problems are primarily found in unsupervised, reinforcement or self-supervised learning. We collectively refer to these domains as *non-supervised* machine

learning. In non-supervised learning, supervision is limited or may not exist at all, because it is too expensive to acquire or otherwise infeasible. In the absence of supervision, it is natural to design probabilistic models that capture uncertainty. Many of these models incorporate latent random variables $\boldsymbol{X} \sim q_{\boldsymbol{\theta}}$ and pose the non-supervised learning problem,

$$\min_{\boldsymbol{\theta}, \boldsymbol{\theta}_{\mathbf{sup}}} F(\boldsymbol{\theta}, \boldsymbol{\theta}_{\mathbf{sup}}) := \mathbb{E}_{\boldsymbol{\theta}} \left[L\left(\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\mathbf{sup}}\right) \right] \qquad \text{where } \boldsymbol{X} \sim q_{\boldsymbol{\theta}} \tag{2.2}$$

This framework captures the essence of a wide range of problems in non-supervised learning, including for example clustering, representation learning, generative modeling or stochastic control in discrete time. For illustration, we survey an example from each of the three domains of non-supervised learning below.

- Variational Autoencoders [49, VAEs] are a popular class of latent variable models in unsupervised learning. VAEs use a neural network encoder with parameters $\boldsymbol{\theta}$ and input $\boldsymbol{x}_{\text{sup}}$ to parameterize a conditional distribution $q_{\boldsymbol{\theta}}$ over the latent variables. A sample from the latent distribution $\boldsymbol{X} \sim q_{\boldsymbol{\theta}}$ is then drawn and decoded by another neural network with parameters $\boldsymbol{\theta}_{\mathbf{sup}}$ to reconstruct the original input $\boldsymbol{x}_{\text{sup}}$. The expected loss is a variational objective to train the model.

- Policy gradient methods [see e.g., 50] are a prominent class of techniques in reinforcement learning. They are covered by the formulation above, when the covariate $\boldsymbol{x}_{\text{sup}}$ may contain a sequence of state representations. The random variable $\boldsymbol{X}$ is a sequence of actions and the parameters $\boldsymbol{\theta}$ specify a policy $q_{\boldsymbol{\theta}}$ over actions. The parameters $\boldsymbol{\theta}_{\mathbf{sup}}$ are void and the loss L may negate a reward signal.

- SimCLR [51] is a simple framework for self-supervised learning of visual representations. This method applies two random data augmentations to an input image and maps each perturbed image via a base encoder to a latent representation. For SimCLR, the original input image is $\boldsymbol{x}_{\text{sup}}$. The latent variable $\boldsymbol{X} \sim q_{\boldsymbol{\theta}}$ contains the latent representations of the two perturbed images. They are random, because the data augmentations are randomly sampled. The parameters of the base encoder are $\boldsymbol{\theta}$. The contrastive loss L is defined over a batch of images. The parameters $\boldsymbol{\theta}_{\mathbf{sup}}$ include the projection head and the temperature parameter in SimCLR.

As with supervised learning, most methods in non-supervised learning typically rely on gradient-based optimization. However, computing

gradients for the distributional parameters $\boldsymbol{\theta}$ in (2.2) is challenging. The random variable $\boldsymbol{X}$ usually is high-dimensional and evaluations of the loss function L are often expensive. In all but the simplest cases, we are not able to analytically evaluate the expectation in (2.2). Instead, both the objective and its gradient with respect to $\boldsymbol{\theta}$

$$\boldsymbol{\eta} \coloneqq \frac{d}{d\boldsymbol{\theta}} F(\boldsymbol{\theta}, \boldsymbol{\theta}_{\mathbf{sup}}) = \frac{d}{d\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{\theta}} \left[L\left(\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\mathbf{sup}}\right) \right] \tag{2.3}$$

must be *estimated* to facilitate learning. As a result, gradient estimators for non-supervised learning are of great interest, and the following two chapters of this thesis contribute to this line of research. For clarity, we omit the $\boldsymbol{\theta}_{\mathbf{sup}}$ and $\boldsymbol{x}_{\text{sup}}$ from the loss $L(\cdot)$ and the objective $F(\cdot)$ in the remainder of this thesis when this does not cause confusion.

2.2 LEARNING WITH DISCRETE DISTRIBUTIONS

In this thesis, we focus on cases of non-supervised learning where the random variable X is discrete. Such models are of interest for applications in computer vision and natural language processing as we demonstrate in our experiments in Chapter 3 and 4. They also feature prominently in bioinformatics, where domains are often highly structured, and reinforcement learning, where the control of systems with large discrete action spaces is a long-standing goal [50]. Notably, state-of-the-art generative models [10, 52] rely on discrete latent representations. In these models, discrete variables may aid with the modeling of multimodal distributions and improve sample diversity. More generally, discrete variables are useful, because the naturally induce sparsity. This can provide an effective form of regularization, render models more interpretable, facilitate conditional combination or give rise to *hybrid* models that integrate traditional algorithmic procedures.

REGULARIZATION Discrete variables naturally give rise to sparse representations and it is well known that sparsity can regularize learning effectively [53]. The earliest examples of this idea focussed on selecting variables or features to reduce model cardinality [54]. More recently the benefits of using *structured sparsity* [55, 56] have been demonstrated in computer vision [57], natural language processing [58, 59] or bioinformatics [60]. In our experiments in chapter 3, we learn distributions over highly structured discrete objects, such as trees or correlated subsets. We show that this may provide a useful prior in some applications and improves overall

performance, in particular if training data is scarce or model capacity must be limited, such as in edge computing.

INTERPRETABILITY The ability to question, understand and trust machine learning models is central to enable their adoption and deployment [e.g., 61, 62]. Unfortunately, the most performant models tend to be opaque and black-box. Methods are needed that improve their interpretability or provide explanations. The use of discrete variables can facilitate such methods. For example, Chen *et al.* [63] develop a variational framework for model explanation that uses discrete variables to identify subsets of salient input features. This method can guide the attention of a practitioner when inspecting a model prediction. We adopt this framework for some of our experiments in chapter 3. Another prominent direction is to use categorical variables to learn disentangled and interpretable representations of data [64].

MODEL DESIGN Sometimes, the choice of discrete variables results from considerations of model design. For example, Bengio, Léonard & Courville [65] observed that the sparsity of discrete variables can facilitate conditional computation. The idea is to condition part of the computation in a feed-forward model on the value of a discrete variable. For example, sparse gating units can be trained to select which part of a model is activated for a particular input. This approach can give rise to models that are lightweight and fast [66] or improve performance [67]. The idea has also been exploited in dynamic [68] and routing neural networks [69], to design custom convolutional layers [70–72] or to align computation along a learnt tree-structured computational graph [73]. We use this model in some of our experiments in chapter 4.

ALGORITHMIC INTEGRATION A promising direction to apply machine learning to increasingly complicated tasks is the design of *hybrid models*. These are models that combine learnable components with classical algorithms, for example for search [18], logical reasoning [74], planning [75] or optimization [76]. Sometimes, these models involve discrete distributions to facilitate end-to-end training. For example, the model in [75] learns a distribution over propositional symbols from image pixels and processes them with the aid of a classic planning algorithm. The model is trained without supervision of the propositional symbols or expert plans.

In this thesis, we will particularly particularly focus on structured discrete variables and formalize this notion in chapter 3. Such discrete variables may for example include sets, permutations or other objects that can be derived from undirected and directed graphs. In contrast to (unstructured) categorical variables, structured variables often admit more compact representations, as a result they typically scale better and may improve generalization. As machine learning is applied to relational data and increasingly complex tasks, structured discrete variables are of growing importance and have diverse applications. For example, in chapter 3 we consider structured discrete variables for neural relational inference [77] and model explanation [63].

2.3 ESTIMATING GRADIENTS FOR DISCRETE DISTRIBUTIONS

As argued in Section 2.1, to facilitate non-supervised learning the gradient of the distributional parameters $\boldsymbol{\theta}$ in equation (2.3) must be estimated. The method of choice in machine learning is Monte Carlo, because it scales favorably to higher dimensions. There are two general-purpose strategies to design Monte Carlo gradient estimators, one is based on the score function of the distribution $q_{\boldsymbol{\theta}}$ and one is based on the reparameterization of the random variable $\boldsymbol{X}$. The vast majority of gradient estimators in machine learning rely on either the score function or reparameterization, only few alternatives exist and have been considered in some non-supervised settings [e.g., 78]. Unfortunately, only the former strategy applies directly to discrete distributions, while the latter must be modified. Below, we describe the two strategies. We also note that a number of methods have been developed for computing gradients of *deterministic* discrete variables, for example to integrate optimization programs as layers in neural networks and train these models end-to-end [76, 79–83]. However, our focus in this thesis is on learning the parameters of discrete distributions and differentiating through *stochastic* discrete variables.

SCORE FUNCTION The score function estimator [84, 85] is based on the score function, which is the gradient of the log of the probability distribution $q_{\boldsymbol{\theta}}$ with respect to its distributional parameters. The estimator

relies on the properties of the score and on an interchange of differentiation and integration that recasts the gradient in (2.3) as an expectation:

$$\boldsymbol{\eta} = \frac{d}{d\boldsymbol{\theta}} \mathbb{E}_\theta \left[L\left(\boldsymbol{X}\right) \right] \tag{2.4}$$

$$= \mathbb{E}_\theta \left[L\left(\boldsymbol{X}\right) \frac{d}{d\boldsymbol{\theta}} \log q_{\boldsymbol{\theta}}\left(\boldsymbol{X}\right) \right] \tag{2.5}$$

This interchange is valid under very general assumptions on the smoothness of $q_{\boldsymbol{\theta}}$ [see 86, and references therein]. In particular, these assumptions permit the use of discrete distributions that our focus is on. Using the expression above, an unbiased gradient estimator can be obtained by approximating the expectation in equation (2.5) via Monte Carlo. The simplest instantiation of this idea is the score function estimator with a single sample,

$$\hat{\boldsymbol{\eta}}_{\text{VOID}} := L\left(\boldsymbol{X}\right) \frac{d}{d\boldsymbol{\theta}} \log q_{\boldsymbol{\theta}}\left(\boldsymbol{X}\right) \tag{2.6}$$

Unfortunately, the variance of this estimator tends to be very large and it cannot be used in non-supervised machine learning. As a result, several gradient estimators have been developed that are based on the score function estimator but attempt to reduce is variance. Most of them center the loss using a baseline $B(\cdot)$ to reduce variance,

$$\hat{\boldsymbol{\eta}}_{\text{SCORE}} := \left(L\left(\boldsymbol{X}\right) - B(\cdot)\right) \frac{d}{d\boldsymbol{\theta}} \log q_{\boldsymbol{\theta}}(\boldsymbol{X}) \tag{2.7}$$

This estimator remains unbiased, if the baseline $B(\cdot)$ does not depend on the variable $\boldsymbol{X}$, because the expectation of the score function is zero, i.e., $\mathbb{E}\left[\frac{d}{d\boldsymbol{\theta}} \log q_{\boldsymbol{\theta}}(\boldsymbol{X})\right] = 0$. However, the baseline effectively reduces the variance of the estimator, if its variance or the variance of the loss and the score are not too large. Based on equation (2.7), several gradient estimators have been proposed that choose different baselines. Most of them try to approximate the expected loss. A simple and common choice is to compute a moving average over past evaluations of the loss function when iteratively optimizing equation (2.2) via gradient descent. More sophisticated approaches consider Taylor approximations [87, 88] or consider input-dependent baselines $B(\boldsymbol{x}_{\text{sup}})$. For some applications, in particular variational inference, specialized control variates have been designed [89]. Another approach is to directly learn a good baseline Gregor *et al.* [90], Tucker *et al.* [91], and Grathwohl *et al.* [92]. Finally, the combination of several baselines is possible [93].

Other approaches to reduce variance exist. A straightforward approach for variance reduction is to use multiple samples. Several gradient estimators use multiple samples [94]. Because in principle the variance of any Monte Carlo estimator can be reduced by repeated sampling, we view this approach as complementary to our discussion and focus on single sample estimation in this thesis. In addition, we note that in modern machine learning, multiple evaluations of the loss function as a result of repeated sampling can be costly and may be inappropriate for some applications. Thus, there is a particular need for estimators that only require a single loss evaluation to compute gradient estimates. In chapter 3 and chapter 4 we develop such estimators.

REPARAMETERIZATION The reparameterization gradient estimator [49, 95–97] is based on a reparameterization of the random variable X in terms of a standard random variable $\boldsymbol{U}_0 \sim q_0$ and a deterministic transformation G_θ, such that

$$\boldsymbol{X} \sim q_{\boldsymbol{\theta}} \;\equiv\; \boldsymbol{X} = G_{\boldsymbol{\theta}}(\boldsymbol{U}_0) \qquad \text{where } \boldsymbol{U}_0 \sim q_0 \tag{2.8}$$

Several reparameterizations for common probability distributions are known and different methods exist to derive reparameterizations [98]. As with the score function estimator, the reparameterization estimator relies on an interchange of differentiation and integration, but additionally requires the loss function L to be differentiable with respect to $\boldsymbol{X}$, such that

$$\boldsymbol{\eta} = \frac{d}{d\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{\theta}} \left[L\left(\boldsymbol{X}\right) \right] = \frac{d}{d\boldsymbol{\theta}} \mathbb{E}_0 \left[L\left(G_{\boldsymbol{\theta}}(\boldsymbol{U}_0)\right) \right] \tag{2.9}$$

$$= \mathbb{E}_0 \left[\frac{\partial L(\boldsymbol{X})}{\partial \boldsymbol{X}} \frac{d}{d\boldsymbol{\theta}} G_{\boldsymbol{\theta}}(\boldsymbol{U}_0) \right] \tag{2.10}$$

Estimating the gradient via Monte Carlo is immediate from equation (2.10). Such reparameterization estimators enjoy several favorable properties. In particular, they are unbiased and low variance even in high dimensions [99]. As a result, they are widely used in machine learning and state-of-the-art [100].

RELAXATION Unfortunately, gradient estimators based on (2.10) are invalid for discrete variables that are our interest. This is, because any reparameterization of a discrete variable $\boldsymbol{X}$ necessarily requires jump discontinuities in the deterministic transformation $G_{\boldsymbol{\theta}}$. These discontinuities violate the assumptions underlying the interchange of differentiation and

integration in equation (2.10) as discussed in Chapter 7.2 of [97]. A remedy is to replace the discrete variable $\boldsymbol{X}$ with a continuous variable $\boldsymbol{X}^*$ at train time to learn the distributional parameters $\boldsymbol{\theta}$. The continuous variable $\boldsymbol{X}^*$ must admit a reparameterization gradient based on 2.10 and be coupled to the distributional parameters $\boldsymbol{\theta}$ of the discrete variable $\boldsymbol{X}$. At test time, the discrete variable $\boldsymbol{X}$ is used. These estimators are known as *relaxed* gradient estimators, because they relax the discrete random variable to a continuous random variable.

The Gumbel-Softmax estimator [101, 102] is a relaxed gradient estimator that has been proposed for categorical random variables. It is based on the Gumbel-Max trick [103, 104] that recasts categorical sampling as an optimization program. Let $X \sim \text{Categorical}(\boldsymbol{\theta})$ be a categorical random variable, such that $\mathbb{P}(X = i) \propto \exp(\theta_i)$. Then, the Gumbel-Max tricks perturbs the log-mass $\boldsymbol{\theta}$ with Gumbel utilities $\boldsymbol{U}_0 \sim \text{Gumbel}(0)$ and selects the category with the largest perturbed mass,

$$X \sim \text{Categorical}(\boldsymbol{\theta}) \;\equiv\; X = \arg\max_i (\boldsymbol{\theta} + \boldsymbol{U}_0)_i \tag{2.11}$$

The equivalence is due to the properties of the Gumbel distribution [see e.g., 36, Chapter 7]. As before, the Gumbel-Max does not admit a reparameterization gradient for X, because the arg max introduces jump discontinuities. Therefore, the Gumbel-Softmax uses the tempered softmax[1] in place of the arg max and considers the relaxed variable

$$\boldsymbol{X}_\tau^* = \text{softmax}_\tau(\boldsymbol{\theta} + \boldsymbol{U}_0) \tag{2.12}$$

As $\tau \to 0^+$, the approximation becomes more faithful, i.e., the relaxed variable converges to a categorical variable (in a one-hot embedding). The continuous random variable $\boldsymbol{X}_\tau^*$ admits a reparameterization gradient that can be used to devise the gradient estimator,

$$\hat{\boldsymbol{\eta}}_{\text{GS}} := \frac{\partial L(\boldsymbol{X}_\tau^*)}{\partial \boldsymbol{X}_\tau^*} \frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{\theta} + G) \tag{2.13}$$

This relaxed estimator is the Gumbel-Softmax gradient estimator. The estimator requires that the loss and its gradient can be evaluated for $\boldsymbol{X}_\tau^*$. It is a biased estimator of the gradient of interest in equation (2.3), but it is an unbiased[2] estimator of the gradient of the surrogate objective

$$F_\tau^*(\boldsymbol{\theta}, \boldsymbol{\theta}_{\textbf{sup}}) := \mathbb{E}_{\boldsymbol{\theta}}[L(\boldsymbol{X}_\tau^*, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\textbf{sup}})] \tag{2.14}$$

1 $\text{softmax}_\tau(\boldsymbol{u})_i = \exp(\boldsymbol{u}_i/\tau) / \sum_{j=1}^n \exp(\boldsymbol{u}_j/\tau)$ for $\boldsymbol{u} \in \mathbb{R}^n, \tau > 0$

2 Technically, one needs an additional local Lipschitz condition for $L(\boldsymbol{X}_\tau^*)$ in $\boldsymbol{\theta}$ [86, Prop. 2.3, Chap. 7].

Unfortunately, the Gumbel-Softmax estimator only handles categorical variables. However, as previously argued, many applications of interest involve structured discrete variables. The use of categorical variables in these settings is undesirable, because they do not scale and their representation are not distributed. To address this, a key contribution of this thesis is the design of new relaxed gradient estimator for structured variables. In chapter 3, we present *stochastic softmax tricks* that generalize the Gumbel-Softmax, offer a unified perspective on existing relaxed estimators and contain many new relaxations.

Another shortcoming of relaxed gradient estimators is that the benefits of discrete variables we surveyed in the previous section are lost at train time. For example, to achieve regularization via sparsity, it may be required to tune the temperature parameter carefully [101] and avoid a performance loss due to the integrality gap at train and test time. Also, Choi, Yoo & Lee [73] argues for the use of discrete variables in their model at train time to facilitate conditional computation. The model they propose does not admit the use of relaxed gradient estimators. To preserve the benefits of discrete variables or facilitate the use of relaxed gradient estimation, straight-through variants of relaxed gradient estimators have been proposed [see e.g., 102]. In chapter 4, we revisit the straight-through variant of Gumbel-Softmax estimator. We propose a simple scheme to provably reduce its variance and improve its properties for gradient estimation which leads to improved performance in our experiments.

3

GRADIENT ESTIMATION WITH STOCHASTIC SOFTMAX TRICKS

SYNOPSIS In the previous chapter, we reviewed gradient estimation in non-supervised machine learning. In particular, we discussed relaxed gradient estimators as a promising approach to estimate gradients for discrete variables. We introduced the Gumbel-Softmax estimator that is based on the Gumbel-Max trick for categorical variables. In this chapter we generalize this estimator to other discrete structured variables. We present *stochastic argmax tricks* that are perturbation models that naturally extend the Gumbel-Max trick to other discrete structured variables. As the Gumbel-Max, they can be relaxed to emit a gradient estimator based on reparameterization. We call these relaxations *stochastic softmax tricks*. They generalize the Gumbel-Softmax, offer a unified perspective on existing relaxed estimators and contain many new relaxations. We design structured relaxations for subset selection, spanning trees, arborescences, and others with stochastic softmax tricks. Experimentally, when compared to less structured baselines, we find that stochastic softmax tricks can be used to train latent variable models that perform better and discover more latent structure.

ATTRIBUTION This chapter is largely based on the following publication that was jointly authored with Dami Choi, Daniel Tarlow, Andreas Krause and Chris J. Maddison.

- Paulus, M. B. *et al. Gradient Estimation with Stochastic Softmax Tricks* in *Advances in Neural Information Processing Systems* (2020)

3.1 STRUCTURED EMBEDDINGS OF DISCRETE VARIABLES

In the previous chapter, we reviewed the Gumbel-Softmax gradient estimator for categorical distributions. In principle, any discrete distribution may be represented as a categorical distribution and the Gumbel-Softmax could be used to estimate gradients. However in practice, this approach is usually undesirable, because it cannot scale to large sets of discrete objects and standard representations carry no semantic information across categories.

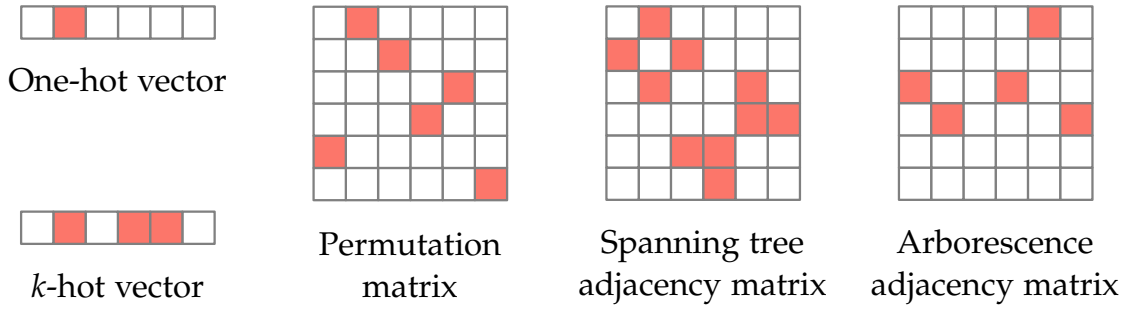


FIGURE 3.1: Structured discrete objects can be represented by binary arrays. In these graphical representations, color indicates 1 and no color indicates 0. For example, "Spanning tree" is the adjacency matrix of an undirected spanning tree over 6 nodes; "Arborescence" is the adjacency matrix of a directed spanning tree rooted at node 3.

To illustrate, let $\mathcal{S}$ be a non-empty, finite set of discrete objects. For example, $\mathcal{S}$ could contain all spanning trees of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In general, we can represent the objects $s \in \mathcal{S}$ with the use of an embedding function $\text{rep} : \mathcal{S} \to \mathbb{R}^n$, such that $\mathcal{X} = \{\text{rep}(s) \mid s \in \mathcal{S}\} \subseteq \mathbb{R}^n$ is convex independent[1] and contains the embedded objects. If we choose to model a categorical distribution over the objects $s \in \mathcal{S}$, we need to enumerate $s_1, \ldots, s_{|\mathcal{S}|}$ in $\mathcal{S}$ and we can let $\text{rep}(s)$ be the one-hot binary vector of length $|\mathcal{S}|$, with $\text{rep}(s)_i = 1$ iff $s = s_i$ to use the Gumbel-Softmax estimator. Unfortunately, this requires a very large ambient dimension $n = |\mathcal{S}|$ that scales unfavorably with the size of the graph. For example, the number of distinct spanning trees in a complete graph grows super-exponentially with the number of vertices. In addition, the categorical representation of the objects is *local* and not *distributed* [105]. The categories do not encode any notion of similarity between the discrete objects that in many applications is likely present and would aid generalization. For example, if two spanning trees share a large number of edges, we may often reasonably expect them to incur a loss of similar size. But in the categorical encoding, any two spanning trees are equally similar or dissimilar regardless of how many edges they share.

We can overcome these disadvantaged by choosing a more structured embedding function. For example, in the case of spanning trees we could use a structured representation where $\text{rep}(s)$ is a binary indicator vector of length $|\mathcal{E}| \ll |\mathcal{S}|$, with $\text{rep}(s)_e = 1$ iff edge e is in the tree s. The number of edges in a complete graph only grows quadratically with the number of vertices of the graph. Using this representation, spanning trees that

1 Convex independence is the analog of linear independence for convex combinations.

are similar, because they share a large number of edges, are embedded more closely together, i.e., their embeddings have a larger cosine similarity. Depending on the application, many choices for $\text{rep}(s)$ naturally emerge and we give some additional examples in Figure 3.1.

Our goal is now to design relaxed gradient estimators for discrete distributions over these structured domains $\mathcal{X}$. Our strategy will be to develop stochastic programs that extend the Gumbel-Max trick to structured domains and identify a method to relax those programs to emit gradient estimators based on reparameterization.

3.2 STOCHASTIC ARGMAX TRICKS

Recall the Gumbel-Max from chapter 2 that recasts categorical sampling as identifying a category with the largest perturbed probability mass. When we use one-hot embeddings $\mathcal{X} \ni \boldsymbol{x}$ to represent each category, we can rewrite the Gumbel-Max trick as a random linear program

$$\boldsymbol{X} = \arg\max_{\boldsymbol{x} \in \mathcal{X}} (\boldsymbol{\theta} + \boldsymbol{U}_0)^\intercal \boldsymbol{x} \qquad \text{where } \boldsymbol{U}_0 \sim \text{Gumbel}(\boldsymbol{0}) \tag{3.1}$$

where $\boldsymbol{X} \sim \text{Categorical}(\boldsymbol{\theta})$. This is revealing, because we can identify generalizations to the Gumbel-Max trick by considering random linear programs of the same form, but for other structured embeddings $\mathcal{X}$. We call these *stochastic argmax tricks* (SMTs), because they are perturbation models [106, 107], which we relax into stochastic softmax tricks in the next section.

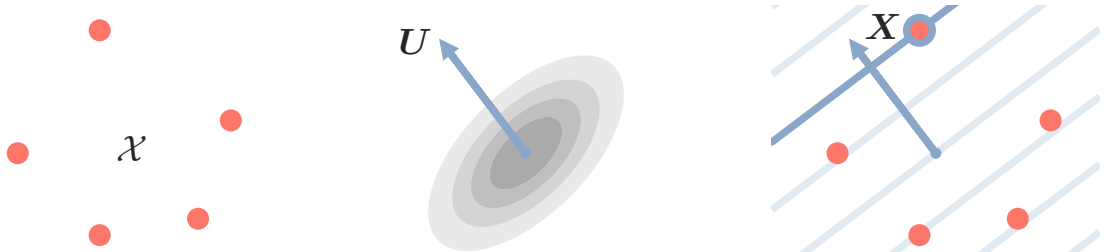


FIGURE 3.2: Stochastic argmax tricks are random linear programs that parameterize discrete probability distributions. A stochastic argmax trick is defined over the finite set $\mathcal{X}$ and with the aid of a random utility $\boldsymbol{U}$ with distributional parameters $\boldsymbol{\theta}$. The solution $\boldsymbol{X}$ of the linear program is a sample from the discrete distribution.

Definition 1. *Given a non-empty, convex independent, finite set $\mathcal{X} \subseteq \mathbb{R}^n$ and a random utility $\boldsymbol{U}$ whose distribution is parameterized by $\boldsymbol{\theta} \in \mathbb{R}^m$, a stochastic argmax trick for $\boldsymbol{X}$ is the linear program,*

$$\boldsymbol{X} = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\top \boldsymbol{x} \tag{3.2}$$

Of course, the Gumbel-Max trick is recovered with one-hot $\mathcal{X}$ and $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$. We illustrate stochastic argmax tricks in Figure 3.2. We assume that (3.2) is a.s. unique, which is guaranteed if $\boldsymbol{U}$ a.s. never lands in any particular lower dimensional subspace.

Proposition 1. *If $\mathbb{P}(\boldsymbol{U}^\top \boldsymbol{x} = 0) = 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$ such that $\boldsymbol{x} \neq 0$, then $\boldsymbol{X}$ in Definition 1 is a.s. unique.*

Proof. It suffices to show that for all subsets $\mathcal{X}' \subseteq \mathcal{X}$ with $|\mathcal{X}'| > 1$, the event $\{\mathcal{X}' = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\top \boldsymbol{x}\}$ has zero measure. Let $Z = \max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\top \boldsymbol{x}$. If $|\mathcal{X}'| > 1$, then we can pick two distinct points $\boldsymbol{x}^1, \boldsymbol{x}^2 \in \mathcal{X}'$ with $\boldsymbol{x}^1 \neq \boldsymbol{x}^2$. Now,

$$\mathbb{P}\left(\mathcal{X}' = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\top \boldsymbol{x}\right) = \mathbb{P}(\forall \boldsymbol{x} \in \mathcal{X}', \boldsymbol{U}^\top \boldsymbol{x} = Z) \tag{3.3}$$

$$\leq \mathbb{P}\left(\boldsymbol{U}^\top (\boldsymbol{x}^1 - \boldsymbol{x}^2) = 0\right) = 0. \tag{3.4}$$

□

Because efficient linear solvers are known for many structured $\mathcal{X}$, SMTs are capable of scaling to very large $\mathcal{S}$ [108–110]. For example, if $\mathcal{X}$ are the edge indicator vectors of spanning trees $\mathcal{S}$, then (3.2) is the maximum spanning tree problem, which is solved by Kruskal's algorithm [32].

The role of the SMT in our framework is to reparameterize $q_{\boldsymbol{\theta}}$ in (2.2). Ideally, *given* $q_{\boldsymbol{\theta}}$, there would be an efficient (e.g., $\mathcal{O}(n)$) method for simulating *some* $\boldsymbol{U}$ such that the marginal of $\boldsymbol{X}$ in (3.2) is $q_{\boldsymbol{\theta}}$. The Gumbel-Max trick shows that this is possible for one-hot $\mathcal{X}$, but the situation is not so simple for structured $\mathcal{X}$. Characterizing the marginal of $\boldsymbol{X}$ in general is difficult [106, 111], but $\boldsymbol{U}$ that are efficient to sample from typically induce conditional independencies in $q_{\boldsymbol{\theta}}$ [107]. Therefore, we are not able to reparameterize an arbitrary $q_{\boldsymbol{\theta}}$ on structured $\mathcal{X}$. Instead, for structured $\mathcal{X}$ we *assume* that $q_{\boldsymbol{\theta}}$ is reparameterized by (3.2), and treat $\boldsymbol{U}$ as a modeling choice. Thus, we caution against the standard approach of taking $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ or $\boldsymbol{U} \sim \text{Normal}(\boldsymbol{\theta})$ without further analysis. Practically, in experiments we show that the difference in noise distribution can have a large impact

on quantitative results. Theoretically, we show in section 3.4 that an SMT over directed spanning trees with negative exponential utilities has a more interpretable structure than the same SMT with Gumbel utilities.

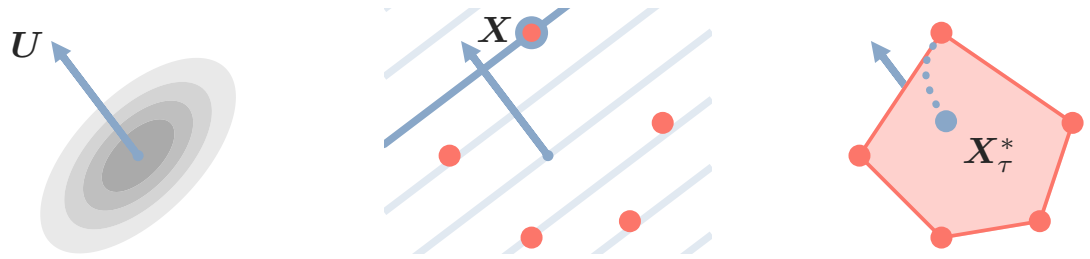
3.3 STOCHASTIC SOFTMAX TRICKS

If we assume that $\boldsymbol{X} \sim q_{\boldsymbol{\theta}}$ is reparameterized as an SMT, then a stochastic softmax trick (SST) is a random convex program with a solution that relaxes $\boldsymbol{X}$. An SST has a valid reparameterization gradient estimator. Thus, we propose using SSTs as surrogates for estimating the gradients in (2.3), a generalization of the Gumbel-Softmax approach we presented in chapter 2. Because we want gradients with respect to $\boldsymbol{\theta}$, we assume that $\boldsymbol{U}$ is also reparameterizable.

Given an SMT, an SST incorporates a strongly convex regularizer to the linear objective, and expands the state space to the convex hull of the embeddings $\mathcal{X} = \{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^m\} \subseteq \mathbb{R}^n$,

$$P := \operatorname{conv}(\mathcal{X}) := \left\{ \sum_{i=1}^m \lambda_i \boldsymbol{x}^i \;\middle|\; \boldsymbol{\lambda} \geq \mathbf{0}, \sum_{i=1}^m \lambda_i = 1 \right\}. \tag{3.5}$$

Expanding the state space to a convex polytope makes it path-connected, and the strongly convex regularizer ensures that the solutions are continuous over the polytope.



Random utility | Stochastic Argmax Trick | Stochastic Softmax Trick

FIGURE 3.3: Stochastic softmax tricks relax discrete distributions that are defined via stochastic argmax tricks. $\boldsymbol{X}$ is the solution of the stochastic argmax trick for the random utility $\boldsymbol{U}$ with distributional parameters $\boldsymbol{\theta}$. To design relaxed gradient estimators with respect to $\boldsymbol{\theta}$, $\boldsymbol{X}_\tau^*$ is the solution of a random convex program that continuously approximates $\boldsymbol{X}$ from within the convex hull of $\mathcal{X}$. The Gumbel-Softmax [101, 102] is an example of a stochastic softmax trick.

Definition 2. *Given a stochastic argmax trick* $(\mathcal{X}, U)$ *where* $P := \text{conv}(\mathcal{X})$ *and a proper, closed, strongly convex function* $R : \mathbb{R}^n \to \{\mathbb{R}, \infty\}$ *whose domain contains the relative interior of* P*, a* stochastic softmax trick *for* $\boldsymbol{X}$ *at temperature* $\tau > 0$ *is the convex program,*

$$\boldsymbol{X}_\tau^* = \arg\max_{\boldsymbol{x} \in P} \boldsymbol{U}^\top \boldsymbol{x} - \tau R(\boldsymbol{x}) \tag{3.6}$$

We illustrate stochastic softmax tricks in Figure 3.3. For one-hot $\mathcal{X}$, the Gumbel-Softmax is a special case of an SST where P is the probability simplex, $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, and $R(\boldsymbol{x}) = \sum_i x_i \log(x_i)$. Objectives like (3.6) have a long history in convex analysis [e.g., 112, Chap. 12] and machine learning [e.g., 113, Chap. 3]. In general, the difficulty of computing the SST will depend on the interaction between R and $\mathcal{X}$.

$\boldsymbol{X}_\tau^*$ is suitable as an approximation of $\boldsymbol{X}$. At positive temperatures τ, $\boldsymbol{X}_\tau^*$ is a function of $\boldsymbol{U}$ that ranges over the faces and relative interior of P. The degree of approximation is controlled by the temperature parameter, and as $\tau \to 0^+$, $\boldsymbol{X}_\tau^*$ converges a.s. to $\boldsymbol{X}$.

Proposition 2. *If* $\boldsymbol{X}$ *in Definition 1 is a.s. unique, then for* $\boldsymbol{X}_\tau^*$ *in Definition 2,* $\lim_{\tau \to 0^+} \boldsymbol{X}_\tau^* = \boldsymbol{X}$ *a.s. If additionally the loss* $L : P \to \mathbb{R}$ *is bounded and continuous, then*

$$\lim_{\tau \to 0^+} F_\tau^*(\boldsymbol{\theta}) = F(\boldsymbol{\theta})$$

where $F_\tau^*(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}}[L(\boldsymbol{X}_\tau^*)]$ *as in equation* (2.14) *and* $F(\boldsymbol{\theta}) := \mathbb{E}_{\boldsymbol{\theta}}[L(\boldsymbol{X})]$ *as in equation* (2.2)*.*

Proof. Let H_τ^* be defined as in (A.19). We have by Lemma 3 in Appendix A.2,

$$\boldsymbol{X}_\tau^* = \arg\max_{\boldsymbol{x} \in P} \boldsymbol{U}^\top \boldsymbol{x} - \tau R(\boldsymbol{x}) = \nabla H_\tau^*(\boldsymbol{U}). \tag{3.7}$$

If $\boldsymbol{X}$ is a.s. unique, then again by Lemma 3,

$$\begin{aligned} \mathbb{P}\left(\lim_{\tau \to 0^+} \boldsymbol{X}_\tau^* = \boldsymbol{X}\right) &= \mathbb{P}\left(\lim_{\tau \to 0^+} \nabla H_\tau^*(\boldsymbol{U}) = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\top \boldsymbol{x}\right) \\ &\geq \mathbb{P}\left(\boldsymbol{X} \text{ is unique}\right) \\ &= 1 \end{aligned}$$

The last part of the proof follows from the dominated convergence theorem, since the loss in bounded on P by assumption, so $|L(\boldsymbol{X}_\tau^*)|$ is surely bounded. □

It is common to consider temperature parameters that interpolate between marginal inference and a deterministic, most probable state. While superficially similar, our relaxation framework is different; as $\tau \to 0^+$, an SST approaches *a sample from the SMT model* as opposed to a deterministic state.

$\boldsymbol{X}_\tau^*$ also admits a reparameterization trick. The SST reparameterization gradient estimator given by,

$$\hat{\boldsymbol{\eta}}_{\text{SST}} = \frac{\partial L(\boldsymbol{X}_\tau^*)}{\partial \boldsymbol{X}_\tau^*} \frac{\partial \boldsymbol{X}_\tau^*}{\partial \boldsymbol{U}} \frac{d\boldsymbol{U}}{d\boldsymbol{\theta}}. \tag{3.8}$$

If L is differentiable on P, then this is an unbiased estimator[2] of the gradient of the surrogate $F_\tau^*(\boldsymbol{\theta})$, because $\boldsymbol{X}_\tau^*$ is continuous and a.e. differentiable:

Proposition 3. *$\boldsymbol{X}_\tau^*$ in Definition 2 exists, is unique, and is a.e. differentiable and continuous in $\boldsymbol{U}$.*

Proof. For H_τ^* defined in (A.19), we have by Lemma 3 in Appendix A.2,

$$\boldsymbol{X}_\tau^* = \arg\max_{\boldsymbol{x} \in P} \boldsymbol{U}^\intercal \boldsymbol{x} - \tau R(\boldsymbol{x}) = \nabla H_\tau^*(\boldsymbol{U}). \tag{3.9}$$

Our result follows by the other results of Lemma 3. □

Taken together, Proposition 2 and Proposition 3 suggest our proposed use for SSTs: optimize $F_\tau^*(\boldsymbol{\theta})$ at a positive temperature, where unbiased gradient estimation is available, but evaluate $F(\boldsymbol{\theta})$ at test time. We find that this works well in practice if the temperature used during optimization is treated as a hyperparameter and selected over a validation set. It is worth emphasizing that the choice of relaxation is unrelated to the distribution $q_{\boldsymbol{\theta}}$ of $\boldsymbol{X}$ in the corresponding SMT. R is not only a modeling choice; it is a computational choice that will affect the cost of computing (3.6) and the quality of the gradient estimator.

3.3.1 *Implementing Relaxed Gradient Estimators*

In general, the Jacobian $\partial \boldsymbol{X}_\tau^* / \partial \boldsymbol{U}$ will need to be derived separately given a choice of R and $\mathcal{X}$. For implementing the relaxed gradient estimator given in (3.8), several options are available. For example, the forward computation of $\boldsymbol{X}_\tau^*$ may be unrolled, such that the estimator can be computed with

2 Technically, one needs an additional local Lipschitz condition for $L(\boldsymbol{X}_\tau^*)$ in $\boldsymbol{\theta}$ [86, Prop. 2.3, Chap. 7].

the aid of modern software packages for automatic differentiation [114–116]. Moreover, as pointed out by [117], because the Jacobian of $\boldsymbol{X}_\tau^*$ is symmetric [118, Cor. 2.9], local finite difference approximations can be used to approximate $\hat{\boldsymbol{\eta}}_{\text{SST}}$

$$\frac{dL(\boldsymbol{X}_\tau^*)}{d\boldsymbol{U}} \approx \frac{\boldsymbol{X}_\tau^*(\boldsymbol{U} + \epsilon \partial L(\boldsymbol{X}_\tau^*)/\partial \boldsymbol{X}_\tau^*) - \boldsymbol{X}_\tau^*(\boldsymbol{U} - \epsilon \partial L(\boldsymbol{X}_\tau^*)/\partial \boldsymbol{X}_\tau^*)}{2\epsilon} \tag{3.10}$$

with equality in the limit as $\epsilon \to 0$ and where $\boldsymbol{X}_\tau^*(\boldsymbol{u}) = \arg\max_{\boldsymbol{x} \in P} \boldsymbol{x}^\intercal \boldsymbol{u} - \tau R(\boldsymbol{x})$ as in (3.6) This approximation is valid, because the Jacobian of $\boldsymbol{X}_\tau^*$ is symmetric [118, Corollary 2.9]. It is derived from the vector chain rule and the definition of the derivative of $\boldsymbol{X}_\tau^*$ in the direction $\partial L(\boldsymbol{X}_\tau^*)/\partial \boldsymbol{X}_\tau^*$. This method only requires two additional calls to a solver for (3.6) and does not require additional evaluations of $L(\cdot)$. We found this method helpful in a few experiments (c.f., Section 3.6).

In addition, for some specific choices of R and $\mathcal{X}$, it may be more efficient to compute the estimator exactly via a custom backward pass [see e.g., 119, 120]. There are many, well-studied R for which (3.6) is efficiently solvable. If $R(\boldsymbol{x}) = \|\boldsymbol{x}\|^2/2$, then $\boldsymbol{X}_\tau^*$ is the Euclidean projection of $\boldsymbol{U}/\tau$ onto P. Efficient projection algorithms exist for some convex sets [see 121–124, and references therein]. Moreover, there are generic algorithms that only call linear solvers as subroutines [125]. In some of the settings we consider, generic negative-entropy-based relaxations are also applicable. We refer to relaxations with $R(\boldsymbol{x}) = \sum_{i=1}^n \boldsymbol{x}_i \log(\boldsymbol{x}_i)$ as *categorical entropy relaxations* [e.g., 124, 126]. We refer to relaxations with $R(\boldsymbol{x}) = \sum_{i=1}^n \boldsymbol{x}_i \log(\boldsymbol{x}_i) + (1 - \boldsymbol{x}_i)\log(1 - \boldsymbol{x}_i)$ as *binary entropy relaxations* [e.g., 119].

Marginal inference in exponential families is a rich source of SST relaxations. Consider an exponential family over the finite set $\mathcal{X}$ with natural parameters $\boldsymbol{u}/\tau \in \mathbb{R}^n$ such that the probability of $\boldsymbol{x} \in \mathcal{X}$ is proportional to $\exp(\boldsymbol{u}^\intercal \boldsymbol{x}/\tau)$. The *marginals* $\mu_\tau : \mathbb{R}^n \to \text{conv}(\mathcal{X})$ of this family are solutions of a convex program in exactly the form (3.6) [113], i.e., there exists $R' : \text{conv}(\mathcal{X}) \to \{\mathbb{R}, \infty\}$ such that,

$$\mu_\tau(\boldsymbol{u}) := \sum\nolimits_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{x} \frac{\exp(\boldsymbol{u}^\intercal \boldsymbol{x}/\tau)}{\sum_{\boldsymbol{x}' \in \mathcal{X}} \exp(\boldsymbol{u}^\intercal \boldsymbol{x}'/\tau)} = \arg\max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x} - \tau R'(\boldsymbol{x}) \tag{3.11}$$

The definition of R', which generates μ_τ in (3.11), can be found in [113, Thm. 3.4]. R' is a kind of negative entropy and in our case it satisfies the assumptions in Definition 2. Computing μ_τ amounts to marginal inference in the exponential family, and efficient algorithms are known in many cases [see 110, 113], including those we consider. We call $\boldsymbol{X}_\tau^* = \mu_\tau(\boldsymbol{U})$ the *exponential family entropy relaxation*.

3.3.2 *Stochastic Softmax Tricks for Variational Inference*

Some objectives in non-supervised machine learning involve the density of $\boldsymbol{X}$. Variational inference with the evidence lower bound is a classic example where the objective includes the Kullback-Leibler divergence between the a prior and the model $q_{\boldsymbol{\theta}}$ distribution. In the case of variational inference with discrete latent variables and relaxed gradient estimators, there are at least three possible options for relaxing the objective [see e.g., 101, Section C.3]. We may define the Kullback-Leibler divergence with respect to $\boldsymbol{X}$ (using a prior over $\mathcal{X}$), with respect to $\boldsymbol{X}_\tau^*$ (using a prior over $\text{conv}(\mathcal{X})$) or with respect to $\boldsymbol{U}$ (using a prior over $\mathbb{R}^n$). In general, we do not an explicit tractable density of $\boldsymbol{X}_\tau^*$ or $\boldsymbol{X}$ and therefore suggest to compute the Kullback-Leibler divergence with respect to a prior for $\boldsymbol{U}$. The Kullback-Leibler divergence with respect to $\boldsymbol{U}$ is an *upper-bound* to the Kullback-Leibler divergence with respect to $\boldsymbol{X}_\tau^*$ due to a data processing inequality. Therefore, the evidence lower bound that we are optimizing is a *lower bound* to the relaxed variational objective where the Kullback-Leibler divergence is defined with respect to $\boldsymbol{X}_\tau^*$. Whether or not this is a good choice is an empirical question. Note, that *when optimizing the relaxed objective*, using a Kullback-Leibler divergence with respect to $\boldsymbol{X}$ does not result in a lower bound to the relaxed variational objective, as it is not necessarily an evidence lower bound for the continuous relaxed model [see again e.g., 101, Section C.3].

3.4 EXAMPLES OF STOCHASTIC SOFTMAX TRICKS

The Gumbel-Softmax [101, 102] introduced neither the Gumbel-Max trick nor the softmax. The novelty of this work is neither the pertubation model framework nor the relaxation framework in isolation, but their combined use for gradient estimation. Here, we layout some example SSTs, organized by the set $\mathcal{S}$ with a choice of embeddings $\mathcal{X}$. For each $\mathcal{S}$, we identify an appropriate set $\mathcal{X} \subseteq \mathbb{R}^n$ of structured embeddings. We also discuss utility distributions for which in some cases we can provide a simple, "closed-form", categorical sampling process for X, i.e., a generalization of the Gumbel-Max trick. We also cover potential relaxations used in the experiments as well as bespoke relaxations that have previously been described but are not SSTs.

3.4.1 *Element Selection*

ONE-HOT BINARY EMBEDDINGS. Given a finite set $\mathcal{S}$ with $|\mathcal{S}| = n$, we can associate each $s \in \mathcal{S}$ with a one-hot binary embedding. Let $\mathcal{X} \subseteq \mathbb{R}^n$ be the following set of one-hot embeddings,

$$\mathcal{X} = \left\{ \boldsymbol{x} \in \{0,1\}^n \,\middle|\, \sum_i x_i = 1 \right\}. \tag{3.12}$$

For $\boldsymbol{u} \in \mathbb{R}^n$, a solution to the linear program $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ is given by setting $x^\star_{j^\star} = 1$ for some $j^\star \in \arg\max_i u_i$ and $x^\star_i = 0$ otherwise.

RANDOM UTILITIES. If $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, then $\boldsymbol{X} \sim \text{Categorical}(\boldsymbol{\theta})$. This recovers the by now familiar Gumbel-Max trick [103, 104], which follows from Propositions 6 and 7.

RELAXTIONS. If $R(\boldsymbol{x}) = \sum_i x_i \log x_i$, then the SST solution $\boldsymbol{X}^*_\tau$ is given by

$$\boldsymbol{X}^*_\tau = \left(\frac{\exp(U_i/\tau)}{\sum_{j=1}^n \exp(U_j/\tau)} \right)_{i=1}^n. \tag{3.13}$$

In this case, the categorical entropy relaxation and the exponential family relaxation coincide. When $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, this recovers the Gumbel-Softmax trick from the previous chapter. If $R(\boldsymbol{x}) = \|\boldsymbol{x}\|^2/2$, then $\boldsymbol{X}^*_\tau$ can be computed using the `sparsemax` operator [127]. In analogy, we name this relaxation with $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ the Gumbel-Sparsemax trick.

3.4.2 *Subset Selection*

BINARY VECTOR EMBEDDINGS. Given a finite set $\mathcal{V}$ with $|\mathcal{V}| = n$, let $\mathcal{S}$ be the set of all subsets of $\mathcal{V}$, i.e., $\mathcal{S} = 2^{\mathcal{V}} := \{s \subseteq \mathcal{V}\}$. The indicator vector embeddings of $\mathcal{S}$ is the set,

$$\mathcal{X} = \{\boldsymbol{x}_s : s \in 2^{\mathcal{V}}\} = \{0,1\}^{|\mathcal{V}|} \tag{3.14}$$

For $\boldsymbol{u} \in \mathbb{R}^n$, a solution to the linear program $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ is given by setting $x^\star_i = 1$ if $u_i > 0$ and $x^\star_i = 0$ otherwise, for all $i \leq n$.

RANDOM UTILITIES. If $\boldsymbol{U} \sim \text{Logistic}(\boldsymbol{\theta})$, then $\boldsymbol{X} \sim \text{Bern}(\text{sigmoid}(\boldsymbol{\theta}))$, where $\text{sigmoid}(\cdot)$ is the sigmoid function. This corresponds to an application of the Gumbel-Max trick independently to each element in $\mathcal{V}$.

$\boldsymbol{U} \sim \text{Logistic}(\boldsymbol{\theta})$ has the same distribution as $\boldsymbol{\theta} + \log \boldsymbol{U}_0' - \log(1 - \boldsymbol{U}_0')$ for $\boldsymbol{U}_0' \sim \text{Uniform}(0, 1)$.

RELAXATIONS. For this case, the exponential family and the binary entropy relaxation, where $R(\boldsymbol{x}) = \sum_{i=1}^n x_i \log(x_i) + (1 - x_i)\log(1 - x_i)$, coincide. The SST solution $\boldsymbol{X}_\tau^*$ is given by

$$\boldsymbol{X}_\tau^* = (\text{sigmoid}(U_i/\tau))_{i=1}^n \tag{3.15}$$

where $\text{sigmoid}(\cdot)$ is the sigmoid function. For the categorical entropy relaxation with $R(\boldsymbol{x}) = \sum_{i=1}^n x_i \log(x_i)$, the SST solution is given by $\boldsymbol{X}_\tau^* = (\min(1, \exp(U_i/\tau)))_{i=1}^n$ [124].

3.4.3 *k-Subset Selection*

k-HOT BINARY EMBEDDINGS. Given a finite set $\mathcal{V}$ with $|\mathcal{V}| = n$, let $\mathcal{S}$ be the set of all subsets of $\mathcal{V}$ with cardinality $1 \leq k < n$, i.e., $\mathcal{S} = \{s \subseteq \mathcal{V} \,|\, |s| = k\}$. The indicator vector embeddings of $\mathcal{S}$ is the set,

$$\mathcal{X} = \{\boldsymbol{x}_s : s \subseteq \mathcal{V},\ |s| = k\} \tag{3.16}$$

For $\boldsymbol{u} \in \mathbb{R}^n$, let $\arg\text{topk}\, \boldsymbol{u}$ be the operator that returns the indices of the k largest values of $\boldsymbol{u}$. For $\boldsymbol{u} \in \mathbb{R}^n$, a solution to the linear program $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ is given by setting $x_i^\star = 1$ for $i \in \arg\text{topk}\, \boldsymbol{u}$ and $x_i^\star = 0$ otherwise.

RANDOM UTILITIES. If $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, this induces a Plackett-Luce model [103][128] over the indices that sort $\boldsymbol{U}$ in descending order. In particular, $\boldsymbol{X}$ may be sampled by sampling k times without replacement from the set $\{1, \ldots, n\}$ with probabilities proportional to $\exp(\theta_i)$, setting the sampled indices of $\boldsymbol{X}$ to 1, and the others to 0 [129]. This can be seen as a consequence of Corollary 2.

RELAXATIONS. For the Euclidean relaxation with $R(\boldsymbol{x}) = \|\boldsymbol{x}\|^2/2$, we computed $\boldsymbol{X}_\tau^*$ using a bisection method to solve the constrained quadratic program, but note that other algorithms are available [124]. For the categorical entropy relaxation with $R(\boldsymbol{x}) = \sum_{i=1}^n x_i \log(x_i)$, the SST solution $\boldsymbol{X}_\tau^*$ can be computed efficiently using the algorithm described in [120]. For the binary entropy relaxation with $R(\boldsymbol{x}) = \sum_{i=1}^n x_i \log(x_i) + (1 - x_i)\log(1 - x_i)$, the SST solution can be computed using the algorithm in [119]. Finally, for the exponential family relaxation, the SST solution can be computed using

dynamic programming as described in [130]. *L2X* [63] and *SoftSub* [131] are two bespoke relaxations for subset selection that cannot be described with our framework.

3.4.4 *Correlated k-Subset Selection*

CORRELATED k-HOT BINARY EMBEDDINGS. Given a finite set $\mathcal{V}$ with $|\mathcal{V}| = n$, let $\mathcal{S}$ be the set of all subsets of $\mathcal{V}$ with cardinality $1 \leq k < n$, i.e., $\mathcal{S} = \{s \subseteq \mathcal{V} \,|\, |s| = k\}$. We can associate each $s \in \mathcal{S}$ with a $(2n-1)$-dimensional binary embedding with a k-hot cardinality constraint on the first n dimensions and a constraint that the $n-1$ dimensions indicate correlations between adjacent dimensions in the first n, i.e. the vertices of the correlation polytope of a chain [113, Ex. 3.8] with an added cardinality constraint [132]. Let $\mathcal{X} \subseteq \mathbb{R}^{2n-1}$ be the set of all such embeddings,

$$\mathcal{X} = \left\{ \boldsymbol{x} \in \{0,1\}^{2n-1} \,\middle|\, \textstyle\sum_{i=1}^{n} x_i = k;\ x_i = \boldsymbol{x}_{i-n}\boldsymbol{x}_{i-n+1} \text{ for all } n < i \leq 2n-1 \right\}. \tag{3.17}$$

For $\boldsymbol{u} \in \mathbb{R}^n$, a solution to the linear program $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ can be computed using dynamic programming [130, 132].

RANDOM UTILITIES. In our experiments for correlated k-subset selection we considered Gumbel unary utilities with fixed pairwise utilities. This is, we considered $\boldsymbol{U}_{1:n} \sim \text{Gumbel}(\boldsymbol{\theta}_{1:n})$ and $\boldsymbol{U}_{(n+1):(2n-1)} = \boldsymbol{\theta}_{(n+1):(2n-1)}$.

RELAXATIONS. The exponential family relaxation for correlated k-subsets can be computed using dynamic programming as described in [130, 132].

3.4.5 *Perfect Bipartite Matchings*

PERMUTATION MATRIX EMBEDDINGS. Given a complete bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{V}', \mathcal{E})$, let $\mathcal{S}$ be the set of all perfect matchings. We can associate each $s \in \mathcal{S}$ with a permutation matrix and let $\mathcal{X}$ be the set of all such matrices,

$$\mathcal{X} = \left\{ x \in \{0,1\}^{n\times n} \,\middle|\, \text{for all } 1 \leq i,j \leq n, \sum_i x_{ij} = 1, \sum_j x_{ij} = 1 \right\}. \tag{3.18}$$

For $\boldsymbol{u} \in \mathbb{R}^{n\times n}$, a solution to the linear program $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ can be computed using the Hungarian method [33].

RANDOM UTILITIES. Previously, [133] considered $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ and [134] uses correlated Gumbel-based utilities that induce a Plackett-Luce model [103][128].

RELAXATIONS. For the categorical entropy relaxation with $R(\boldsymbol{x}) = \sum_{i=1}^{n} x_i \log(x_i)$, the SST solution $\boldsymbol{X}_\tau^*$ can be computed using the Sinkhorn algorithm [135]. When choosing Gumbel utilities, this recovers Gumbel-Sinkhorn [133]. This relaxation can also be used to relax the Plackett-Luce model, if combined with the utility distribution in [134]. Grover *et al.* [134] proposed a bespoke relaxation.

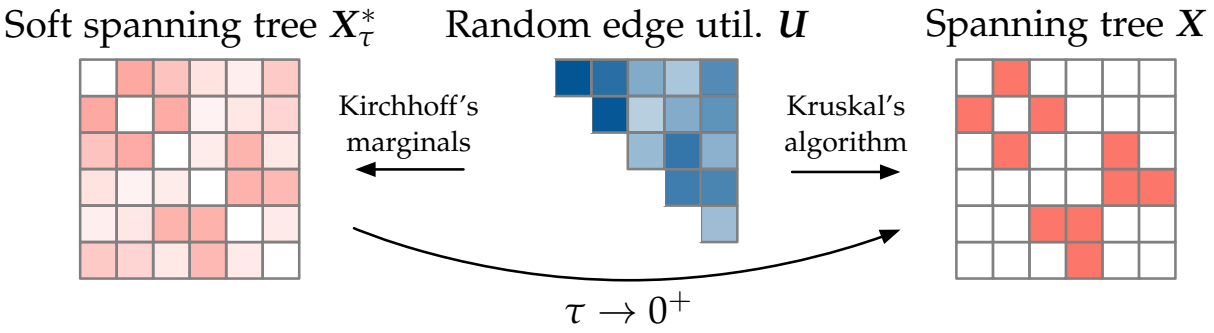


FIGURE 3.4: An example realization of a spanning tree SST for an undirected graph. Middle: Random undirected edge utilities. Left: The random soft spanning tree $\boldsymbol{X}_\tau^*$, represented as a weighted adjacency matrix, can be computed via Kirchhoff's Matrix-Tree theorem. Right: The random spanning tree $\boldsymbol{X}$, represented as an adjacency matrix, can be computed with Kruskal's algorithm.

3.4.6 *Undirected Spanning Trees*

EDGE INDICATOR EMBEDDINGS. Given a undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, let $\mathcal{S}$ be the set of spanning trees of $\mathcal{G}$ represented as subsets of the edges. The indicator vector embeddings of $\mathcal{S}$ is the set,

$$\mathcal{X} = \cup_{s \in \mathcal{S}} \{\boldsymbol{x}_s\} \subseteq \{0,1\}^{|\mathcal{E}|} \tag{3.19}$$

We assume that $\mathcal{G}$ has at least one spanning tree, and thus $\mathcal{X}$ is non-empty. A linear program over $\mathcal{X}$ is known as a maximum weight spanning tree problem. It is efficiently solved by Kruskal's algorithm [32].

RANDOM UTILITIES. In our experiments, we used $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$. In this case, there is a simple, categorical sampling process that describes the

distribution over $\boldsymbol{X}$. The sampling process follows Kruskal's algorithm [32]. The steps of Kruskal's algorithm are as follows: sort the list of edges $e \in \mathcal{E}$ in non-increasing order according to their utilities $\boldsymbol{U}_e$, greedily construct a tree by adding edges to s as long as no cycles are created, and return the indicator vector $\boldsymbol{x}_s$. Using Corollary 2 and Proposition 6, for Gumbel utilities this is equivalent to the following process: sample edges $e \in \mathcal{E}$ without replacement with probabilities proportional to $\exp(\theta_e)$, add edges e to s in the sampled order as long as no cycles are created, and return the indicator vector x_s.

RELAXATIONS. The exponential family relaxation for spanning trees can be computed using Kirchhoff's Matrix-Tree Theorem. Here we present a quick informal review. Consider an exponential family with natural parameters $\boldsymbol{u} \in \mathbb{R}^{|\mathcal{E}|}$ over $\mathcal{X}$ such that the probability of $\boldsymbol{x} \in \mathcal{X}$ is proportional to $\exp(\boldsymbol{u}^\top \boldsymbol{x})$. Define the weights,

$$w_{ij} = \begin{cases} \exp(u_e) & \text{if } i \neq j \text{ and } \exists\, e \in \mathcal{E} \text{ connecting nodes } i \text{ and } j \\ 0 & \text{otherwise} \end{cases}. \tag{3.20}$$

Consider the graph Laplacian $\boldsymbol{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ defined by

$$\boldsymbol{A}_{ij} = \begin{cases} \sum_{k \neq j} w_{kj} & \text{if } i = j \\ -w_{ij} & \text{if } i \neq j \end{cases} \tag{3.21}$$

Let $\boldsymbol{A}^{k,k}$ be any submatrix of $\boldsymbol{A}$ obtained by deleting the kth row and kth column. The Kirchhoff Matrix-Tree Theoreom states that

$$\log\det \boldsymbol{A}^{k,k} = \log\left(\sum_{s \in \mathcal{S}} \exp\left(\boldsymbol{u}^\top \boldsymbol{x}_s\right)\right). \tag{3.22}$$

See [136, p. 14] for a reference. We can use this to compute the marginals of the exponential family via its derivative [113]. In particular,

$$\mu(\boldsymbol{u}) := \left(\frac{\partial \log\det A^{k,k}}{\partial \boldsymbol{u}_e}\right)_{e \in \mathcal{E}} = \sum_{s \in \mathcal{S}} \boldsymbol{x}_s \frac{\exp\left(\boldsymbol{u}^\top \boldsymbol{x}_s\right)}{\sum_{s' \in \mathcal{S}} \exp\left(\boldsymbol{u}^\top \boldsymbol{x}_{s'}\right)}. \tag{3.23}$$

These partial derivatives can be computed with standard software for automatic differentiation. All together, we may define the exponential family relaxation via $\boldsymbol{X}^*_\tau = \mu(\boldsymbol{U}/\tau)$. Notably, we can compute the expression in

(3.23) from the inverse of $\boldsymbol{A}^{k,k}$ [see e.g., 137, Section A.4.1]. In practice, matrix inversion may suffer from numerical instability when entries $\boldsymbol{A}$ vary widely. Since (3.23) is valid for any submatrix $\boldsymbol{A}^{k,k}$, we suggest to use the index k to be the row in which $\boldsymbol{A}$ has its maximum entry. In addition, numerical stability can be improved by clipping entries from below using a maximum range and by using the log-sum-exp trick.

3.4.7 *Rooted, Directed Spanning Trees*

EDGE INDICATOR EMBEDDINGS. Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, consider the set of r-arborescences for some root $r \in \mathcal{V}$. An r-arborescence is a subgraph of $\mathcal{G}$ that is a spanning tree if the edge directions are ignored and that has a directed path from r to every node in $\mathcal{V}$. For an r-arborescence $s \in \mathcal{S}(r)$, let $\boldsymbol{x}_s$ be the indicator vector of its edges and define the indicator vector embeddings of $\mathcal{S}(r)$ as,

$$\mathcal{X} = \cup_{s \in \mathcal{S}(r)} \{\boldsymbol{x}_s\} \subseteq \{0,1\}^{|\mathcal{E}|} \tag{3.24}$$

We assume that $\mathcal{G}$ has at least one r-arborescence, and thus $\mathcal{X}$ is non-empty. A linear program over $\mathcal{X}$ is known as a maximum weight r-arborescence problem. It is efficiently solved by the Chu-Liu-Edmonds algorithm (CLE) [138, 139], see Algorithm 1 for an implementation by [140].

RANDOM UTILITIES. In the experiments, we tried $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, $-\boldsymbol{U} \sim \text{Exp}(\boldsymbol{\theta})$ with $\boldsymbol{\theta} > \boldsymbol{0}$, and $\boldsymbol{U} \sim \text{Normal}(\boldsymbol{\theta})$. As far as we know $\boldsymbol{X}$ does not have any particularly simple closed-form categorical sampling process in the cases $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ or $\boldsymbol{U} \sim \text{Normal}(\boldsymbol{\theta})$.

In contrast, for negative exponential utilities $-\boldsymbol{U} \sim \text{Exp}(\boldsymbol{\theta})$, $\boldsymbol{X}$ can be sampled using the sampling process given in Algorithm 2. In some sense, Algorithm 2 is an elaborate generalization of the Gumbel-Max trick to arborescences.

We will argue that Algorithm 2 produces the same distribution over its output as Algorithm 1 does on negative exponential $\boldsymbol{U}_e$. To do this, we will argue that joint distribution of the sequence of edge choices (lines 2-4 colored red in Algorithm 1), after integrating out $\boldsymbol{U}$, is given by lines 2-4 (colored blue) of Algorithm 2. Consider the first call to CLE: all $\boldsymbol{U}_e$ are negative and distinct almost surely, for each node $v \neq r$ the maximum utility edge is picked from the set of entering edges $\mathcal{E}_v$, and all edges have their utilities modified by subtracting the maximum utility. The argmax of $\boldsymbol{U}_e$ over $\mathcal{E}_v$ is a categorical random variable with mass function proportional

Algorithm 1: Maximum r-arborescence [140]

Init: graph G, node r, $U_e \in \mathbb{R}$, $T = \emptyset$;
foreach *node* $v \neq r$ **do**
  $E_v = \{\text{edges entering } v\}$;
  $U'_e = U_e - \max_{e \in E_v} U_e$, $\forall e \in E_v$;
  Pick $e \in E_v$ s.t. $U'_e = 0$;
  $T = T \cup \{e\}$;
end
if T *is an arborescence* **then**
  return x_T;
else there is a directed cycle $C \subseteq T$
  Contract C to supernode, form graph G';
  Recurse on (G', r, U') to get arbor. T';
  Expand T' to subgraph of G and add
  all but one edge of C;
  return $x_{T'}$;
end

Algorithm 2: Equiv. for neg. exp. $\boldsymbol{U}$

Init: graph G, node r, $\theta_e > 0$, $T = \emptyset$;
foreach *node* $v \neq r$ **do**
  $E_v = \{\text{edges entering } v\}$;
  Sample $e \sim \theta_e \mathbf{1}_{E_v}(e)$;
  $T = T \cup \{e\}$;
  $\lambda'_a = \lambda_a$ if $a \neq e$ else ∞, $\forall a \in E_v$;
end
if T *is an arborescence* **then**
  return x_T;
else there is a directed cycle $C \subseteq T$
  Contract C to supernode, form graph G';
  Recurse on (G', r, λ') to get arbor. T';
  Expand T' to subgraph of G and add
  all but one edge of C;
  return $x_{T'}$;
end

FIGURE 3.5: Algorithm 1 and Algorithm 2 have the same output distribution for negative exponential $\boldsymbol{U}$, i.e., Algorithm 2 is an equivalent categorical sampling process for $\boldsymbol{X}$. Algorithm 1 computes the maximum point of a stochastic r-arborescence trick with random utilities $\boldsymbol{U}_e$ [140]. When $-U_e \sim \text{Exp}(\theta_e)$, it has the same distribution as Algorithm 2. Algorithm 2 samples a random r-arborescence given rates $\theta_e > 0$ for each edge. Both algorithms assume that $\mathcal{G}$ has at least one r-arbor. Color indicates the main difference.

to the rates θ_e, and it is independent of the max of $\boldsymbol{U}_e$ over $\mathcal{E}_v$ by Prop. 7. By Corollary 1, the procedure of modifying the utilities leaves the distribution of all unpicked edges invariant and sets the utility of the argmax edge to 0. Thus, the distribution of $\boldsymbol{U}'$ passed one level up the recursive stack is the same as $\boldsymbol{U}$ with the exception of a randomly chosen subset of utilities $\boldsymbol{U}'_{\mathcal{E}}$ whose rates have been set to ∞. The equivalence in distribution between Algorithm 1 and Algorithm 2 follows by induction.

RELAXATIONS. The exponential family relaxation for r-arborescences can be computed using the directed version of Kirchhoff's Matrix-Tree Theorem. Here we present a quick informal review. Consider an exponential family with natural parameters $\boldsymbol{u} \in \mathbb{R}^{|\mathcal{E}|}$ over $\mathcal{X}$ such that the probability of $\boldsymbol{x} \in \mathcal{X}$ is proportional to $\exp(\boldsymbol{u}^\intercal \boldsymbol{x})$. Define the weights,

$$w_{ij} = \begin{cases} \exp(u_e) & \text{if } i \neq j \text{ and } \exists\, e \in \mathcal{E} \text{ from node } i \to j \\ 0 & \text{otherwise} \end{cases}. \tag{3.25}$$

Consider the graph Laplacian $\boldsymbol{A} \in \mathbb{R}^{|V| \times |V|}$ defined by

$$A_{ij} = \begin{cases} \sum_{k \neq j} w_{kj} & \text{if } i = j \\ -w_{ij} & \text{if } i \neq j \end{cases} \tag{3.26}$$

Let $\boldsymbol{A}^{r,r}$ be the submatrix of $\boldsymbol{A}$ obtained by deleting the rth row and rth column. The result by Tutte [136, p. 140] states that

$$\log\det \boldsymbol{A}^{r,r} = \log\left(\sum_{s \in \mathcal{S}(r)} \exp\left(\boldsymbol{u}^\intercal \boldsymbol{x}_s\right)\right) \tag{3.27}$$

We can use this to compute the marginals of the exponential family via its derivative [113]. In particular,

$$\mu(\boldsymbol{u}) := \left(\frac{\partial \log\det \boldsymbol{A}^{r,r}}{\partial u_e}\right)_{e \in \mathcal{E}} = \sum_{s \in \mathcal{S}(r)} \boldsymbol{x}_s \frac{\exp\left(\boldsymbol{u}^\intercal \boldsymbol{x}_s\right)}{\sum_{s' \in \mathcal{S}(r)} \exp\left(\boldsymbol{u}^\intercal \boldsymbol{x}_{s'}\right)}. \tag{3.28}$$

Again, these partial derivatives can be computed with standard software for automatic differentiation. All together, we may define the exponential family relaxation via $\boldsymbol{X}^*_\tau = \mu(\boldsymbol{U}/\tau)$.

3.5 RELATED WORK

Here we review perturbation models (PMs) and methods for relaxation more generally. SMTs are a subclass of PMs, which draw samples by optimizing a random objective. Perhaps the earliest example comes from Thurstonian ranking models [141], where a distribution over rankings is formed by sorting a vector of noisy scores. Perturb & MAP models [142, 143] were designed to approximate the Gibbs distribution over a combinatorial output space using low-order, additive Gumbel noise. Randomized Optimum models [106, 107] are the most general class, which include non-additive noise distributions and non-linear objectives. Recent work [78] uses PMs to construct finite difference approximations of the expected loss' gradient. It requires optimizing a non-linear objective over $\mathcal{X}$, and making this applicable to our settings would require significant innovation.

Using SSTs for gradient estimation requires differentiating through a convex program. This idea is not ours and is enjoying renewed interest in [81, 82, 144]. In addition, specialized solutions have been proposed for quadratic programs [76, 127, 145] and linear programs with entropic regularizers over various domains [119, 120, 133, 145, 146]. In graphical modeling, several works have explored differentiating through marginal inference [83, 117, 147–150] and our exponential family entropy relaxation builds on this work. The most superficially similar work is [151], which uses noisy utilities to smooth the solutions of linear programs. In [151], the noise is a tool for approximately relaxing a deterministic linear program. Our framework uses relaxations to approximate *stochastic* linear programs.

3.6 EXPERIMENTS

Our goal in these experiments was to evaluate the use of SSTs for learning distributions over structured latent spaces in deep structured models. We chose frameworks (NRI [77], L2X [63], and a latent parse tree task) in which relaxed gradient estimators are the methods of choice, and investigated the effects of $\mathcal{X}$, R, and $\boldsymbol{U}$ on the task objective and on the unsupervised structure discovery. For NRI, we also implemented the standard single-loss-evaluation score function estimators (REINFORCE [85] and NVIL [152]), and the best SST outperformed these baselines both in terms of average performance and variance, see results in Table C.1 in Appendix C. All SST models were trained with the "soft" SST and evaluated with the "hard" SMT. We optimized hyperparameters (including fixed training temperature

τ) using random search over 20 (NRI, unsupervised parsing) or 25 (L2X) independent runs. For each method, we selected the best model on a validation set and report its test set performance. To obtain standard errors, we bootstrap over the model selection process. Specifically, we randomly sampled with replacement 20 (NRI, unsupervised parsing) or 25 (L2X) times from the available runs and select the best model from the sampled runs on the validation set. We repeat this procedure 100,000 times to compute standard deviations over the set set metrics. We give more experimental details in B. The experiments on with NRI on graph layout were conducted entirely by Dami Choi with code available at `https://github.com/choidami/sst`. They are included in this thesis for completion and with permission. The experiments on unsupervised parsing were conducted together by Dami Choi and me.

3.6.1 *Neural Relational Inference (NRI) for Graph Layout*

	ELBO (↑)	**Edge Prec. (↑)**	**Edge Rec. (↑)**
$T = 10$			
Indep. Dir. Edges [77]	-1370 ± 20	48 ± 2	$\mathbf{93 \pm 1}$
E.F. Ent. Top $\lvert V\rvert - 1$	-2100 ± 20	41 ± 1	41 ± 1
Spanning Tree	$\mathbf{-1080 \pm 110}$	$\mathbf{91 \pm 3}$	91 ± 3
$T = 20$			
Indep. Dir. Edges [77]	-1340 ± 160	97 ± 3	$\mathbf{99 \pm 1}$
E.F. Ent. Top $\lvert V\rvert - 1$	-1700 ± 320	98 ± 6	98 ± 6
Spanning Tree	$\mathbf{-1280 \pm 10}$	$\mathbf{99 \pm 1}$	$\mathbf{99 \pm 1}$

TABLE 3.1: *Spanning Tree* performs best on structure recovery, despite being trained on the ELBO. The differences are more pronounced when data is more scarce ($T = 10$). Test ELBO and structure recovery metrics are shown from models selected on validation ELBO.

With NRI we investigated the use of SSTs for latent structure recovery and final performance. NRI is a graph neural network (GNN) model that samples a latent interaction graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and runs messages over the adjacency matrix to produce a distribution over an interacting particle system. NRI is trained as a variational autoencoder to maximize a lower bound

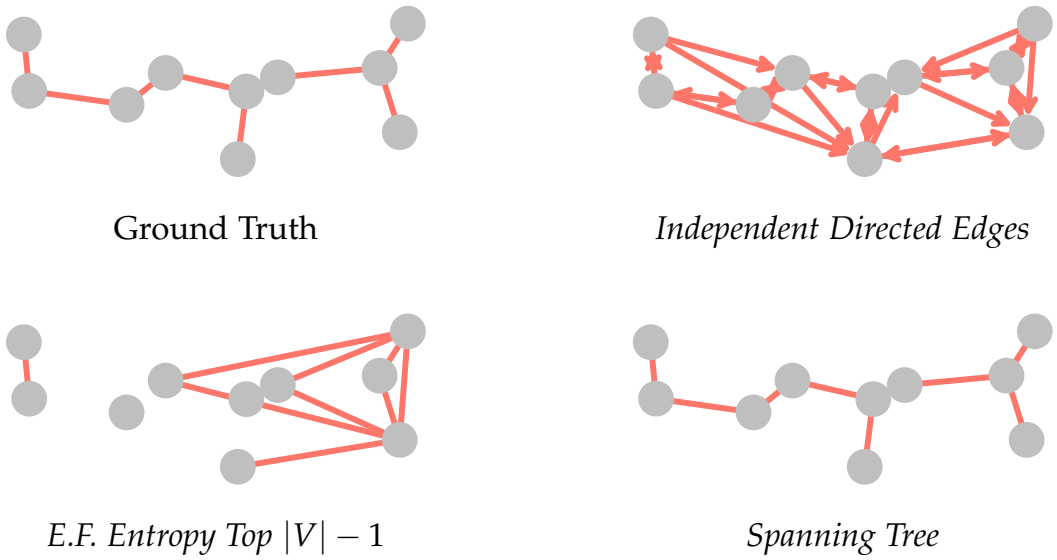


FIGURE 3.6: The stochastic softmax trick *Spanning Tree* recovers the ground truth latent graph perfectly on a test sample in neural relational inference for graph layout, while the less structured relaxations *Independent Directed Edges* and *E.F. Entropy Top* $|V| - 1$ fail.

(ELBO) on the marginal log-likelihood of the time series. We experimented with three SSTs for the encoder distribution: *Indep. Binary* over directed edges, which is the baseline NRI encoder [77], *E.F. Ent. Top* $|\mathcal{V}| - 1$ over undirected edges, and *Spanning Tree* over undirected edges. We computed the KL with respect to the random utility $\boldsymbol{U}$ for all SSTs; see Appendix B for details. Our dataset consisted of latent prior spanning trees over 10 vertices sampled from the Gumbel($\mathbf{0}$) prior. Given a tree, we embed the vertices in $\mathbb{R}^2$ by applying $T \in \{10, 20\}$ iterations of a force-directed algorithm [153]. The model saw particle locations at each iteration, not the underlying spanning tree.

We found that *Spanning Tree* performed best (Table 3.1), improving on both ELBO and the recovery of latent structure over the baseline [77]. For structure recovery, we measured edge precision and recall against the ground truth adjacency matrix. It recovered the edge structure well even when given only a short series ($T = 10$, Figure 3.6). Less structured baselines were only competitive on longer time series.

3.6.2 *Unsupervised Parsing on ListOps*

We investigated the effect of $\mathcal{X}$'s structure and of the utility distribution in a latent parse tree task. We used a simplified variant of the ListOps dataset [154], which contains sequences of prefix arithmetic expressions, e.g., `max[ 3 min[ 8 2 ]]`, that evaluate to an integer in $[0, 9]$. The arithmetic syntax induces a directed spanning tree rooted at its first token with directed edges from operators to operands. We modified the data by removing the `summod` operator, capping the maximum depth of the ground truth dependency parse, and capping the maximum length of a sequence. This simplifies the task considerably, but it makes the problem accessible to GNN models of fixed depth. Our models used a bi-LSTM encoder to produce a distribution over edges (directed or undirected) between all pairs of tokens, which induced a latent (di)graph. Predictions were made from the final embedding of the first token after passing messages in a GNN architecture over the latent graph. For undirected graphs, messages were passed in both directions. We experimented with the following SSTs for the edge distribution: *Indep. Undirected Edges*, *Spanning Tree*, *Indep. Directed Edges*, and *Arborescence* (with three separate utility distributions). *Arborescence* was rooted at the first token. For baselines we used an unstructured LSTM and the GNN over the ground truth parse. All models were trained with cross-entropy to predict the integer evaluation of the sequence.

The best performing models were structured models whose structure better matched the true latent structure (Table 3.2). For each model, we measured the accuracy of its prediction (task accuracy). We measured both precision and recall with respect to the ground truth parse's adjacency matrix. [3] Both tree-structured SSTs outperformed their independent edge counterparts on all metrics. Overall, *Arborescence* achieved the best performance in terms of task accuracy and structure recovery. We found that the utility distribution significantly affected performance (Table 3.2). For example, while negative exponential utilities induce an interpretable distribution over arborescences as demonstrated in Section 3.4, we found that the multiplicative parameterization of exponentials made it difficult to train competitive models. Despite the LSTM baseline performing well on task accuracy, *Arborescence* additionally learns to recover much of the latent parse tree.

git

3 We exclude edges to and from the closing symbol "]". Its edge assignments cannot be learnt from the task objective, because the correct evaluation of an operation does not depend on the closing symbol.

	Task Acc. (↑)	**Edge Prec. (↑)**	**Edge Rec. (↑)**
LSTM			
—	92.1 ± 0.2	—	—
GNN on latent graph			
Indep. Undirected Edges	89.4 ± 0.6	20.1 ± 2.1	45.4 ± 6.5
Spanning Tree	91.2 ± 1.8	33.1 ± 2.9	47.9 ± 5.2
GNN on latent digraph			
Indep. Directed Edges	90.1 ± 0.5	13.0 ± 2.0	56.4 ± 6.7
Arborescence			
Neg. Exp.	71.5 ± 1.4	23.2 ± 10.2	20.0 ± 6.0
Gaussian	$\mathbf{95.0 \pm 2.2}$	65.3 ± 3.7	60.8 ± 7.3
Gumbel	$\mathbf{95.0 \pm 3.0}$	$\mathbf{75.5 \pm 7.0}$	$\mathbf{71.9 \pm 12.4}$
Ground Truth	98.1 ± 0.1	100	100

TABLE 3.2: Matching ground truth structure (non-tree → tree) improves performance on ListOps. The utility distribution impacts performance. Test task accuracy and structure recovery metrics are shown from models selected on valid. task accuracy. Note that because we exclude edges to and from the closing symbol "]", recall is not equal to twice of precision for *Spanning Tree* and precision is not equal to recall for *Arborescence*.

3.6.3 *Learning To Explain (L2X) Aspect Ratings*

With L2X we investigated the effect of the choice of relaxation. We used the BeerAdvocate dataset [155], which contains reviews comprised of free-text feedback and ratings for multiple aspects (appearance, aroma, palate, and taste (Figure 3.7). Each sentence in the test set is annotated with the aspects that it describes, allowing us to define structure recovery metrics. We considered the L2X task of learning a distribution over k-subsets of words that best explain a given aspect rating.[4] Our model used word embeddings from [156] and convolutional neural networks with one (simple) and three

4 While originally proposed for model interpretability, we used the original aspect ratings. This allowed us to use the sentence-level annotations for each aspect to facilitate comparisons between subset distributions.

(complex) layers to produce a distribution over k-hot binary latent masks. Given the latent masks, our model used a convolutional net to make predictions from masked embeddings. We used k in $\{5, 10, 15\}$ and the following SSTs for the subset distribution: {*Euclid.*, *Cat. Ent.*, *Bin. Ent.*, *E.F. Ent.*} *Top k* and *Corr. Top k*. For baselines, we used bespoke relaxations designed for this task: *L2X* [63] and *SoftSub* [131]. We trained separate models for each aspect using mean squared error (MSE).

We found that SSTs improve over bespoke relaxations (Table 3.3 for aspect aroma, and show the results for other aspects in Appendix C). For unsupervised discovery, we used the sentence-level annotations for each aspect to define ground truth subsets against which precision of the k-subsets was measured. SSTs tended to select subsets with higher precision across different architectures and cardinalities and achieve modest improvements in MSE. We did not find significant differences arising from the choice of regularizer R. Overall, the most structured SST, *Corr. Top k*, achieved the lowest MSE, highest precision and improved interpretability: The correlations in the model allowed it to select contiguous words, while subsets from less structured distributions were scattered (Figure 3.7).

Pours a slight tangerine orange and straw yellow. The head is nice and bubbly but fades very quickly with a little lacing. Smells like Wheat and European hops, a little yeast in there too. There is some fruit in there too, but you have to take a good whiff to get it. The taste is of wheat, a bit of malt, and a little fruit flavour in there too. Almost feels like drinking Champagne, medium mouthful otherwise. Easy to drink, but not something I'd be trying every night.

Appearance: 3.5 **Aroma: 4.0** Palate: 4.5 Taste: 4.0 Overall: 4.0

FIGURE 3.7: For k-subset selection on the aroma aspect in learning to explain aspect ratings, the more structured stochastic softmax trick *Correlation Top k* selects contiguous words that are easier to parse and more relevant (in red), while *Top k* picks scattered words on a test sample (in blue).

	$k = 5$		$k = 10$		$k = 15$	
Relaxation	MSE	Subs. Prec.	MSE	Subs. Prec.	MSE	Subs. Prec.
			Simple model			
L2X [63]	3.6 ± 0.1	28.3 ± 1.7	3.0 ± 0.1	25.5 ± 1.2	2.6 ± 0.1	25.5 ± 0.4
SoftSub [131]	3.6 ± 0.1	27.2 ± 0.7	3.0 ± 0.1	26.1 ± 1.1	2.6 ± 0.1	25.1 ± 1.0
Euclid. Top k	3.5 ± 0.1	25.8 ± 0.8	2.8 ± 0.1	32.9 ± 1.2	2.5 ± 0.1	29.0 ± 0.3
Cat. Ent. Top k	3.5 ± 0.1	26.4 ± 2.0	2.9 ± 0.1	32.1 ± 0.4	2.6 ± 0.1	28.7 ± 0.5
Bin. Ent. Top k	3.5 ± 0.1	29.2 ± 2.0	2.7 ± 0.1	33.6 ± 0.6	2.6 ± 0.1	28.8 ± 0.4
E.F. Ent. Top k	3.5 ± 0.1	28.8 ± 1.7	2.7 ± 0.1	32.8 ± 0.5	2.5 ± 0.1	29.2 ± 0.8
Corr. Top k	$\mathbf{2.9 \pm 0.1}$	$\mathbf{63.1 \pm 5.3}$	$\mathbf{2.5 \pm 0.1}$	$\mathbf{53.1 \pm 0.9}$	$\mathbf{2.4 \pm 0.1}$	$\mathbf{45.5 \pm 2.7}$
			Complex model			
L2X [63]	2.7 ± 0.1	50.5 ± 1.0	2.6 ± 0.1	44.1 ± 1.7	2.4 ± 0.1	44.4 ± 0.9
SoftSub [131]	2.7 ± 0.1	57.1 ± 3.6	$\mathbf{2.3 \pm 0.1}$	50.2 ± 3.3	2.3 ± 0.1	43.0 ± 1.1
Euclid. Top k	2.7 ± 0.1	61.3 ± 1.2	2.4 ± 0.1	52.8 ± 1.1	2.3 ± 0.1	44.1 ± 1.2
Cat. Ent. Top k	2.7 ± 0.1	61.9 ± 1.2	$\mathbf{2.3 \pm 0.1}$	52.8 ± 1.0	2.3 ± 0.1	44.5 ± 1.0
Bin. Ent. Top k	2.6 ± 0.1	62.1 ± 0.7	$\mathbf{2.3 \pm 0.1}$	50.7 ± 0.9	2.3 ± 0.1	44.8 ± 0.8
E.F. Ent. Top k	2.6 ± 0.1	59.5 ± 0.9	$\mathbf{2.3 \pm 0.1}$	54.6 ± 0.6	2.2 ± 0.1	44.9 ± 0.9
Corr. Top k	$\mathbf{2.5 \pm 0.1}$	$\mathbf{67.9 \pm 0.6}$	$\mathbf{2.3 \pm 0.1}$	$\mathbf{60.2 \pm 1.3}$	$\mathbf{2.1 \pm 0.1}$	$\mathbf{57.7 \pm 3.8}$

TABLE 3.3: For k-subset selection on aroma aspect, SSTs tend to outperform baseline relaxations. Test set MSE ($\times 10^{-2}$) and subset precision (%) is shown for models selected on validation MSE.

3.7 DISCUSSION

We introduced stochastic softmax tricks, which are random convex programs that capture a large class of relaxed distributions over structured, combinatorial spaces. We designed stochastic softmax tricks for subset selection and a variety of spanning tree distributions. We tested their use in deep latent variable models, and found that they can be used to improve performance and to encourage the unsupervised discovery of true latent structure. There are future directions in this line of work. The relaxation framework can be generalized by modifying the constraint set or the utility distribution at positive temperatures. Some combinatorial objects might benefit from a more careful design of the utility distribution, while others, e.g., matchings, are still waiting to have their tricks designed.

4

THE GUMBEL-RAO GRADIENT ESTIMATOR

SYNOPSIS The relaxed gradient estimators presented in the previous chapter are often effective for non-supervised machine learning with discrete variables. However, as argued in chapter 2, there are applications where the relaxation of the forward-pass is undesirable and they cannot be used. For this reason, in this chapter we propose the *Gumbel-Rao* estimator, a relaxed gradient estimator for categorical variables that does not relax the forward pass. Our estimator is based on the straight-through variant of the Gumbel-Softmax estimator, but uses Rao-Blackwellization to provably reduce its variance. This offers improved properties for gradient estimation in particular when bias and variance are traded off via the temperature τ. In contrast to many other estimators that avoid relaxing the forward pass, it is easy to implement, requires minimal tuning and is cheaper. Experimentally, we demonstrate that our estimator lead to faster convergence and generally improved performance in two unsupervised latent variable tasks.

ATTRIBUTION This chapter is largely based on the following publication that was jointly authored with Chris J. Maddison and Andreas Krause.

- Paulus, M. B., Maddison, C. J. & Krause, A. *Rao-Blackwellizing the Straight-Through Gumbel-Softmax Gradient Estimator* in *International Conference on Learning Representations* (2021)

4.1 REVISITING GRADIENT ESTIMATORS FOR DISCRETE VARIABLES

In chapter 2 we reviewed gradient estimators for non-supervised machine learning. We broadly distinguished between estimators that are based on the score function and those that are based on reparameterization. Now, we consider a finer categorization that positions the gradient estimator we develop in this chapter. In Figure 4.1 we broadly partition existing estimation techniques, based on whether they require one loss evaluation or multiple loss evaluations [88, 91, 94] per estimate. Those estimators that require only a single evaluation of the loss function are particularly desirable, because loss evaluations can be costly across many applications in non-supervised machine learning. Single-evaluation estimators may be

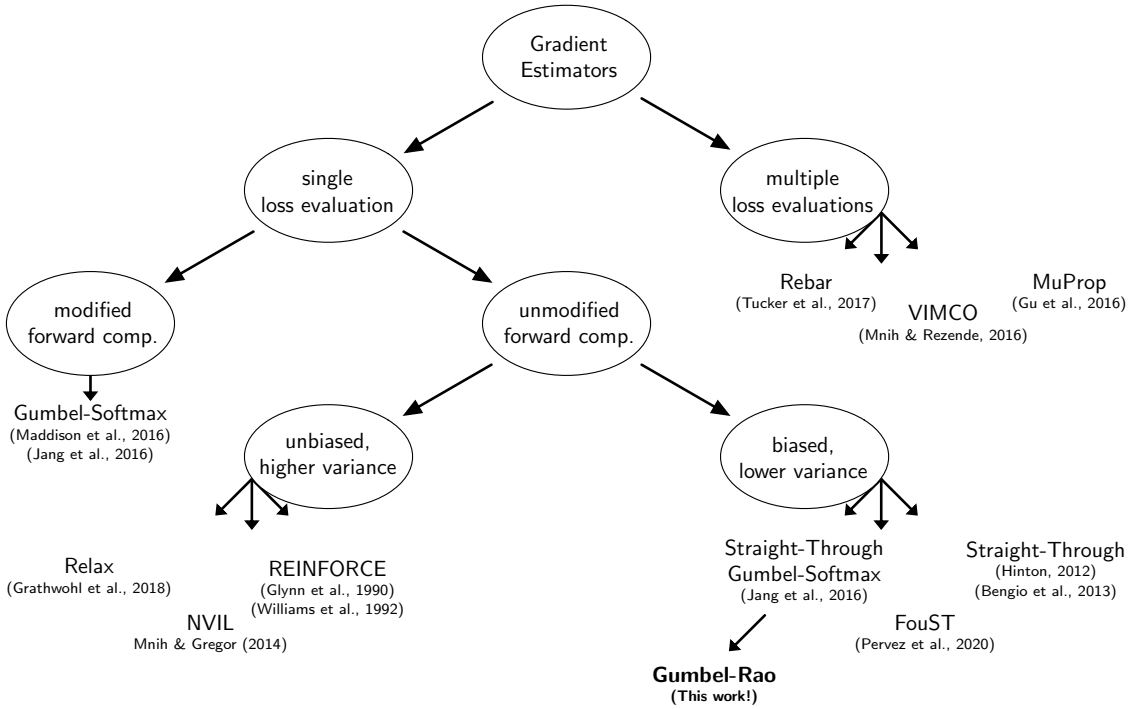


FIGURE 4.1: We taxonomize existing gradient estimators. First, we group gradient estimators by their computational overhead, distinguishing between estimators that only require a single loss evaluation and those that require multiple loss evaluations [e.g., 88, 91, 94]. Second, we distinguish between estimators that modify the forward computation [e.g., 101, 102, other relaxed estimators] and those that do not. Finally, we group estimators based on their statistical properties and distinguish between estimators that are unbiased and often higher variance [e.g., 84, 85, 92, 94, 152, score function estimators] and estimators that are biased, but often lower variance [e.g., 65, 102, 157, 158, straight-through estimators]

further categorized into those that do and do not modify the forward computation of the model. In the previous chapter, we introduced several estimators in the former group for different structured domains. These estimators relax the discrete random variable to a continuous random variable at train time. For categorical random variables, the most popular such estimator is the Gumbel-Softmax estimator from chapter 2. However, as we described there, in some applications it is desirable or required to leave the forward pass unmodified. For example, because this avoids the accumulation of errors in the forward direction or allows the model to exploit the sparsity of discrete computation [65, 73, 159]. Thus, there is a particular need for single evaluation estimators that do not modify the loss computation. These estimators may be grouped based on whether or

not they introduce bias to reduce variance. Those estimators that do not introduce bias are generally based on the score function and tend to be higher variance. We reviewed several such estimators in chapter 2. The other estimators reduce variance by introducing bias and are generally known as *straight-through* estimators [65, 102, 158, 159]. They differ in the strategies they employ to trade off bias for variance with the overall goal being to reduce the total mean squared error in gradient estimation.

4.1.1 *Straight-through estimators*

Straight-through estimators are single-evaluation estimators that leave the forward computation unmodified, but introduce bias to reduce variance and improve overall gradient estimation. Generally, the strategy is to evaluate the loss function on the discrete variable, but to perform backpropagation and estimate gradients "through" a surrogate. Thus, in the case of categorical variables $\boldsymbol{X} \sim \text{Categorical}(\boldsymbol{\theta})$, they do not require the loss function to be defined on the interior of the simplex, but they do require it to be differentiable at the corners. The original straight-through estimator was introduced by [65, 159]. For categorical distribution, this estimator takes as a surrogate the tempered probabilities of $\boldsymbol{X}$, resulting in the *slope-annealed straight-through estimator* (ST):

$$\hat{\boldsymbol{\eta}}_{\text{ST}} := \frac{\partial L(\boldsymbol{X})}{\partial \boldsymbol{X}} \frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{\theta}). \tag{4.1}$$

For binary variables, a lower bias variant of this estimator (FouST) was proposed by Pervez, Cohen & Gavves [158].

Arguably, the most popular straight-through estimator for categorical variables is known as the *straight-through Gumbel-Softmax* gradient estimator [ST-GS, 160]. This estimator is based on the Gumbel-Softmax estimator we previously considered. But it replaces the continuous relaxation $\boldsymbol{X}_\tau^*$ with the discrete variable $\boldsymbol{X}$ in the forward pass to evaluate the loss function. Both the discrete variable $\boldsymbol{X}^*$ and the relaxed variable $\boldsymbol{X}_\tau^*$ are coupled, because they share the same $\boldsymbol{U}$. For n-ary categorical distributions, the discrete variable is $\boldsymbol{X} = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{U}^\intercal \boldsymbol{x}$ for one-hot embeddings $\mathcal{X} = \{\boldsymbol{x} \in \{0,1\}^n \,|\, \sum_i x_i = 1\}$ as given in (3.1) and the relaxed variable is $\boldsymbol{X}_\tau^* = \text{softmax}_\tau(\boldsymbol{U})$ as given in (2.13) for $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$. The ST-GS estimator is given by

$$\hat{\boldsymbol{\eta}}_{\text{STGS}} := \frac{\partial L(\boldsymbol{X})}{\partial \boldsymbol{X}} \frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{U}). \tag{4.2}$$

In Figure 4.2 we illustrate the ST-GS estimator for a variational autoencoder with categorical variables on the MNIST dataset. We experimented with this model in section 4.4. The ST-GS estimator uses the discrete random variable in the forward pass to compute the loss, but uses the relaxed sample as a surrogate in the backward pass to propagate gradients.

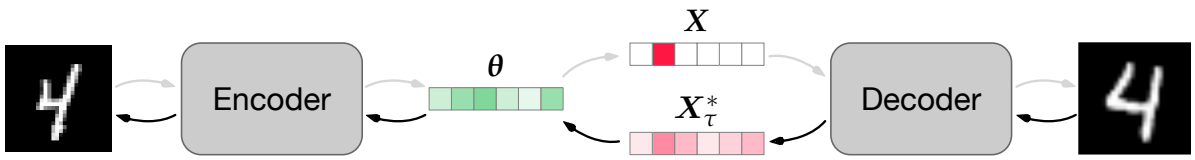


FIGURE 4.2: We sketch a categorical variational autoencoder that is trained with the Straight-Through Gumbel-Softmax estimator on the MNIST dataset [161]. The estimator uses the discrete categorical variable $\boldsymbol{X}$ in the forward pass, but uses a corresponding relaxed variable $\boldsymbol{X}_\tau^*$ in the backward pass to estimate gradients for the parameters of the encoder.

Previously, we observed that relaxed gradient estimators are biased estimators for the gradient of the objective $F(\boldsymbol{\theta})$, but unbiased estimators for the gradient of the surrogate objective $F_\tau^*(\boldsymbol{\theta})$. Notably, straight-through estimators are not known to be unbiased estimators of *any* gradient. In spite of this, they are easily implemented in any software for automatic differentiation that standardly lets users detach variables from the computation graph to interrupt gradient flow.

In the next section, we introduce a new straight-through estimator for categorical distributions that is based on a Rao-Blackwellization scheme for the straight-through variant of the Gumbel-Softmax estimator above. It enjoys all the benefits of straight-through estimators including single evaluation, unmodified forward computation and ease of implementation, but in addition provably reduces the variance of ST-GS to improve overall mean squared error in gradient estimation.

4.2 GUMBEL-RAO GRADIENT ESTIMATOR

In this section, we describe our new straight-through gradient estimator for categorical distributions. We first derive a Rao-Blackwelization scheme for the ST-GS estimator. Our approach is based on the observation that there is a *many-to-one* relationship between realizations of $\boldsymbol{U}$ and $\boldsymbol{X}$ in the Gumbel-Max trick and that the variance introduced by $\boldsymbol{U}$ can be marginalized out. The resulting estimator, which we call the *Gumbel-Rao (GR)* estimator, is

guaranteed by the Rao-Blackwell theorem to have lower variance than the ST-GS estimator. Unfortunately, in general it is not possible to analytically integrate out the utilities $\boldsymbol{U}$. However, in practice the Gumbel-Rao estimator can be efficiently approximated via Monte Carlo. Finally, we analyze our estimator in the minibatch and multi-sample setting.

4.2.1 *Rao-Blackwellization of ST-Gumbel-Softmax*

Recall that in the Gumbel-max trick, the categorical random variable is obtained via an argmax operation, i.e., $\boldsymbol{X} = \arg\max_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{U}^\intercal \boldsymbol{x}$. As previously observed, this argmax operation is non-invertible. As a result there are many input configurations of $\boldsymbol{U}$ that correspond to the same $\boldsymbol{X}$ solution. Consider an alternate factorization of the joint distribution of $(\boldsymbol{U}, \boldsymbol{X})$. Instead of sampling $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$, we could first sample $\boldsymbol{X} \sim \text{Categorical}(\boldsymbol{\theta})$, and then sample $\boldsymbol{U}$ given $\boldsymbol{X}$. In this view, the utilities are auxiliary random variables, at which the Jacobian of the tempered softmax is evaluated and which locally increase the variance of the estimator. This local variance can be removed by marginalization, i.e., by considering $\mathbb{E}_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{U}) \middle| \boldsymbol{X}\right]$. This is the key insight of our GR estimator, which is given by,

$$\hat{\boldsymbol{\eta}}_{\text{GR}} := \frac{\partial L(\boldsymbol{X})}{\partial \boldsymbol{X}} \mathbb{E}_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{U}) \middle| \boldsymbol{X}\right] = \mathbb{E}_{\boldsymbol{\theta}}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}} | \boldsymbol{X}\right] \tag{4.3}$$

where the expectation is taken over the utilities $\boldsymbol{U}$ out. The GR estimator only depends on $\boldsymbol{X}$, while the ST-GS estimator depends on $(\boldsymbol{U}, \boldsymbol{X})$. By the law of iterated expectations, GR has the same expected value as ST-GS and is an instance of a Rao-Blackwell estimator [162, 163]. However, the variance of GR estimator is generally lower than the variance of the ST-GS estimator resulting in a improved mean squared error for gradient estimation.

Proposition 1 (MSE of GR)**.** Let $\hat{\boldsymbol{\eta}}_{\text{STGS}}$ and $\hat{\boldsymbol{\eta}}_{\text{GR}}$ be the estimators defined in (4.2) and (4.3). Let $\boldsymbol{\eta}$ be the gradient defined in (2.3) that we seek to estimate. We have

$$\mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\hat{\boldsymbol{\eta}}_{\text{GR}} - \boldsymbol{\eta}\right\|^2\right] \leq \mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta}\right\|^2\right]. \tag{4.4}$$

Proof. The proposition follows from Jensen's inequality. By (4.3), we have $\mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\hat{\boldsymbol{\eta}}_{\text{GR}} - \boldsymbol{\eta}\right\|^2\right] = \mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\mathbb{E}_{\boldsymbol{\theta}}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta} | \boldsymbol{X}\right]\right\|^2\right]$ where we moved $\boldsymbol{\eta}$ inside the inner expectation. By Jensen's inequality, we have

$$\mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\mathbb{E}_{\boldsymbol{\theta}}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta} | \boldsymbol{X}\right]\right\|^2\right] \leq \mathbb{E}_{\boldsymbol{\theta}}\left[\mathbb{E}_{\boldsymbol{\theta}}\left[\left\|\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta}\right\|^2 | \boldsymbol{X}\right]\right]$$

and $\mathbb{E}_{\boldsymbol{\theta}}\left[\mathbb{E}_{\boldsymbol{\theta}}\left[\|\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta}\|^2 | \boldsymbol{X}\right]\right] = \mathbb{E}_{\boldsymbol{\theta}}\left[\|\hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta}\|^2\right]$ by the law of iterated expectations. The inequality is strict whenever $\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}} | \boldsymbol{X}\right] > 0$, where $\mathbb{V}(\cdot)$ denotes the trace of the covariance matrix. This is the case if $\tau < \infty$ and $|\theta_i| < \infty$ for all i. □

While GR is only guaranteed to reduce the variance of ST-GS, Proposition 1 guarantees that, as a function of τ, the MSE of GR is a pointwise lower bound on ST-GS. This means GR can be used for estimation at temperatures, where ST-GS has low bias but prohibitively high variance. Thus, GR extends the region of suitable temperatures over which one can tune. This allows a practitioner to explore an expanded set when trading-off of bias and variance. Empirically, lower temperatures tend to reduce the bias of ST-GS, but we are not aware of any work that studies the convergence of the derivative in the temperature limit. In our experiments in section 4.4, we observe that our estimator facilitates training at lower temperatures to improve in both bias and variance over ST-GS. Thus, our estimator retains the favourable properties of ST-GS (single, unmodified evaluation of loss $L(\cdot)$) while improving its performance.

4.2.2 *Monte Carlo Approximation*

The GR estimator requires computing the expected value of the Jacobian of the tempered softmax over the distribution $\boldsymbol{U}|\boldsymbol{X}$. Unfortunately, an analytical expression for this is only available in the simplest cases.[1] In this section we provide a simple Monte Carlo (MC) estimator with sample size $N_{\boldsymbol{U}}$ for $\mathbb{E}_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}} \text{softmax}_\tau(\boldsymbol{U}) \middle| \boldsymbol{X}\right]$, which we call the *Gumbel-Rao Monte Carlo Estimator (GR-MC)*. This estimator can be computed locally at a cost that only scales like $\mathcal{O}(nN_{\boldsymbol{U}})$ (the arity of $\boldsymbol{X}$ times the number of Monte Carlo samples $N_{\boldsymbol{U}}$).

They key property exploited by GR-MC is that $\boldsymbol{U}|\boldsymbol{X}$ can be reparameterized in the following closed form. Given a realization of $\boldsymbol{X}$ where $X_{j^\star} = 1$ and $\boldsymbol{U}_0 \sim \text{Exp}(\mathbf{1})$, we have the following equivalence in distribution [91, 164, 165].

$$U_j|\boldsymbol{X} \stackrel{d}{=} \begin{cases} -\log\left(U_j\right) + \log\sum_{i=1}^n \exp(\theta_i) & \text{if } j = j^\star \\ -\log\left(\frac{U_j}{\exp(\theta_j)} + \frac{U_{j^\star}}{\sum_{i=1}^n \exp(\theta_i)}\right) & \text{otherwise} \end{cases} \tag{4.5}$$

1 For example, in the case of $n = 2$ (binary) and $\tau = 1$ an analytical expression for the GR estimator is available.

With this in mind, we define the GR-MC estimator with $N_{\boldsymbol{U}}$ samples as:

$$\hat{\boldsymbol{\eta}}_{\text{GRMC}} := \frac{\partial L(\boldsymbol{X})}{\partial \boldsymbol{X}} \left[\frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \frac{d}{d\boldsymbol{\theta}} \operatorname{softmax}_\tau(\boldsymbol{U}^i) \right] = \frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}^i_{\text{STGS}} \tag{4.6}$$

where $\boldsymbol{X} \sim \text{Categorical}(\theta)$ and $\boldsymbol{U}^i|\boldsymbol{X}$ are sampled with the reparameterizaton in (4.5) above, such that $\hat{\boldsymbol{\eta}}^i_{\text{STGS}}$ are only conditionally on $\boldsymbol{X}$ independent. For the case $N_{\boldsymbol{U}} = 1$, our estimator reduces to the standard ST-GS estimator. The computational cost for drawing multiple samples $\boldsymbol{U}^i|\boldsymbol{X}$ scales only *linearly* in the arity of $\boldsymbol{X}$ and is usually negligible in modern applications, where the bulk of computation accrues from the computation of the loss $L(\cdot)$. Moreover, drawing multiple samples of $\boldsymbol{U}^i|\boldsymbol{X}$ can easily be parallelized on modern accelerated workstations. Our estimator remains a single-evaluation estimator under this scheme, because the loss function $L(\cdot)$ is still only evaluated at $\boldsymbol{X}$. Finally, as with GR, the GR-MC is guaranteed to improve in MSE over ST-GS.

Proposition 2. [MSE of GR-MC] Let $\hat{\boldsymbol{\eta}}_{\text{STGS}}$ and $\hat{\boldsymbol{\eta}}_{\text{GRMC}}$ be the estimators defined in (4.2) and (4.6). Let $\boldsymbol{\eta}$ be the gradient defined in (2.3) that we seek to estimate. For all $N_{\boldsymbol{U}} \geq 1$, we have

$$\mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \hat{\boldsymbol{\eta}}_{\text{GRMC}} - \boldsymbol{\eta} \right\|^2 \right] \leq \mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right] \tag{4.7}$$

Proof. The proof is similar to the proof for Proposition 1, but directly uses the linearity of expectation. Using (4.6), we have $\mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \hat{\boldsymbol{\eta}}_{\text{GRMC}} - \boldsymbol{\eta} \right\|^2 \right] = \mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}^i_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right]$. By Jensen's inequality we have

$$\mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}^i_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right] \leq \mathbb{E}_{\boldsymbol{\theta}} \left[\frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \left\| \hat{\boldsymbol{\eta}}^i_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right]$$

and $\mathbb{E}_{\boldsymbol{\theta}} \left[\frac{1}{N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{U}}} \left\| \hat{\boldsymbol{\eta}}^i_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right] = \mathbb{E}_{\boldsymbol{\theta}} \left[\left\| \hat{\boldsymbol{\eta}}_{\text{STGS}} - \boldsymbol{\eta} \right\|^2 \right]$ by the linearity of expectation. The inequality is strict under the same conditions as previously and if $N_{\boldsymbol{U}} > 1$. □

4.2.3 *Variance Reduction in Minibatches*

The variance of the GR-MC estimator can be reduced by increasing the number of Monte Carlo samples $N_{\boldsymbol{U}}$. In addition, as for any other gradient

estimator, variance may also be reduced by averaging the gradient estimator over $N_{\boldsymbol{X}}$ samples of the random variable $\boldsymbol{X}$ instead of a single one as we observed in chapter 2. This comes at the cost of additional function evaluations of the loss $L(\cdot)$. Finally, variance may be reduced by minibatching when $\boldsymbol{\theta}$ depends on an additional source of randomness, i.e., $\boldsymbol{\theta} = h(\boldsymbol{x}_{\text{sup}})$ where $\boldsymbol{x}_{\text{sup}}$ is sampled from the data distribution. This was the case in our experiments in the previous chapter and is also true for our experiments on unsupervised parsing and generative modeling in the next section of this chapter. We can sample $N_{\boldsymbol{x}_{\text{sup}}}$ examples of $\boldsymbol{x}_{\text{sup}}$ with replacement from our dataset and average the resultant gradient estimators. Again, this comes at the cost of additional function evaluations, for both $h(\cdot)$ and $L(\cdot)$.

We consider the effectiveness of the GR-MC estimator when multiple samples or minibatches are used. We study the effect of increasing $N_{\boldsymbol{x}_{\text{sup}}}$, $N_{\boldsymbol{X}}$ and $N_{\boldsymbol{U}}$ separately. The $\hat{\boldsymbol{\eta}}_{\text{STGS}}$ estimator becomes a random variable that depends on the supervised input $\boldsymbol{x}_{\text{sup}}$, the categorical variable $\boldsymbol{X}|\boldsymbol{x}_{\text{sup}}$ and the utilities $\boldsymbol{U}|\boldsymbol{X}$. The $\hat{\boldsymbol{\eta}}_{\text{GR}}$ becomes a random variable that depends on the categorical variable $\boldsymbol{X}|\boldsymbol{x}_{\text{sup}}$ and on the supervised input $\boldsymbol{x}_{\text{sup}}$. Expectations are taken over all the randomness and samples from the same distribution are drawn independently and identically.

Let $\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}$ be the following "batched" GR-MC gradient estimator

$$\hat{\boldsymbol{\eta}}_{\text{B-GRMC}} := \frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \hat{\boldsymbol{\eta}}_{\text{GRMC}}^{ij} = \frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \sum_{k=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk} \tag{4.8}$$

where $\hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk}$ is as in (4.2) for utilities $\boldsymbol{U}^{ijk}|\boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}$, categorical variable $\boldsymbol{X}^{ij}|\boldsymbol{x}_{\text{sup}}^{i}$ and supervised input $\boldsymbol{x}_{\text{sup}}^{i}$. Proposition 4 decomposes the variance of (4.8) and affords insight how it scales with the number of samples.

Proposition 4. *Let* $\hat{\boldsymbol{\eta}}_{\text{STGS}}$, $\hat{\boldsymbol{\eta}}_{\text{GR}}$ *and* $\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}$ *be the estimators defined in* (4.2), (4.3) *and* (4.8). *We have*

$$\begin{aligned}\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}\right] =& \frac{\mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}}|\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \\ &+ \frac{\mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{GR}}|\boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} + \frac{\mathbb{V}\left[\mathbb{E}\left[\hat{\boldsymbol{\eta}}_{\text{GR}}|\boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}}}\end{aligned} \tag{4.9}$$

where $\mathbb{V}$ *is the trace of the covariance matrix.*

Proof. The proposition follows a generalization of the law of total variance and is given in Appendix A.4. □

The total variance of the batched GR-MC gradient estimator decomposes into three terms. The first term in (4.9) is the average variance in gradient estimation with ST-GS due to the utilities $\boldsymbol{U}|\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}$. This variance is effectively reduced by increasing the number of Monte Carlo samples for GR-MC and approaches zero as $N_{\boldsymbol{U}} \to \infty$. However, it can also be reduced by increasing the number of categorical samples $N_{\boldsymbol{X}}$ or by increasing the number of supervised examples $N_{\boldsymbol{x}_{\text{sup}}}$, albeit at a higher cost, because this will require additional function evaluations. The second term in (4.9) is the average variance in gradient estimation with the GR estimator due to the categorical sample $\boldsymbol{X}$. This variance can be reduced by increasing the number of categorical samples $N_{\boldsymbol{X}}$ or by increasing the number of supervised examples $N_{\boldsymbol{x}_{\text{sup}}}$, because categorical samples $\boldsymbol{X}$ are drawn for every $\boldsymbol{\theta} = h(\boldsymbol{x}_{\text{sup}})$. This variance cannot be reduced by increasing the number of Monte Carlo samples $N_{\boldsymbol{U}}$. The third term in (4.9) is the variance due to the covariate $\boldsymbol{x}_{\text{sup}}$. This variance can only be reduced by increasing the number of supervised examples $N_{\boldsymbol{x}_{\text{sup}}}$. It cannot be reduced by increasing the number of Monte Carlo samples or the number of categorical samples.

Proposition 4 illustrates the practical trade-offs in variance reduction. Increasing the batchsize $N_{\boldsymbol{x}_{\text{sup}}}$ is most effective, because it reduces all variance terms in (4.9), but it is also the most expensive choice, because it requires additional function evaluations of both $h(\cdot)$ and $L(\cdot)$. In contrast, increasing the number of discrete samples $N_{\boldsymbol{X}}$ only requires an additional evaluation of the loss function $L(\cdot)$, but does not reduce the variance due to the covariate $\boldsymbol{x}_{\text{sup}}$. Finally, using GR-MC and increasing the number of Monte Carlo samples is the only choice that reduces variance without additional function evaluations. But naturally it can only reduce the variance due to the utilities $\boldsymbol{U}|\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}$. This illustrates that the effectiveness of our estimator will depend on the relative size of the three variance terms. It suggests that our estimator GRMC will tend to be more effective at small batchsizes and when a single sample $\boldsymbol{X}$ is used. It also highlights that there are diminishing returns to increasing $N_{\boldsymbol{U}}$ for fixed $N_{\boldsymbol{X}}$ and $N_{\boldsymbol{x}_{\text{sup}}}$ as is expected with Monte Carlo. In our experiments in the next section, we explore various choices for $N_{\boldsymbol{U}}$ and $N_{\boldsymbol{x}_{\text{sup}}}$ and study the effect on gradient estimation in more detail.

Finally, we note that the choice of a Monte Carlo scheme to approximate $\mathbb{E}_{\boldsymbol{\theta}}\left[\frac{d}{d\boldsymbol{\theta}} \text{softmax}_{\tau}(\boldsymbol{U}) \middle| \boldsymbol{X}\right]$ permits the use of additional well-known variance reduction methods to improve the estimation properties of our gradient estimator. For example, antithetic variates or importance sampling are sensible methods to explore in this setting [166]. For low-dimensional discrete random variables, Gaussian quadrature or other numerical methods could

be employed. However, we found the simple Monte Carlo scheme described above effective in practice and report results based on this procedure.

4.3 RELATED WORK

The idea of using Rao-Blackwellization to reduce the variance of gradient estimators for discrete latent variable models has been explored in machine learning. For example, Liu *et al.* [167] describe a sum-and-sample style estimator that analytically computes part of the expectation to reduce the variance of the gradient estimates. The favorable properties of their estimator are due to the Rao-Blackwell theorem. Kool, van Hoof & Welling [168] describe a gradient estimator based on sampling without replacement. Their estimator emerges naturally as the Rao-Blackwell estimator of the importance-weighted estimator [169] and the estimator described by Liu *et al.* [167]. Both of these estimators rely on multiple function evaluations to compute a gradient estimate. In contrast, our work is the first to consider Rao-Blackwellization in the context of a single-evaluation estimator.

.

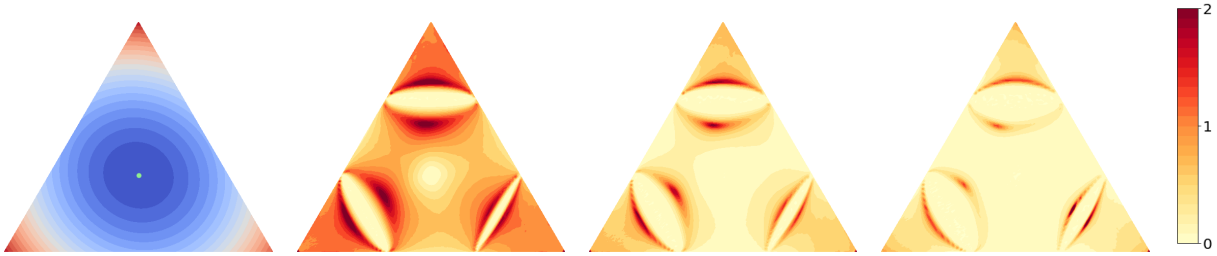
4.4 EXPERIMENTS

In this section, we study the effectiveness of our gradient estimator in practice. In particular, we evaluate its performance with respect to the temperature τ, the number of MC samples $N_{\boldsymbol{U}}$ and the batch size $N_{\boldsymbol{x}_{\text{sup}}}$. We measure the variance reduction and improvements in MSE our estimator achieves in practice, and assess whether its lower variance gradient estimates accelerate the convergence on the objective or improve final test set performance. Our focus is on single-evaluation gradient estimation and we compare against other non-relaxing estimators. This includes the tempered straight-through estimator (ST), the FouST estimator [158] for binary variables, the straight-through Gumbel-Softmax estimator (ST-GS), and the score function estimator in (2.7) (SCORE) with a running mean as a baseline. We also compare against the relaxing Gumbel-Softmax estimator (GS) where possible.

First, we consider a toy example which allows us to explore and visualize the variance of our estimator and suggests that it is particularly effective at low temperatures. Next, we evaluate the effect of τ and $N_{\boldsymbol{U}}$ in a latent parse tree task which does not permit the use of relaxed gradient estimators. For ease of notation, we abbreviate the Gumbel-Rao estimator with $N_{\boldsymbol{U}}$ Monte

Carlo samples as GR-MCN_U. Here, our estimator facilitates training at low temperatures to improve overall performance and is effective even with few MC samples. Finally, we train variational autoencoders with categorical latent variables [49, 95]. Our estimator yields improvements at small batch sizes and obtains competitive or better performance than the GS estimator at the largest arity.

4.4.1 *Quadratic Programming on the Simplex*



Objective function | $\Delta \log_{10}(\mathbb{V})$ (τ=0.1) | $\Delta \log_{10}(\mathbb{V})$ (τ=0.5) | $\Delta \log_{10}(\mathbb{V})$ (τ=1.0)

FIGURE 4.3: Our estimator (GR-MC) effectively reduces the variance over the entire simplex and is particularly effective at low temperatures. Contours for the quadratic program in three dimensions (left) and difference in log10-trace of the covariance matrix between ST-GS and GR-MC1000 at different temperatures. Warmer means difference is larger.

As a toy problem, we consider the problem of minimizing a quadratic program $(\boldsymbol{x}-\boldsymbol{b})^\top \boldsymbol{Q}(\boldsymbol{x}-\boldsymbol{b})$ over the probability simplex $P = \{\boldsymbol{x} \in \mathbb{R}^n : x_i \geq 0, \sum_{i=1}^n x_i = 1\}$ for $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ positive-definite and $\boldsymbol{b} \in \mathbb{R}^n$. This problem can be cast as the following stochastic optimization problem,

$$\min_{\boldsymbol{p}\in P} \mathbb{E}_{\boldsymbol{p}}[L(\boldsymbol{X}, \boldsymbol{p})] = \min_{\boldsymbol{p}\in P} \mathbb{E}_{\boldsymbol{p}}[(\boldsymbol{X}-\boldsymbol{b})^\top \boldsymbol{Q}'(\boldsymbol{p}, \boldsymbol{b})(\boldsymbol{X}-\boldsymbol{b})]$$

where $\boldsymbol{X} \sim \text{Categorical}(\boldsymbol{\theta})$ with $\boldsymbol{\theta} = \log(\boldsymbol{p})$. The stochastic objective matrix Q' is defined as

$$Q'_{ij} = \begin{cases} \frac{(p_i - b_i)^2}{p_i - 2p_i b_i + b_i^2} Q_{ii} & \text{if } i = j \\ \frac{(p_i - b_i)(p_j - b_j)}{b_i b_j - p_i b_j - b_i p_j} Q_{ij} & \text{if } i \neq j \end{cases}$$

While solving the above problem is simple using standard methods, it provides a useful testbed to evaluate the effectiveness of our variance

	$T \leq 10$			$T \leq 50$		
Estimator	$\tau = 0.01$	$\tau = 0.1$	$\tau = 1.0$	$\tau = 0.01$	$\tau = 0.1$	$\tau = 1.0$
ST-GS	38.8	59.3	65.8	46.8	56.8	59.6
GR-MC10	66.4	66.9	66.7	58.7	59.1	59.6
GR-MC100	65.6	66.3	65.9	59.6	59.1	59.6
GR-MC1000	**66.5**	***67.1***	**67.0**	***60.0***	**59.8**	**59.9**

TABLE 4.1: Our estimator (GR-MCK) facilitates training at lower temperatures with improved performance on the latent parse tree task. Best test classification accuracy on the ListOps dataset selected on the validation set. Best estimator at given temperature in bold, best estimator across temperatures in italics. Higher is better.

reduction scheme. For this purpose, we consider $Q_{ij} = \exp(-2|i-j|)$ and $b_i = \frac{1}{3}$ in three dimensions.

Our estimator reduces the variance in the gradient estimation over the entire simplex and is particularly effective at low temperatures in this problem. In Figure 4.3, we compare the log10-trace of the covariance matrix of ST-GS and GR-MC1000 at three different temperatures and display their difference over the entire domain. The improvement is universal. The pattern is not always intuitive (oval bull's eyes), despite the simplicity of the objective function. Compared with ST-GS, our estimator on this example appears more effective closer to the corners and edges, which is important for learning discrete distributions. At lower temperatures, the difference between the two estimators becomes particularly acute. This suggests that our estimator may train better at lower temperatures and be more responsive to optimizing over the temperature to successfully trade off bias and variance.

4.4.2 *Unsupervised Parsing on ListOps*

Straight-through estimators feature prominently in NLP [170] where latent discrete structure arises naturally, but the use of relaxations is often infeasible. Therefore, we evaluate our estimator in a latent parse tree task on subsets of the ListOps dataset [154]. This dataset contains sequences of prefix arithmetic expressions $\boldsymbol{x}_{\text{sup}}$ (e.g., `max[ 3 min[ 8 2 ]]`) that evaluate to

an integer $y \in \{0, 1, \dots 9\}$. The arithmetic syntax induces a latent parse tree $\boldsymbol{X}$. We consider the model by [73] that learns a distribution over plausible parse trees of a given sequence and use the following objective

$$\min_{\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{sup}}} \mathbb{E}_{\boldsymbol{\theta}}[L(\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\text{sup}}, y)] = \mathbb{E}_{\boldsymbol{\theta}}\left[-\log \hat{p}(y|\boldsymbol{X}, \boldsymbol{\theta}_{\text{sup}})\right]$$

Both the conditional distribution over parse trees $q_{\boldsymbol{\theta}}(\boldsymbol{X}|y)$ and the classifier $\hat{p}_{\boldsymbol{\theta}_{\text{sup}}}(y|\boldsymbol{X}, \boldsymbol{x}_{\text{sup}})$ are parameterized using neural networks. In this model, a parse tree $\boldsymbol{X}$ for a given sentence is sampled bottom-up by successively combining the embeddings of two tokens that appear in a given sequence until a single embedding for the entire sequence remains. This is then used for performing the subsequent classification. Because it is computationally infeasible to marginalize over all trees, Choi, Yoo & Lee [73] rely on the ST-GS estimator for training. The decision which two (consecutive) embeddings to combine is modeled as a categorical random variable. We compare this estimator against our estimator GR-MC$N_{\boldsymbol{U}}$ with $N_{\boldsymbol{U}} \in \{10, 100, 1000\}$. We consider temperatures $\tau \in \{0.01, 0.1, 1.0\}$ and experiment with shallow and deeper trees by considering sequences of length T up to 10, 25 and 50. All models are trained with stochastic gradient descent with a batch size equal to the maximum T. Because we are interested in a controlled setting to investigate the effect of τ and $\mathbb{N}_{\boldsymbol{U}}$, our experimental set-up is significantly simpler than elsewhere [e.g., 171]. We give details and highlight important differences in Appendix B.

Our estimator facilitates training at lower temperatures and achieves better final test set accuracy than ST-GS (Table 4.1). Increasing $N_{\boldsymbol{U}}$ improves the performance at low temperatures, where the differences between the estimators are most pronounced. Overall, across all temperatures this results in modest improvements, particularly for shallow trees and small batch sizes. We also find evidence for diminishing returns: The differences between ST-GS and GR-MC10 are larger than between GR-MC100 or GR-MC1000, suggesting that our estimator is effective even with few MC samples.

4.4.3 *Generative Modeling with Categorical Variational autoencoders*

Finally, we train variational autoencoders [49, 95] with categorical latent random variables on the MNIST dataset of handwritten digits [161]. We used

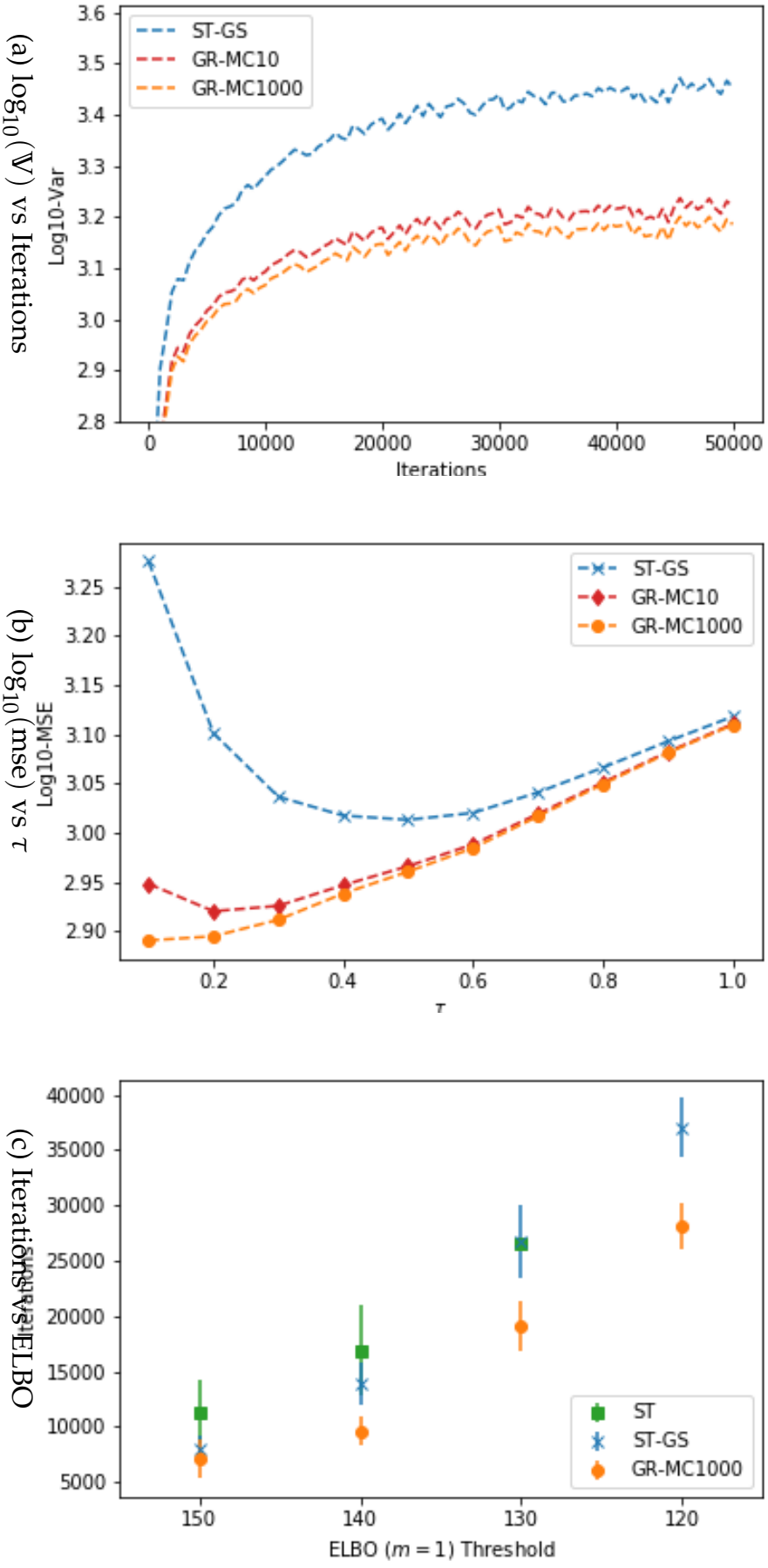


FIGURE 4.4: Our estimator (GR-MCK) effectively reduces the variance over the entire training trajectory (4.4a), achieves a lower mean squared error at a lower temperature (4.4b) and converges faster than ST and ST-GS on the discrete VAE objective (4.4c). Log10-trace of the covariance matrix over a training trajectory (4.4a) and log10-MSE (4.4b) at different temperatures during training, average number of iterations and standard error to reach various thresholds of the objective on the validation set (4.4c).

the fixed binarization of [172] and the standard split into train, validation and test sets. Our objective is

$$\min_{\boldsymbol{\theta},\boldsymbol{\theta}_{\mathbf{sup}}} \mathbb{E}_{\boldsymbol{\theta}}[L(\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\mathbf{sup}}, y)] = \mathbb{E}_{\boldsymbol{\theta}}\left[\log\left(\frac{1}{N_{\boldsymbol{X}}}\sum_{i=1}^{N_{\boldsymbol{X}}}\frac{\hat{p}(\boldsymbol{x}_{\text{sup}}|\boldsymbol{X}^i, \boldsymbol{\theta}_{\mathbf{sup}})}{q_{\boldsymbol{\theta}}(\boldsymbol{X}^i|\boldsymbol{x}_{\text{sup}})}\right)\right]$$

where $\boldsymbol{x}_{\text{sup}}$ denotes the input image and $\boldsymbol{X}^i \sim q_{\boldsymbol{\theta}}(\boldsymbol{X}|\boldsymbol{x}_{\text{sup}})$ denotes a vector of categorical latent random variables. This variational objective is an upper bound on the negative log-likelihood for all choices of $N_{\boldsymbol{X}} \geq 1$ [173]. For training, the bound is approximated using only a single sample ($N_{\boldsymbol{X}} = 1$). For final validation and testing, we use 5000 samples ($N_{\boldsymbol{X}} = 5000$). Both the generative model $\hat{p}(\boldsymbol{x}_{\text{sup}}|\boldsymbol{X}^i, \boldsymbol{\theta}_{\mathbf{sup}})$ and the variational distributions $q_{\boldsymbol{\theta}}(\boldsymbol{X}^i|\boldsymbol{x}_{\text{sup}})$ were parameterized using neural networks. We experiment with different batch sizes and categorical random variables of arities in $\{2, 4, 8, 16\}$ as in Maddison, Mnih & Teh [174]. To facilitate comparisons, we do not alter the total dimension of the latent space and train all models for 50,000 iterations using stochastic gradient descent with momentum. Hyperparameters are optimised for each estimator using random search [175] over twenty independent runs. Additional details are given in Appendix B.4.

Our estimator effectively reduces the variance over the entire training trajectory (Figure 4.4a). Even a small number of MC samples ($N_{\boldsymbol{U}} = 10$) results in sizable variance reductions. The variance reduction compares favorably to the magnitude of the minibatch variance (Figure C.1 in Appendix C). Empirically, we find that lower temperatures tend to reduce bias. Our estimator facilitates training at lower temperatures and thus features a lower MSE (Figure 4.4b). During training our estimator can trade off bias and variance to improve the gradient estimation. Empirically, we observed that on this task, the best models using ST-GS trained at an average temperature of 0.65, while the best models using GR-MC1000 trained at an average temperature of 0.35. This is interesting, because it indicates that our estimator may make the use of temperature annealing during training more effective. We find lower variance gradient estimates improve convergence of the objective (Figure 4.4c). GR-MC1000 reaches various performance thresholds on the validation set with reliably fewer iterations than ST or ST-GS. This effect is observable at different arities and persistent over the entire training trajectory.

For final test set performance, our estimator outperforms REINFORCE and all other straight-through estimators (Table 4.2). The improvements over ST-GS extend up to two nats (for batch size 20, 16-ary) at small batch sizes and are more modest at large batch sizes as expected (Table C.5 in Ap-

	Binary		4-ary		8-ary		16-ary	
$N_{\boldsymbol{x}_{\text{sup}}}$	20	200	20	200	20	200	20	200
GS	*98.2*	*96.4*	*95.7*	93.8	*95.5*	92.3	*96.8*	94.3
SCORE	202.6	121.4	173.7	122.2	203.9	124.9	169.4	129.5
ST	105.5	103.1	106.2	104.5	107.2	105.1	108.2	104.5
FouST	101.5	97.8	-	-	-	-	-	-
ST-GS	100.7	97.1	99.1	93.7	98.0	92.8	98.8	92.6
GR-MC10	100.7	97.4	97.8	93.8	97.4	93.1	97.9	92.4
GR-MC100	100.6	**96.8**	**97.5**	94.0	96.8	***92.2***	97.3	92.4
GR-MC1000	***100.5***	97.0	97.6	***93.5***	***96.5***	92.5	***96.8***	***92.2***

TABLE 4.2: Our estimator (GR-MC*K*) outperforms other straight-through estimators for discrete-latent-space VAE objectives on the MNIST dataset and is competitive with the Gumbel-Softmax (*GS*) at large arities. Best bound on the test negative log-likelihood selected on the validation set. Best straight-through estimator in bold, best estimator in italics. Lower is better.

pendix C). This confirms that our estimator might be particularly effective in settings, where training at high batch sizes is prohibitively expensive. The improvements from increasing the number of MC samples tend to saturate at $N_{\boldsymbol{U}} = 100$ on this task. Further, our results suggest that relaxed estimators may be preferred (if they can be used) for categorical random variables of smaller arity. For example, the GS estimator outperforms all straight-through estimators for binary variables for both batch sizes. For large arities however, we find that straight-through estimators can perform competitively: Our estimator GR-MC1000 achieves the best performance overall and outperforms the GS estimator for 16-ary variables.

4.5 DISCUSSION

In this chapter, we presented the *Gumbel-Rao* estimator, a new single-evaluation non-relaxing gradient estimator for non-supervised machine learning with categorical random variables. Our estimator is a Rao-Blackwellization

of the state-of-the-art straight-through Gumbel-Softmax estimator. It enjoys lower variance and can be implemented efficiently using Monte Carlo methods. In particular and in contrast to most other work, it does not require additional evaluations of the loss function. Experimentally, we demonstrated that our estimator improved final test set performance in an unsupervised parsing task and on a variational autoencoder loss. It accelerated convergence on the objective and compared favorably to other standard gradient estimators. Even though the gains were sometimes modest, they were persistent and particularly pronounced when models must be trained at low temperatures or with small batch sizes. We expect that our estimator will be most effective in these settings and that further improvements may be achieved when combining *Gumbel-Rao* with an annealing schedule for the temperature.

The approach we presented in this chapter may be extended to the gradient estimators for structured domains we introduced in the previous chapter. It is possible to use Rao-Blackwellization to reduce the variance of their corresponding straight-through variants, whenever an efficient reparameterization of the perturbation conditional on the discrete structure $\boldsymbol{X}$ is available [107]. For example, this is the case for the stochastic softmax trick for subset selection (on the unit cube) we presented. In other cases, it may be possible to achieve *partial* variance reduction by conditioning both $\boldsymbol{X}$ and $\boldsymbol{U}$ on an auxiliary random variable. For example, in the case of k-subset selection with $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ it is difficult to simulate $\boldsymbol{U}|\boldsymbol{X}$ efficiently, but easier to simulate $\boldsymbol{U}|\sigma$ where σ is the order in which elements are added to $\boldsymbol{X}$ if $\boldsymbol{X}$ is sampled from the Plackett-Luce model. Finally, the results in this chapter have inspired subsequent work in gradient estimation for categorical variables. On the theoretical side, [176] derives an analytic expression of the Gumbel-Rao estimator as $\tau \to 0^+$ and shows that it recovers the arithmetic mean of the ST estimator and the DARN [87] estimator. On the practical side, [177] develops an estimator based on the GR estimator that eliminates the variance from sampling $\boldsymbol{U}$ without the need for a Monte Carlo approximation.

Part II

LEARNING FOR DISCRETE OPTIMIZATION

5

BACKGROUND

5.1 MIXED INTEGER LINEAR PROGRAMS

Mixed integer linear programs are optimization problem in which some variables are restricted to take integer values and others are continuous, while the objective and constraints are linear. They are important, because many real-world problems involve indivisible choices or discrete decisions that can be modeled using integer variables. As a result, integer programs are applied broadly across numerous industrial areas, including operations research, energy and finance, among many others. For example, the operator of a distributed compute cluster may be required to balance incoming workloads robustly across as few servers as possible. This server load balancing problem can be formulated as a mixed integer linear program and we will consider it in some of our experiments in the following chapters. In energy, deciding which power generation units to commit (operate) in order to satisfy the demand for electricity at low cost over a given time horizon is known as the unit commitment problem. It is also a mixed integer linear program. In finance, mixed integer programs can be used to optimize portfolio allocation and make investment decisions. Apart from industrial applications, integer programs include various classes of combinatorial problems, e.g., finding set covers or determining the largest independent set in a graph. Here, general-purpose methods for integer programming complement exact or approximate procedures that are known for special problem classes. Finally, mixed integer programs have attracted interest in machine learning, where they can be used for MAP estimation [25], object recognition [178], clustering [179] or to verify the robustness of a neural network to adversarial examples [27, 28].

In this thesis, we write a mixed integer linear program in standard form with variable bounds as

$$z^\star := \min_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{c}^\intercal \boldsymbol{x}, \qquad \mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\ \underline{\boldsymbol{\pi}} \leq \boldsymbol{x} \leq \overline{\boldsymbol{\pi}},\ x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}\} \tag{5.1}$$

The linear objective is $\boldsymbol{c} \in \mathbb{R}^n$ and the m linear constraints are specified by $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$. In addition, the n variables may be lower or upper-bounded as determined by $\underline{\pi}_j \in \mathbb{R} \cup \{-\infty\}$ and $\overline{\pi}_j \in \mathbb{R} \cup \{\infty\}$, but

a bound may be void, i.e., $\underline{\pi}_j = -\infty$ or $\overline{\pi}_j = \infty$ or both. Finally, the set $\mathcal{I} \subseteq \{1, \ldots, n\}$ indexes the integer variables that are restricted to be integral. Unfortunately, mixed integer linear programs are generally NP-hard to solve. A variety methods exists to solve them, either approximately or exactly, including cutting plane methods, heuristics and specific methods for particular problem classes [e.g., 180]. In this thesis, we focus on variants of branch and bound search which is universally applicable and the backbone of state-of-the-art solvers for integer programming [181, 182].

5.2 BRANCH AND BOUND SEARCH

Branch and bound search [34] is a general-purpose method for integer programming. Branch and bound recursively builds a search tree, whose root represents the original problem in (5.1). Child nodes are created via branching, this means by introducing additional constraints to partition the set of feasible solutions. Typically, branching creates two children from any of the tree's leaf nodes by splitting the domain of a single integer variable. Thus, any descendant node constitutes a mixed integer linear program itself, but deeper nodes have a smaller feasible set, because more variables are restricted to tighter domains. Branch and bound uses bounds on the optimal solution $z^\star$ to prune the tree and direct the search. A (local) lower bound can be obtained from the linear program (LP) relaxation for any node in the tree. The LP relaxation of any integer program is obtained simply by removing the integrality constraints $x_j \in \mathbb{Z}$ for all $j \in \mathcal{I}$. For example, the LP relaxation for the root node is

$$z^* := \min_{\boldsymbol{x} \in P} \boldsymbol{c}^\mathsf{T} \boldsymbol{x}, \qquad P = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},\ \underline{\boldsymbol{\pi}} \leq \boldsymbol{x} \leq \overline{\boldsymbol{\pi}}\} \tag{5.2}$$

Naturally, this gives a global lower bound on $z^\star$. After branching and at any time during the search, a tighter bound may be obtained from the minimum bound over all leaf nodes. This bound is also known as the *dualbound* and we will denote it by $\check{z}$. Moreover, any feasible solution for the program in (5.1) provides a global upper bound on the optimal solution $z^\star$. The best such bound available at any time during the search is known as the *primalbound* and we will denote it by $\mathring{z}$. Branch and bound search is powerful, because nodes with a high lower bound can be pruned and must not be explored, if a lower or equal primalbound is known.

5.2.1 *Cutting Planes and Primal Heuristics*

In practice, the performance of branch and bound is closely tied to the quality of its bounds. Therefore, modern solvers implement variants of branch and bound that aim at improving the primal- and dualbound. Arguably, the two most important such variations are the use of *cutting planes* [183] and *primal heuristics* [184].

Cutting plane methods are used to improve the dualbound of the solver. A cutting plane is a linear inequality constraint that for a given mixed integer linear program separates the convex hull of feasible solutions from the LP solution of its relaxation. For example, at the root node a cutting plane $\chi := (\boldsymbol{\alpha}, \beta)$ satisfies $\boldsymbol{\alpha}^\intercal z^* > \beta$ and $\boldsymbol{\alpha}^\intercal \boldsymbol{x} \leq \beta$ for all $\boldsymbol{x} \in \mathcal{X}$. Naturally, adding a cutting plane to the linear programming relaxation shrinks its feasible region and thus may tighten the bound. However, it also increases the size of the linear program and makes it more costly to resolve. Therefore, while a variety of separation methods to generate cutting planes exist, it is undesirable to add all the cuts they produce. In practice, modern solvers rely on a heuristic procedure to select only some cutting planes and add them to the relaxation. In chapter 6, we show how learning can be leveraged to design application-specific procedures for cutting plane selection that yield tangible improvements over existing selection procedures.

In contrast, primal heuristics are procedures that are designed to find integer feasible solutions. Hence, they may improve the primalbound and a variety of primal heuristics has been developed for this purpose. The most important ones are problem-specific methods [e.g., 185, 186], rounding procedures [e.g., 187, 188], variants of large neighborhood search [e.g., 189–192] or diving heuristics. Berthold [184] gives a broad overview. Again, the design of these methods has been mostly ad-hoc. Primal heuristics are usually generic and fail to exploit structural commonality between similar problem instances that often arise in practice. In chapter 7, we show how learning can be leveraged to design application-specific diving heuristics that can improve the overall performance of branch and bound solvers.

Thus, in this thesis we present methods that leverage learning for improving both the primal- and the dualbound of the solver. These methods are complementary; they may be combined with each other or with additional (learnt) components. This sets the stage for the design of increasingly learning-based methods for solving mixed integer linear programs that we believe will expand the capacity of today's branch and bound solvers.

5.2.2 *Solver Performance*

Branch and bound search is guaranteed to terminate and find the optimal solution when given enough computational resources (time and memory). Unfortunately, the computational requirements of branch and bound may grow exponentially with the size of the problem. Nevertheless, branch and bound is widely used. Modern implementations of branch and bound are capable of efficiently solving problems of large scale and industrial relevance. Even when this is not possible, branch and bound remains a valuable approach, because the search can be terminated prematurely and still yield guarantees on the optimal solution via the primal- and dualbound.

PRIMAL-DUAL GAP When a program can be solved to completion, the most intuitive way to assess the overall performance of the solver is by measuring *solving time*, i.e., the average time it takes to solve an instance or a set of instances. In the cases, where this is not possible, it is common to consider the *primal-dual gap*[1]

$$\gamma_{pd}(\mathring{z}, \check{z}) = \begin{cases} \frac{|\mathring{z}-\check{z}|}{\max(|\mathring{z}|,|\check{z}|)} & \text{if } 0 < \mathring{z}\check{z} < \infty \\ 1 & \text{else} \end{cases} \tag{5.3}$$

where $\mathring{z}$ and $\check{z}$ are the primal- and dualbound respectively as before. Performance can be measured by solving instances with a fixed cutoff time T and then computing the primal-dual gap $\gamma_{pd}(\mathring{z}_T, \check{z}_T)$, where $\mathring{z}_t$ and $\check{z}_t$ denote the solver's primal- and dualbound at time t (if non-existent, then $-\infty$ and $+\infty$ respectively).

PRIMAL-DUAL INTEGRAL Unfortunately, measuring the primal-dual gap at time T is susceptible to the particular choice of cutoff time. This is particularly troublesome, because the lower and upper bounds of branch and bound solvers tend to improve in a stepwise fashion. In order to alleviate this issue, it is common to integrate the primal-dual gap over the solving time and measure the primal-dual integral

$$\Gamma_{pd}(T) = \int_{t=0}^{T} \gamma_{pd}(\mathring{z}_t, \check{z}_t) dt \tag{5.4}$$

1 Note that we use a definition of the primal-dual gap that is used in SCIP 7.0.2 for the computation of the integral and do not quantify the gap or the integral in per cent.

Sometimes, we may not be interested in measuring overall solver performance, but want to specifically consider *primal performance* or *dual performance* instead. This may be, because a particular application only requires a lower or an upper bound on the optimal solution, or because we want to isolate the effect of a particular innovation that is designed to improve the primal- or dualbound.

PRIMAL GAP To assess primal performance and the quality of the feasible solution $\boldsymbol{x} \in \mathcal{X}$ with $\mathring{z} = \boldsymbol{c}^\intercal \boldsymbol{x}$, we can measure the *primal gap*

$$\gamma_p(\mathring{z}) = |\mathring{z} - z^\star| \tag{5.5}$$

Sometimes, the primal gap is normalized by $|z^\star|$ or $|\mathring{z}|$ which can be useful when $\gamma_p(\cdot)$ is averaged across disparate instances. We will use the primal gap to compare the primal performance of different diving heuristics in chapter 7, but do not normalize the primal gap.

DUAL GAP To assess dual performance of a solver or method, for any dualbound $\check{z}$ we can measure the *dual gap*:

$$\gamma_d(\check{z}) = |z^\star - \check{z}| \tag{5.6}$$

Sometimes, the dual gap $\gamma_d(\cdot)$ is normalized by $z^\star$ or $\check{z}$. We will use the dual gap to compare different methods for cutting plane selection in chapter 6. These methods will iteratively improve the dual gap over multiple rounds and we will normalize the dual gap over all rounds against its initial value. There, we will also consider the dual integral from summing the dual gap over all rounds.

5.2.3 *SCIP solver*

Unfortunately, state-of-the-art solvers for integer programming are closed-source [181, 182]. This obstructs research to integrate machine learning into branch and bound search. In this thesis, we use the branch and bound solver SCIP [193] for our experiments. This solver is open-source and performant. We extend the source and interface of the solver to expose separation algorithms and cutting plane selection in chapter 6 and the diving heuristic plug-in in chapter 7. We compare our methods against default and tuned versions of this solver. We reasonably expect that solver performance could further improve, if our methods were integrated into state-of-the-art solvers and hope that this will be facilitated in the future.

5.3 MACHINE LEARNING FOR BRANCH AND BOUND

Branch and bound solvers are general-purpose and as such they are typically calibrated on large sets of diverse classes of mixed integer linear programs. However, in most practical applications of interest it is common to repeatedly solve *similar* problems of the same type. For example, the operator of the distributed computer cluster has the same servers at his disposal and at least some workloads may be regularly recurring. Likewise, the energy provider will optimize demand over the same grid each day and the investment advisory makes decision subject to the same constraints on portfolio allocation. In machine learning too, applications of integer programming are both many and related by design. This is because it is common to deploy a model to several examples. The underlying assumption is that these examples are sampled from the same data generating distribution. As each example gives rise to an integer programming instance and all instances share the same model, integer programming instances in machine learning are necessarily many and related. This presents an opportunity to improve the performance of branch and bound solvers by considering specific applications with similar and multiple instances. This focus of this thesis is on this highly practical setting where as we demonstrate in the following two chapters the use of machine learning can yield substantial improvements.

In principle, branch and bound solvers are highly customizable and therefore could be tailored to particular applications. For example, the open-source solver SCIP 7.0.2 contains over 2,500 parameters that can be set by the user to govern the brand and bound search. A plausible approach to tailor the solver to a specific application and improve performance is therefore to find a better configuration of these parameters. Because performing this task manually is difficult, several strategies for the automatic configuration of branch and bound solvers have been developed, among the most promising ones are those that are based on learning [see e.g., 41, 194, 195]. However, these strategies suffer from at least two shortcomings. First, the parameters that are exposed by the solver immediately restrict the variants of branch and bound that may be explored by algorithm configuration. For example, the SCIP 7.0.2 solver selects cutting planes based on a score for each cut that is restricted to be the weighted average of a cut's efficacy, integral support and parallelism with the objective. Algorithm configuration may vary the corresponding composite weights, but cannot explore alternative criteria for cutting plane selection. This is made worse, because a configuration

is rarely adapted during a solver call, even though this may be beneficial [196]. With respect to primal heuristics, algorithm configuration in SCIP 7.0.2 may explore an ensemble of heuristics by varying the criteria for calling them. But it cannot design novel heuristics that may be particularly suited for a specific application. Second, most approaches for automatic algorithm configuration are difficult to scale. They are inherently sequential: Typically, they propose several promising configurations, test them by calling the solver and update the proposals based on measures of solver performance, e.g., runtime or primal- and dualbounds. This procedure must often be repeated many times and is difficult to parallelize, while solver calls are expensive and incur a significant cost (computation and time) at each iteration. This is made worse when these methods are employed on distributed compute clusters as tends to be necessary with modern applications of interest. There, measures of solver performance can vary significantly [197] which may increase the number of solver calls that is required to identify a good configuration. In this thesis, we explore an alternative strategy that addresses these shortcomings. Instead of learning to configure the solver, our focus is on learning to perform specific sub-routines in branch and bound. We directly incorporate our models into the solver where they replace existing ad-hoc methods. In contrast to algorithm configuration, this approach offers two advantages. First, it is not limited by the existing parameterization of the solver. For example, in chapter 6 we consider cutting plane selection. Our model predicts a score for each cut based on the current state of the solver and features of the respective cutting plane. Naturally, this score is not required to be a composite as above and dynamically changes with the current solver state. In chapter 7 we use machine learning to design custom diving heuristics for specific applications. They find better solutions than existing divers and outperform a *tuned* diving ensemble to improve overall solver performance on real-world applications. Second, this approach can be much easier to scale. This is, because it is possible to learn models for sub-routines without relying on complete solver calls. In the following two chapters, we demonstrate that models for both cutting plane selection and diving may be learnt from data that can be collected more accurately and often at a lower cost than solver performance measures. Moreover, data collection is embarrassingly parallel. The data can be collected offline and only once ahead of model training. There is no need to call the solver *during* training.

The idea to use machine learning inside branch and bound has been considered by others. Several works learn models for variable selection

in branching [198–203]. Others focus on node selection in the search tree [204, 205], deal with cutting plane management [39, 206–209] or consider primal heuristics [e.g., 12, 197, 210–212]. A popular approach is to use imitation learning. For example, Khalil *et al.* [198], Alvarez, Louveaux & Wehenkel [199], Gasse *et al.* [200], and Gupta *et al.* [201] learn models for variable selection by imitating the strong branching rule. For each candidate variable, this rule computes the dual bound improvements that would result from branching on it, and then selects the best candidate. While this rule typically yields trees of small size, it is too expensive for deployment and cheaper heuristics are typically used instead. Learning to imitate strong branching offers the opportunity to amortize the cost of strong branching; these models can deliver performant branching decisions at lower cost than strong branching. In chapter 6 we propose an approach for cutting plane selection that is based on imitation learning. Several other methods are based on supervised learning. For example, some authors train generative models to predict solutions of an integer program that can be used to design primal heuristics [197, 212, 213]. They integrate the model into large neighborhood search or local branching [189] procedures. In chapter 7 we propose a method that trains generative models to design new diving heuristics. Yet, some authors consider reinforcement learning. For example, in variable selection for branching, Sun *et al.* [202] argues that the effectiveness of strong branching is mainly due to solving the auxiliary linear programs as opposed to the selected variable. Therefore, an approach based on reinforcement learning is proposed, but the results are mixed. Other work considers reinforcement learning in cutting plane selection [207] or primal heuristics [e.g., 211]. In this thesis, we do not consider reinforcement learning approaches, but emphasize that our models could potentially benefit from as in other work that improves imitation or supervised models using reinforcement learning [210, 213, 214].

5.3.1 *Integer Programs as Graphs*

A key concern that machine learning approaches for integer programming and specifically branch and bound must address is the covariate representation of an integer program and the model choice. Of course, it would be possible to naïvely concatenate the matrices and vectors in equation (5.1) to completely describe any integer program and use any model that accepts a vector input. However, this choice is undesirable for several reasons. First, it is wasteful, because the matrices and vectors of integer programs typically

tend to be very sparse and large in size. Second, the ordering of variables and constraints in any program is arbitrary. This induces symmetries and it is likely desirable for any model to be invariant to them. Finally, it may be desirable for the model to be applicable to programs of different sizes. Even when all instances under consideration are equally sized, this is a necessity in branch and bound. There, the model may be applied across different nodes in the search tree, but the number of constraints or variables of the corresponding node program may vary as a result of cutting planes or variable fixings.

One design choice that addresses the considerations above is the representation of mixed integer linear programs as bipartite graphs and the use of graph neural networks [215]. Each variable and each constraint in equation (5.1) gives rise to a node in a bipartite graph. Each variable node is joined by an undirected edge to the nodes of the constraints in which the variable appears. Both variable nodes and constraint nodes can be associated with different features. For variables, these may include for example its objective coefficient, bounds or solver information, such as the the variable value in the current LP solution. For constraints, these may include bounds, types and sparsity or solver information, such as activity. In particular, any mixed integer program may be completely described as a bipartite graph. While previous work has adopted the bipartite graph description from Gasse *et al.* [200], we propose a new variant in chapter 6 that extends the representation to include cutting planes. We also expand the set of features for both variables and constraints throughout both chapters which improves performance. The bipartite graph can be processed by a graph neural network to make predictions on the graph or the node level. The graph neural network is naturally invariant to the arbitrary ordering of variables and constraints and can handle graphs of different sizes. Many graph neural networks exist [see e.g., 216]. Gasse *et al.* [200] proposed a model whose core are bipartite graph convolutions between the variable and constraint nodes and that is widely adopted in machine learning for branch and bound [217]. It first embeds both variable and constraint nodes using a multi-layer perceptron with a single hidden layer and layer normalization. We use 64-dimensional embeddings in all our experiments. The model then uses two bipartite graph convolutions After embedding both variable and constraint nodes, first from variables to constraints and then from constraints to variables. Finally, the model makes predictions from the convolved variable embedding using another multi-layer perceptron with a single hidden layer. We build on this model but propose modifications in

chapter 6 and chapter 7 that improve performance. This includes the use of batch normalization instead of layer normalization, changing the order of convolutions where appropriate and tailoring the output layers to various applications of interest. In chapter 6 we propose an extension of the model in Gasse *et al.* [200] that can handle cutting planes explicitly and be used for cut selection.

6

LEARNING TO CUT IN BRANCH AND BOUND

SYNOPSIS In this chapter, we propose a method for cutting plane selection in branch and bound. As described in chapter 5, cutting planes are an important component of modern solvers, because they facilitate bound improvements on the optimal solution value, i.e., they may tighten the dualbound. In practice, this helps to prune the tree and direct the search. For selecting cuts, modern solvers today rely on manually designed heuristics that are tuned to gauge the potential effectiveness of cuts. In this chapter, we show that these heuristics may be ineffective and that a greedy lookahead selection rule that incrementally adds cuts yields better bound improvements. Unfortunately, this rule is too expensive to deploy in practice. Therefore, our approach trains graph neural networks that learn to imitate the expensive lookahead rule and can be deployed for cutting plane selection inside branch and bound at lower cost. Experimentally, we demonstrate that our models select better cuts than standard heuristics and a competing approach based on reinforcement learning on a range of integer programs. Within branch and bound and, the cuts our model selects at the root speed up the remaining tree search to improve the residual time of the solver for neural network verification.

ATTRIBUTION This chapter is largely based on the following publication that was authored jointly with Giulia Zarpellon, Andreas Krause, Laurent Charlin and Chris J. Maddison.

- Paulus, M. B. *et al. Learning to Cut by Looking Ahead: Cutting Plane Selection via Imitation Learning* in *Proceedings of the 39th International Conference on Machine Learning* (2022)

6.1 CUTTING PLANES IN BRANCH AND BOUND

6.1.1 *Cutting Planes*

In integer programming, a cutting plane is a linear inequality constraint that separates the convex hull of feasible solutions from the LP solution of its relaxation. This is illustrated in Figure 6.1. Let $\chi := (\alpha, \beta)$ be a cutting

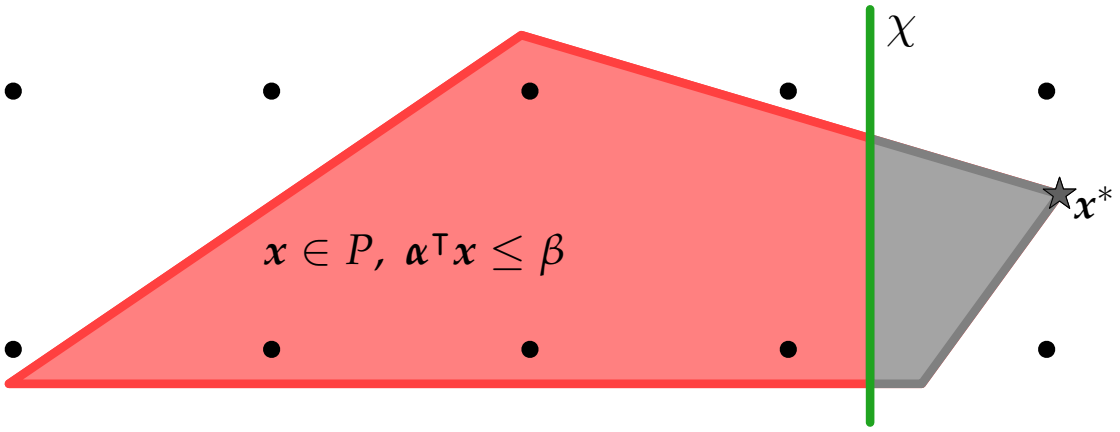


FIGURE 6.1: In integer programming, a cutting plane is a linear inequality constraint that separates the convex hull of feasible solutions from the LP solution of its relaxation. In this figure, black dots represent integer feasible solutions. The linear constraints of the integer program give rise to the feasible region in red and grey. The optimal solution of the LP relaxation is x^*. The cutting plane (in green) cuts off a part of the feasible region that includes x^* but none of the integer feasible solutions.

plane, then it holds that

$$\alpha^\intercal x \leq \beta \text{ for all } x \in \mathcal{X} \quad \text{and} \quad \alpha^\intercal x^* > \beta \tag{6.1}$$

where x^* is the current solution of the LP relaxation, for example at the root node as in (5.2).

Cutting planes are useful for branch and bound, because they facilitate bound improvements. Specifically, the LP relaxation may be tightened by adding a cutting plane. This is because the cut reduces reduces the feasible region of the linear program and invalidates its current solution. Thus, the solution to the relaxation that additionally includes the cutting plane may improve the dualbound. We write the improvement in the dualbound as a result of adding a cut to the LP relaxation at the root node with dualbound $\check{z}$ as

$$\tilde{y}_\chi = |\gamma_d(\check{z}_\chi) - \gamma_d(\check{z})| \tag{6.2}$$

where $\check{z}_\chi$ is the bound after adding the cutting plane χ to the relaxation. The bound will not worsen, i.e., $\tilde{y}_\chi \geq 0$, but bound improvements are not guaranteed.

6.1.2 *Cutting Plane Selection*

Many procedures to generate cutting planes have been developed [see e.g., 218]. Some well-known classes of cutting plane algorithms include *Complemented Mixed Integer Rounding* [219] or *Gomory* [183] cuts. Modern solvers implement numerous algorithms that are called frequently to separate a variety of cutting planes and improve the dualbound. Naturally, adding all available cutting planes $\chi \in \mathcal{C}$ to the LP relaxation would result in the tightest bound. However, this is rarely done, because each additional cut increases the size of and cost to resolve the linear program. It also makes it more likely to encounter numerical instabilities. Thus, there exists a trade-off in cutting plane selection between facilitating bounds improvements and restraining the computational cost for resolving the relaxation. In practice, it is desirable to select only a handful of cuts that yield sizable improvements in the dualbound without significantly increasing the complexity of resolving the linear program.

The open-source branch and bound solver SCIP addresses this problem with an iterative procedure. It first assigns a heuristic score to each cut $\hat{y}_\chi$. As described in chapter 5 the default score is a weighted average of a cut's efficacy, integral support and parallelism with the objective. It then selects the cut with the highest score,

$$\chi^\star \in \arg\max_{\chi \in \mathcal{C}} \hat{y}_\chi \tag{6.3}$$

and discards some of the remaining cuts from $\mathcal{C}$ based on their heuristic score and their parallelism with the selected, before selecting the next best cut of the remaining ones. This procedure is repeated until a maximum number of cuts has been selected or no cuts remain. Many heuristic scores $\hat{y}_\chi$ for cut selection have been proposed and we briefly list and describe the most common ones in Table 6.1. Wesselmann & Stuhl [220] gives a good overview. Unfortunately, these existing heuristics to score cutting planes are generic and fail to exploit problem-specific characteristics. As a result, they may perform poorly in practice and do not select the cutting planes that facilitate the best bound improvements as we show in section 6.2. In applications where similar problem instances are solved repeatedly and structural commonality exists this creates an opportunity to improve over existing methods for cut selection with machine learning. We develop such a method in the next section.

Heuristic	Description
Efficacy	$\hat{y} = \lvert \boldsymbol{\alpha}^\intercal \boldsymbol{x}^* - \beta \rvert \,/\, \lVert \boldsymbol{\alpha} \rVert$
Objective Parallelism	$\hat{y} = \boldsymbol{\alpha}^\intercal \boldsymbol{c} / \lVert \boldsymbol{c} \rVert \lVert \boldsymbol{\alpha} \rVert$
Violation	$\hat{y} = \lvert \boldsymbol{\alpha}^\intercal \boldsymbol{x}^* - \beta \rvert$
Relative Violation	$\hat{y} = \lvert \boldsymbol{\alpha}^\intercal \boldsymbol{x}^* - \beta \rvert \,/\, \lvert \beta \rvert$
Support	$\hat{y} = n - \left\lvert \left\{ j \mid \boldsymbol{\alpha}_j \neq 0 \right\} \right\rvert$
Integer Support	$\hat{y} = \left\lvert \left\{ j \mid \boldsymbol{\alpha}_j \neq 0,\, j \in \mathcal{I} \right\} \right\rvert \,/\, \left\lvert \left\{ j \mid \boldsymbol{\alpha}_j \neq 0 \right\} \right\rvert$
Expected Improvement	$\hat{y} = (\boldsymbol{\alpha}^\intercal \boldsymbol{c} / \lVert \boldsymbol{\alpha} \rVert) \,(\lvert \boldsymbol{\alpha}^\intercal \boldsymbol{x}^* - \beta \rvert \,/\, \lVert \boldsymbol{\alpha} \rVert)$
Default	$\hat{y}$ is a weighted average of *Efficacy*, *Objective Parallelism* and *Integral Support*.

TABLE 6.1: Existing standard heuristics to score a cut $\chi = (\boldsymbol{\alpha}, \beta)$ based on [220]. Ties are broken at random for all methods. In our experiments, we use native functions of SCIP 7.0.2 where available to compute each score.

6.2 LEARNING TO CUT

We propose *NeuralCut* to learn application-specific models that score cutting planes and can replace existing heuristics for cut selection. Our approach is based on the finding that a simple rule for cut selection that *looks ahead* to explicitly compute bound improvements outperforms existing heuristics. Because this rule is too expensive to be used in practice, we train graph neural networks that learn to imitate this rule. At test time, our models may be called to estimate (in parallel) the bound improvements of all available cuts $\chi \in \mathcal{C}$ and provide a score for cut selection in (6.3). *NeuralCut* is extends on the model described in chapter 5 to cut selection and features several innovations including a novel objective for cut selection and a tripartite graph to represent both integer programs and cutting planes.

6.2.1 *Cutting by Looking Ahead*

As argued previously, it is desirable in branch and bound to select only a few cutting planes from a pool of available cuts that facilitate sizable bound improvements. With this objective, it is natural to consider the following

rule: From all available cuts, select the cutting plane that facilitates the largest bound improvement,

$$\chi^\star \in \arg\max_{\chi \in \mathcal{C}} \tilde{y}_\chi \tag{6.4}$$

where ties are broken at random to select a single cut. This rule can be used repeatedly to select multiple cuts as in the selection procedure of the SCIP solver. It then gives rise to a greedy algorithm that scores cuts directly on the bound improvement they yield. Of course, this rule is prohibitively expensive. The rule must *look ahead* and compute the bound improvement for each available cut $\chi \in \mathcal{C}$, if it *was* added to the relaxation. For this reason, we call the rule *Lookahead*. Its high computational cost is likely the reason why *Lookahead* has to the best of our knowledge not been previously described for cut selection, despite the fact that effectively approximating bound improvements is a recognized goal for the heuristic design of cut scores [220, 221].

We compare the performance of *Lookahead* against existing heuristics for cut selection on 510 bounded and feasible instances from the 'easy' collection of MIPLIB 2017 [222]. After pre-solving each instance, we perform $T = 30$ separation rounds and add a single cut per round to the LP relaxation. We discard the instances for which thirty such separation rounds could not be completed within 24 hours with our available hardware. Each method selects the cutting plane it assigns the highest score to and methods only differ in the function they use to score cuts $\chi \in \mathcal{C}$. *Lookahead* uses $\tilde{y}_\chi$ for scoring cuts as described above. For any method, let $\check{z}_t$ be the dualbound after performing t separation rounds and adding a single cut in each. The initial dualbound after presolving is $\check{z}_0$ which is the same for all methods. For each method and each round, we compute the integrality gap closed (IGC) defined as

$$\mathrm{IGC}_t := 1 - \frac{\gamma_d(\check{z}_t)}{\gamma_d(\check{z}_0)} \tag{6.5}$$

In Figure 6.2, the IGC is averaged over all instances for each method at each round. *Lookahead* clearly outperforms other rules for cut selection throughout. It achieves larger IGC with fewer cuts than all baselines. The margin over the default rule of the SCIP solver is sizable; *Lookahead* achieves mean 0.25 IGC (compared to ∼0.15 IGC of the default rule) after 30 cuts and reaches 0.15 IGC after adding only five cuts on average. Further, since MIPLIB contains diverse instances that vary in size, complexity and structure, this suggest that *Lookahead* may be a universally strong criterion for cut selection, potentially useful to improve performance in general-purpose

MILP solvers. This is an interesting avenue for future work, but in this chapter we will focus on designing application-specific models for cut selection.

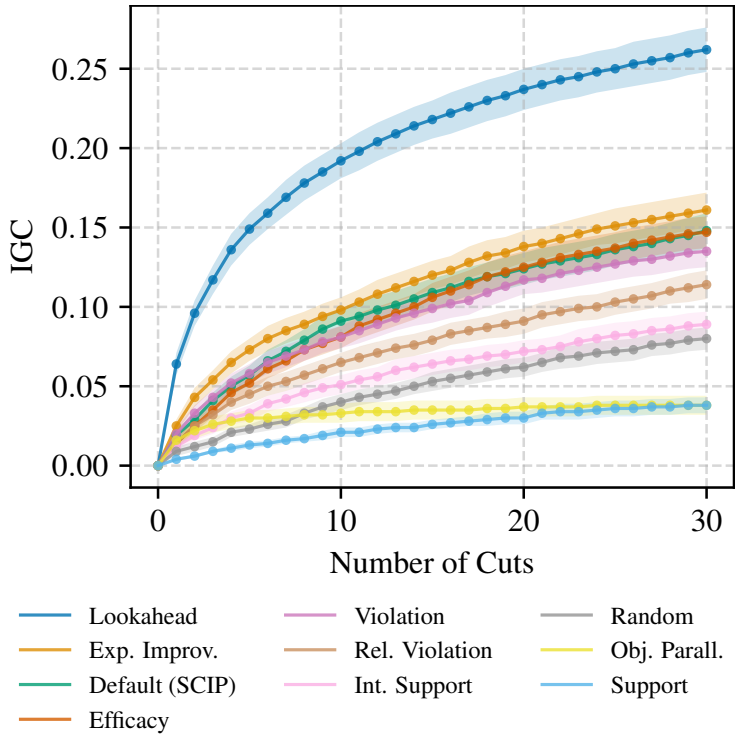


FIGURE 6.2: Lookahead clearly outperforms common heuristics for cut selection on 510 instances from the MIPLIB (easy) collection. It achieves higher mean IGC throughout when performing 30 consecutive separation rounds and adding a single cut per round.

6.2.2 *Learning from Looking Ahead*

Lookahead is a strong criterion for cutting plane selection, but it is too expensive to be deployed in a branch and bound solver. In order to select a single cut, *Lookahead* computes the bound improvement for each cut $\chi \in \mathcal{C}$. Therefore, our strategy is to learn a model that approximates the the bound improvement $\tilde{y}_\chi$ that *Lookahead* uses and can be used for cut selection instead. To learn such a model, our objective is to minimize the following loss function

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) := -\frac{1}{|\mathcal{C}|} \sum_{\chi \in \mathcal{C}} y_\chi \log \hat{y}_\chi + (1 - y_\chi) \log(1 - \hat{y}_\chi) \tag{6.6}$$

where $y_\chi = \tilde{y}_\chi / \tilde{y}_{\chi^\star}$ is the bound fulfillment of the cut $\chi \in \mathcal{C}$ and $\chi^\star$ is the cut that *Lookahead* selects as in (6.4). Our model predicts $\hat{\boldsymbol{y}}$ to approximate the bound fulfillment for each cut in the pool conditional on the integer program, the cut pool and the current solver state, but we have suppressed the dependence of $\hat{\boldsymbol{y}}$ on $\boldsymbol{x}_{\text{sup}}$ and $\boldsymbol{\theta}_{\textbf{sup}}$ above for notational convenience. Notably, we target bound fulfillment $\boldsymbol{y}$ as opposed to bound improvement $\tilde{\boldsymbol{y}}$, the dualbound $\check{\boldsymbol{z}}$ or $(\mathbb{P}(\chi = \chi^\star))_{\chi \in \mathcal{C}}$ for each cut and there are several advantages of this choice. First, bound fulfillment is invariant to the scale of the objective and the absolute improvement a cut facilitates. This prevents the loss from being dominated by only few training examples that admit large bound improvements and instead uniformly weights all training examples. Second, the bound fulfillment of a cut is *comparative*; it depends on the bound improvement that the best available cut in the cutpool facilitates. Thus, to achieve low loss, the model only needs to learn how well a cut performs relative to the other available cuts but is freed from learning how well it performs overall. This is arguably easier and aligns well with the goal of selecting the best available cut. Third, our model makes prediction for each available cut and jointly learns from *all* bound fulfillments. This provides a richer signal than for example predicting which cut *Lookahead* selects and gracefully handles cases where several cuts achieve the same best improvement. At test time, *NeuralCut* selects the cut for which it predicts the highest bound fulfillment

$$\chi^\star \in \arg\max_{\chi \in \mathcal{C}} \hat{y}_\chi(\boldsymbol{x}_{\text{sup}}, \boldsymbol{\theta}_{\textbf{sup}}) \tag{6.7}$$

TRIPARTITE GRAPH Our model makes prediction from the linear program relaxation of a node in the branch and bound tree and the pool of available cuts. For this purpose, we propose a variant of the bipartite graph representation that was discussed in section 5.3.1. Our model extends the bipartite to a tripartite graph where the available cuts form a third group of nodes (Figure 6.3). Each cut node is joined by an undirected edge to the nodes of the variables that appear in the respective cut. This is analogous to how variables and constraints are joined in the bipartite graph described in chapter 5. In addition, each cut node is connected to each constraint node with a weight that measures the degree of parallelism between the corresponding cut and constraint. The intuition for this is if a cut and a constraint are exactly parallel to each other than either the constraint or the cut may be redundant. Additionally, in a graph neural network connecting constraints and cuts will facilitate skip connections between constraints and cut. Intuitively, the model is not forced to communicate

information between cuts and constraints only via the variable nodes, but can communicate information directly. We also expand and adapt the set of features for the nodes of the graph. In particular, we include more structural information about variables, constraint and cuts, use binary variables to encode information. We give a complete list of the features we use in Appendix B.6.2.

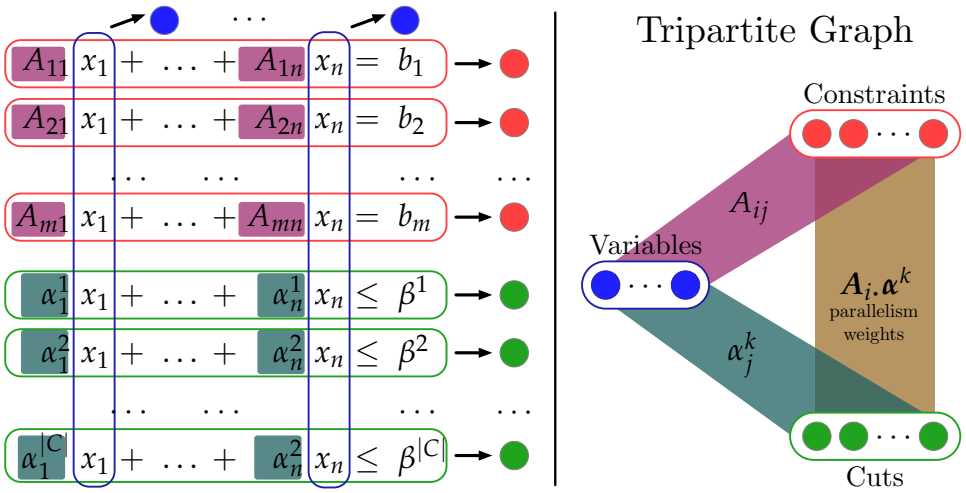


FIGURE 6.3: We encode the linear program relaxation of a node in the search tree and the pool of available cuts using a tripartite graph. Each constraint, variable and candidate cut gives rise to a node in this graph. Constraints and cuts are joined by undirected edges to their variables with non-zero coefficients. In addition, cuts and constraints are joined by edges that are weighted by their degree of parallelism.

MODEL Our model predicts the bound fulfillment for each cut in the pool. For this purpose, we adapt the model from Gasse *et al.* [200] discussed in chapter 5 to cutting plane selection (Figure 6.4). First, our model embeds the features of all nodes. Then, our model uses convolutions between variables and constraints, cuts and variables, and cuts and constraints to update the embedding and pass information about the program and the available cuts. Finally, our model uses an attention between the cuts in the pool before predicting the bound fulfillment for each cut from the embedding of its node. The use of the attention layer between the cuts is motivated by the need of the model to predict the bound fulfillment of each cut which is a *comparative* quantity as discussed earlier.

Our model differs from the model in chapter 5 in several ways, including the use of a tripartite graph to explicitly represent the pool of available cuts, the choice of features for variable, constraint and cut nodes, the use of

batch normalization [223] instead of layer normalization [224] throughout and the use of an attention layer. We validate our design choices in the experimental section where we show that they improve performance.

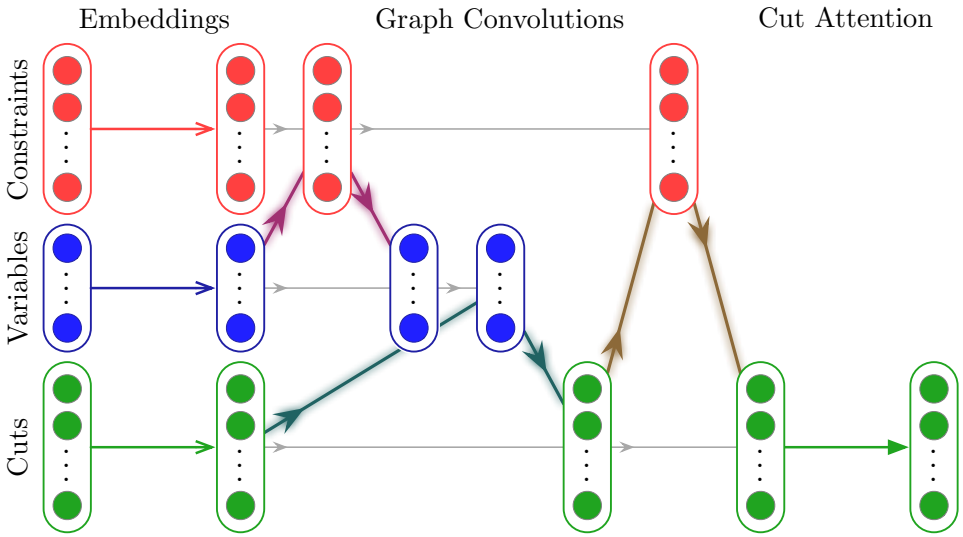


FIGURE 6.4: Our model for *NeuralCut* is a graph neural network. First, it embeds all nodes in the tripartite graph from Figure 6.3. Then, our model uses convolutions between variables and constraints, cuts and variables, and cuts and constraints to update the embeddings and pass information about the program and the available cuts. Finally, our model uses an attention layer between the cuts and the pool before predicting the bound fulfillment for each cut from the embedding of its node.

6.2.3 *Deployment*

We conduct our experiments with the open-source solver SCIP 7.0.2 [193]. Unfortunately, this solver does not directly expose control over separation algorithms and cutting plane selection to the user. This makes it difficult to study cutting plane selection. To compare *NeuralCut* against various heuristics for cutting plane selection in section 6.4.1, we extend the solver's source in C and directly interface the most important separators of SCIP in Python. This includes *Clique*, *Complemented Mixed Integer Rounding*, *Disjunctive*, *Flow Cover*, *Gomory*, *Implied Bounds*, *Multi-Commodity Flow Network*, *Odd-Cycle*, *Strong Chvátal-Gomory* and *Zero-half* separators. This allows us to enforce separation rounds with any subset of separators at our own discretion. Cuts are accumulated into a separate pool, where they can be inspected and applied to the LP relaxation. To test *NeuralCut* for branch

and bound, we integrate it into the open-source solver SCIP 7.0.2 [193]. This solver exposes a plug-in for separation algorithms that we abuse to control cutting plane selection in branch and bound. Our approach is inspired by `https://github.com/avrech/learning2cut`. Via the plug-in, we directly enforce the cut selection our trained models predict and give more details in section 6.4.2. Finally, we note that for future research on cutting plane selection, it may be more convenient to use the cut selection plug-in that was made available with SCIP 8.0.0 [225] after this work was completed.

6.3 RELATED WORK

NeuralCut relies on training a model to imitate a strong but expensive expert, in our case the *Lookahead* rule. This is a common approach in machine learning for branch and bound. For example, in variable selection for branching, several works [200] learn to imitate the *Strong Branching* heuristic. Interestingly, this heuristic performs a lookahead step to compute the bound improvements that result from branching on a particular variable. In contrast, *Lookahead* performs a lookahead step to compute the bound improvements that result from selecting a particular cutting plane. In analogy to *Strong Branching*, our *Lookahead* rule could be dubbed *Strong Cutting*.

Some other works considers machine learning for cutting plane selection. Tang, Agrawal & Faenza [207] propose an approach to select *Gomory* cutting planes. This approach casts cut selection as a reinforcement learning problem and uses bound improvements to reward a model that scores cuts and is trained with evolutionary strategies. Turner *et al.* [209] also adopt reinforcement learning, but consider specialized integer programs and cutting planes. Their model that adapts the weights of existing heuristic scores. Huang *et al.* [226] cast cut selection as multiple instance learning and apply it to a proprietary solver, but they only perform a single separation round. In contrast, our approach is based on imitation learning. It is a generic recipe to learn cut selection tailored to any specific application and can be called in any separation round. Berthold, Francobaldi & Hendel [208] consider cutting plane management. They train a regression forest to predict improvements in solving time from separating cutting planes at nodes other than the root.

6.4 EXPERIMENTS

We evaluated the effectiveness of *NeuralCut* in two different experiments and on a total of five datasets. The first set of experiments (Section 6.4.1) was designed to study the cut selection performance of *NeuralCut* in isolation and compare it against existing heuristics for selecting cutting planes. On a benchmark of four classes of synthetic integer programs from previous work [207], we performed thirty consecutive separation rounds and added a single cut per round. We measured both the bound fulfillment of selected cuts and integrated the normalized dual gap over the thirty rounds. We found that *NeuralCut* outperformed existing heuristics for cut selection and a competing RL approach [207] and approximated the performance of *Lookahead* closely. We also performed ablations to validate the design choices we made for *NeuralCut* and found they improve performance Finally, we tested the transferability of the models we trained and found that *NeuralCut* mostly learns to exploit application-specific patterns for cut selection. The second set of experiments (Section 6.4.2) directly included *NeuralCut* into the branch and bound process of the open source solver SCIP. The solver used *NeuralCut* to select cuts at the root node of the tree. Our goal was to demonstrate that *NeuralCut* selects better cutting planes than the default procedure of the solver and that its selection facilitates improvement in branch and bound search downstream. We considered instances from neural network verification [197] and measured the *residual* solving time after resolving the root node. We found that *NeuralCut* could achieve sizable improvements in the dualbound by selecting only a small number of cuts relative to the default solver and observed that this selection resulted in improvements of residual solving time. The results highlight the potential of machine learning methods to improve cutting plane selection in branch and bound solvers.

We collected data for training and validation: In all experiments, we extracted the tripartite graph input representation described in the previous section and representations of the available cuts $\mathcal{C}$ with the features listed in Appendix B.6.2. We also compute the bound improvement for each cut $\chi \in \mathcal{C}$ without any spill-over effects on the state of the solver with the aid of the built-in diving or probing mode. For each instance, we collect data from several separation rounds and give more details below. We trained separate models for each dataset. We trained each model with ADAM [227] for 100 epochs in the first set of experiments and for 32 epochs in the second set of experiments. For each dataset, we chose the batch size to exhaust the

memory of a single NVIDIA GeForce GTX 1080 Ti device. We validated after every epoch and chose the model that achieved the best validation loss. We did not use any accelerators at test time.

6.4.1 *Cutting with* NeuralCut

With this first set of experiments, we studied the cutting plane selection performance of *NeuralCut* and compared it against existing heuristics for cut selection. We used the same benchmark as previous work [207]. This benchmark consists of four different classes of integer programs, including *Packing*, *Binary Packing* and *Planning* problems as well as finding the maximum cut in a graph (*Max Cut*). For each class, Tang, Agrawal & Faenza [207] generates problems in three different sizes, *snall*, *medium* and *large*. We find that the *snall-* and *medium*-sized problems are often very easy and may be entirely presolved. Therefore, we only consider problems of the *large* size, but note that these are still relatively small. For each class, we trained on 1000 instances and validated and tested on 500 instances. We compare *NeuralCut* against existing standard cut selection heuristics listed in Table 6.1. For comparison, we also report results for *Lookahead* and *Random* that selects a cut from the pool randomly. Finally, we considered a competing approach based on reinforcement learning [207]. This approach cast cut selection as a reinforcement learning problem and uses bound improvements to reward a model for cut selection trained with evolutionary strategies. Tang, Agrawal & Faenza [207] consider only *Gomory* cuts and use the Gurobi solver [228]. To compare, we reimplemented their method and expose it to all available cutting planes. We presolved all instances, but did not branch and disabled primal heuristics and the the default separation procedure of the solver, because we enforce separation rounds manually. We resolve the LP relaxation which gives the initial dualbound $\check{z}_0$ and the initial dual gap $\gamma_d(\check{z}_0)$. At test time, we perform up to thirty discretionary separation rounds where we use all available separators as described in section 6.2.3. In each round $1 \leq t \leq T = 30$, we select a single cut with the highest score and break ties at random for each selection rule. We then measure the dualbound $\check{z}_t$ and compute the normalized the dual gap against the initial dual gap $\gamma_d(\check{z}_0)$ to compute the normalized dual integral

$$\Gamma_d := \sum_{t=1}^{T} \frac{\gamma_d(\check{z}_t)}{\gamma_d(\check{z}_0)} \tag{6.8}$$

To collect data for training *NeuralCut*, we only perform ten separation rounds. We collect the tripartite graph representation of the node program and the bound improvement for each available cut. We then select with equal probability either a cut with the highest bound fulfillment (*Lookahead*) or a cut with the highest default solver score (*Default*) or a random cut (*Random*) and trigger the next separation round. This exploration rule is designed to increase the diversity of the training samples. We collect this data also for the test instances and report the average bound fulfillment each cut selects. This is useful, because it facilitates comparison between all selection rules on the same cut pools and LP relaxations. This is not the case for the normalized dual integral, where comparison is confounded if any two heuristics selected different cutting planes in previous rounds.

NeuralCut clearly outperforms existing heuristics for cut selection on three out of four datasets (*Maximum Cut*, *Binary Packing* and *Planning*). It achieves a lower dual integral on the test instances over the thirty separation rounds (Table 6.3, right panel with visuals in Appendix C.3.3) indicating that it selects better cuts that reduce the dual gap further and earlier. Its performance nearly matches the performance of *Lookahead* which validates our approach. Overall, the results suggest that learning to imitate *Lookahead* is viable. The cuts *NeuralCut* selects on the test set achieve much higher bound fulfillment than for the other heuristics (Table 6.3, left). On *Planning* and *Maximum Cut*, its bound fulfillment is (nearly) optimal. On the *Packing* dataset, *NeuralCut* statistically achieves the same performance as several other heuristics (*Default*, *Efficacy*, *Rel. Violation*). The performance is comparable to *Lookahead*, but the improvement over *Random* is relatively small. This could suggest that the *Packing* instances are not very discriminatory for cut selection, perhaps because the separated cutting planes are homogenous. *NeuralCut* also clearly outperforms the competing RL approach [207]. The models we trained outperform the *Random* heuristic as in the original publication, but failed to outperform more sophisticated heuristics for cut selection against which Tang, Agrawal & Faenza [207] did not compare. Overall, we found it difficult to train their model, but believe our approach based on imitation rather than reinforcement learning may be a more promising approach to design models for cut selection.

MODEL ABLATION We validate the design choices we made in the previous section and test several variants of *NeuralCut* on the *Binary Packing* dataset (Table 6.4). Specifically, we begin with the basic model from Gasse *et al.* [200] that was described in chapter 5 minimally adapted for cut se-

	Bound fulfillment (↑), mean				**Normalized dual integral (↓)**, mean (ste)			
	Max. Cut	Packing	Bin. Packing	Planning	Max. Cut	Packing	Bin. Packing	Planning
Lookahead	*1.0*	*1.0*	*1.0*	*1.0*	*15.1 (0.1)*	*26.4 (0.1)*	*9.9 (0.3)*	*10.3 (0.1)*
NeuralCut	**0.96**	**0.61**	**0.78**	***1.0***	**15.6 (0.1)**	***26.3 (0.1)***	**11.0 (0.3)**	**10.4 (0.1)**
Tang et al. [207]	0.58	0.27	0.22	0.49	19.0 (0.1)	27.6 (0.1)	16.1 (0.4)	15.0 (0.1)
Default (SCIP)	0.71	0.60	0.33	0.64	16.7 (0.1)	***26.3 (0.1)***	15.4 (0.3)	14.0 (0.1)
Exp. Improv.	0.69	0.60	0.32	0.85	19.0 (0.1)	26.3 (0.1)	15.1 (0.3)	11.2 (0.1)
Efficacy	0.65	0.60	0.32	0.46	17.0 (0.1)	***26.3 (0.1)***	15.2 (0.3)	14.5 (0.1)
Obj. Parall.	0.47	0.34	0.27	0.44	24.0 (0.1)	28.3 (0.1)	22.2 (0.3)	27.7 (0.1)
Rel. Violation	0.50	0.60	0.33	0.48	18.0 (0.1)	***26.3 (0.1)***	14.9 (0.3)	15.4 (0.1)
Violation	0.64	0.35	0.21	0.26	23.2 (0.1)	28.8 (0.1)	19.4 (0.3)	25.8 (0.1)
Support	0.57	0.18	0.13	0.29	19.3 (0.1)	28.8 (0.1)	24.8 (0.2)	18.4 (0.1)
Int. Support	0.62	0.18	0.21	0.34	21.9 (0.1)	28.8 (0.1)	21.1 (0.3)	23.3 (0.1)
Random	0.41	0.15	0.16	0.25	22.0 (0.1)	28.7 (0.1)	21.2 (0.3)	23.3 (0.1)

TABLE 6.3: NeuralCut selects cuts with high average bound fulfillment (left) and smaller average reversed IGC integral on four benchmarks. It outperforms both a competing RL approach [207] and manual heuristics for cut selection. It approximates the performance of *Lookahead* closely. Performance on test instances for 30 consecutive separation rounds and adding a single cut per round. Best model or heuristic is bold-faced, best overall is in italics.

lection. This basic variant uses cutting plane nodes instead of constraint nodes and switches the order in which half-convolutions are applied. After training, this model achieves an average test bound fulfillment of 0.54 and a dual integral of 16.09. It outperforms the simplest heuristics for cutting plane selection, but falls short of more sophisticated heuristics, e.g., *Expected Improvement* or *Efficacy*. We then improve this model by using the features of NeuralCut (+ our features) for both cuts and variables instead of the original features from Gasse *et al.* [200]. This improves bound fulfillment to 0.66 and the dual integral to 12.77. This model already outperforms all existing heuristics for cut selection, but falls short of the final performance of *NeuralCut*. Performance is further improved by making architectural changes (+ our architecture). Specifically, we replace layer normalization with batch normalization and add the attention layer for the cut pool to the model. Finally, we use the tripartite graph representation and model variables, constraints and cut nodes jointly which gives the full *NeuralCut* model. We observe that our architectural choices boost performance significantly, bound fulfillment increased from 0.76 to 0.78 and the dual integral is reduced from 12.77 to 10.96. Including constraint nodes as in *NeuralCut* improved bound fulfillment, but did not significantly change the dual integral. We found this surprising, but it is plausible that bound fulfillment and the dual integral are not perfectly correlated, the dual integral is more noisy and a model that does not explicitly model constraint nodes can still reasonably learn, because the variable features may implicitly carry some information about the constraints via their features, e.g., their value in the current LP solution. The ablation on *Binary Packing* was most discriminative, but ablations for the other benchmarks are in Appendix C.3.1. In addition, an ablation on the choice of the loss function is in Appendix C.3.2.

	Bound fullfilment	**Normalized dual integral**
Gasse *et al.* [200]	0.54	16.09 (0.31)
+ our features	0.66	12.77 (0.32)
+ our architecture	0.76	**10.73 (0.33)**
NeuralCut (+ our graph)	**0.78**	**10.96 (0.33)**

TABLE 6.4: Our model choices tend to improve tend to improve both bound fulfillment on test samples and reversed IGC integral on test instances for binary packing. Ablations for the other benchmarks are in Table C.6 in Appendix C.

TRANSFERABILITY Finally, we test the transferability of *NeuralCut* models across different domains of integer programs (Table 6.5). For this purpose, each of the four *NeuralCut* models in Table 6.3 is additionally evaluated on the test instances of the other three domains. For example, the entry in the second row (*Packing*) and fourth column (*Planning*) of Table 6.5 corresponds to the *NeuralCut* model trained on *Packing* instances, but tested on *Planning* instances. We find that *NeuralCut* models do not transfer well to other domains: On all benchmarks, the test performance of transferred models is significantly worse than the performance of the model that was trained on instances of the respective domain. This suggests that *NeuralCut* learns to exploit application-specific patterns to select good cuts.

	Normalized dual integral (↓), mean (ste)			
	Max. Cut	Packing	Bin. Packing	Planning
Max. Cut	**15.55 (0.09)**	28.96 (0.03)	26.33 (0.16)	17.07 (0.07)
Packing	25.03 (0.06)	**26.30 (0.08)**	16.29 (0.27)	29.67 (0.01)
Bin. Packing	21.75 (0.08)	27.66 (0.06)	**10.96 (0.33)**	25.01 (0.12)
Planning	19.13 (0.08)	28.98 (0.03)	22.83 (0.27)	**10.42 (0.04)**

TABLE 6.5: Training NeuralCut on a single domain (row) does not generalize well to other domains (columns). On all four benchmarks, the model that was trained on the same domain as it is tested on (i.e., main diagonal) performs best. Models that were trained on another domain when they are tested on perform significantly worse. The performance drop tends to be less severe when domains are more similar, e.g., packing and binary packing.

6.4.2 NeuralCut *in branch and bound*

With this second set of experiments, our goal was to use *NeuralCut* within branch and bound. The goal of this experiment was to demonstrate that improved cutting plane selection via *NeuralCut* can improve the performance of branch and bound *downstream*. To this end, we included *NeuralCut* into the open-source solver SCIP via a plug-in as described in section 6.2.3 to select cutting planes at the root node. It is common to select multiple cuts from a cut pool. As previously described, this is accomplished in SCIP

with an iterative selection procedure. This selection procedure involves various stopping criteria to terminate the selection and prevent the solver from adding additional cutting planes in any separation round. The criteria have been optimized for the solver's default configuration and its heuristic cut score and this may confound comparison when the score *NeuralCut* predicts is simply used instead. To select multiple cuts with *NeuralCut*, we therefore equip *NeuralCut* with a simple stopping rule parameterized by a threshold parameter $\epsilon \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1\}$ to detect stalling: The selection procedure is terminated after the bound improvement from adding cuts has not exceeded ϵ for 10 consecutive rounds. We report results for all ϵ we tried. We do not use the additional stopping criteria that are part of SCIP, but this may improve performance. The loop to select multiple cutting planes with *NeuralCut* in a single separation round (i.e., from a given cut pool) is implemented entirely in Python and the final selection (of multiple cuts) is then enforced via plug-in. This is not performant, but sufficient for measuring downstream performance improvements resulting from improved cut selection which is the goal of this experiment. For training, for each instance we collect the tripartite graph representation and bound improvements for up to (a total of) fifty iterations across all separation rounds at the root node. We select these fifty samples uniformly at random for each instance and use reservoir sampling [229] to accomplish this, since the number of separation rounds and iterations is unknown a priori. We use the *Default* rule of the solver for data collection.

As an application, we consider verifying the robustness of neural networks. These instances are particularly interesting for cut selection, because the relaxation is notoriously weak and can be significantly improved with cutting planes. We use the instances from Nair *et al.* [197], but we discard instances that are trivial and infeasible or numerically unstable. We also exclude the few instances that could not be solved within a time limit of one hour with the default SCIP solver to give a total of 2384 instances for training, 519 instances for validation and 545 instances for testing.

We measure *residual* solving time to assess *downstream* branch and bound performance. This is the time the branch and bound solver requires after resolving the root node until an instance is completely solved, i.e. the optimal solution is provably found. We leave all other solver parameters unchanged and compare *NeuralCut* with various ϵ against the default solver to isolate the effect the of improved cutting plane selection on branch and bound. Our experiments were conducted on a shared distributed compute cluster. To reduce measurement variability, we ran all test time

	Cut selection at root		Residual solve		
	Cuts	Norm. dualbound	Time	Nodes	LP iters.
SCIP	279	**1.00**	23.65	745	21933
NeuralCut					
$\epsilon = 10^{-5}$	105	**1.00**	22.35	**671**	18846
$\epsilon = 10^{-4}$	81	0.99	**20.89**	756	19310
$\epsilon = 10^{-3}$	48	0.98	22.73	803	**18048**
$\epsilon = 10^{-2}$	27	0.94	24.06	861	**18048**
$\epsilon = 10^{-1}$	11	0.76	25.09	998	21996
$\epsilon = 1$	**10**	0.53	23.35	952	21611

TABLE 6.6: *NeuralCut* in a B&C solver – Median metrics for NN verification test instances and different values for the threshold parameter ϵ. NeuralCut improves the dualbound at the root at a fraction of the number of cuts the default solver SCIP selects. When subsequently branching from the root node, this can reduce the remaining solving time by lowering the number of LP iterations or expanded nodes in the search tree. Best are bold-faced.

evaluations repeatedly on machines equipped with the same Intel Intel Xeon Gold 51118 CPU 2.3 GHz processor for three different seedings of the solver. We batched evaluations randomly across instance and methods to be processed sequentially on the same machine. We report test set means and standard errors over the three different random seeds. NEED TO CLEAN UP RESULTS (MEDIAN ETC.) In addition to residual solving time, we report the number of cuts selected at the root node, the bound improvement at the root relative to the default solver, the mean number of nodes of the search tree and the residual number of LP iterations.

NeuralCut improves residual solving time against the default solver from an average by nearly 10% (Table 6.6). It selects more effective cutting planes at the root node, where it achieves similar bound improvements by only adding a fraction of the cuts the default solver selects. For example, the best performing variant ($\epsilon = 10^{-}4$) selects only 30% as many cuts as the default solver, but achieves nearly the same bound improvements. The most aggressive variant selects only 10 cutting planes on average compared to 279 cutting planes, but still achieves more than half of the

bound improvement. Selecting fewer cutting planes without compromising the dualbound facilitates improvements in residual solving time. We find that these improvements are likely because the number of LP iterations in the branch and bound search is reduced, but the search tree does not grow significantly in size, because the node dualbounds are still tight.

However, we stress that residual solving time does not account for the in-service overhead of *NeuralCut*, because it excludes the time required to resolve the root node where the model is deployed. While calling *NeuralCut* to predict bound fulfillment is much cheaper than computing bound fulfillment exactly as *Lookahead* does, it is still expensive to call the model once for every cut that is selected. Our experimental results are encouraging, because they highlight that improved cutting plane selection via machine learning can indeed facilitate improvements in branch and bound search. To improve overall solver performance however, we believe that future work must holistically address cutting plane management instead of cutting plane selection. We discuss various opportunities for future work in the next section.

6.5 DISCUSSION

In this chapter we presented *NeuralCut* to learn application-specific cutting plane selection in branch and bound. We considered a lookahead rule that greedily selects cuts that yield the best bound improvement. This rule is effective, but too expensive to be deployed in practice. In response, we proposed *NeuralCut* and presented a method for learning to imitate the *Lookahead* rule. Our model is trained to predict the bound fulfillment of any cut under consideration with a novel objective for cut selection. It is based on a tripartite graph representation of integer programs and cut pools that we developed as well as a graph neural network that includes several innovations over previous work. We tested *NeuralCut* on a range of integer programs a real-world application from neural network verification. *NeuralCut* closely approximated the performance of *Lookahead* at a fraction of its costs and clearly outperformed existing heuristics for cut selection and a competing reinforcement learning approach. In branch and bound, *NeuralCut* facilitated improvements in *downstream* solving time and clearly highlights the potential for improving solver performance via learnt models for improved cut selection.

There are exciting avenues for future work to extend the methods we presented in this chapter. For example, our findings suggest that in order to

improve overall solver performance it is likely desirable to select multiple cuts with a single model call. To this end, a model could learn to predict the cut selection after applying *Lookahead* repeatedly to the same cut pool and pairing it with a stopping criterion. This is in analogy to how we deployed *NeuralCut* in section 6.4.2. Alternatively, a strong selection of multiple cuts may be derived by considering the following minimax optimization problem,

$$\max_{\boldsymbol{\xi} \in \{0,1\}^{|\mathcal{C}|}} \min_{\boldsymbol{x} \in P'} \boldsymbol{c}^\intercal \boldsymbol{x} - \varepsilon \sum_{\chi \in \mathcal{C}} \xi_\chi \tag{6.9}$$

where $P' = P \cap \{\boldsymbol{\alpha}_\chi^\intercal \boldsymbol{x} \leq \beta_\chi + M_\chi(1 - \xi_\chi) \, \forall \chi \in \mathcal{C}\}$. The decision variables $\boldsymbol{\xi} \in \{0,1\}^{|\mathcal{C}|}$ are used to select cutting planes. If $\xi_\chi = 1$, the corresponding cutting plane is active and $\boldsymbol{x}^* \in \arg\min_{\boldsymbol{x} \in P'} \boldsymbol{c}^\intercal \boldsymbol{x}$ will be such that $\boldsymbol{\alpha}_\chi^\intercal \boldsymbol{x} \leq \beta_\chi$. If $\xi_\chi = 0$, the corresponding cutting plane is active. This is ensured by choosing a large constant M_χ which is guaranteed to render the cutting plane inactive, because $\boldsymbol{\alpha}_\chi^\intercal \boldsymbol{x} \leq \beta_\chi + M_\chi$ is guaranteed to hold for the optimal LP solution. The hyperparameter $\varepsilon > 0$ is a penalty that is incurred for every cutting plane that is activated. For the optimal selection $\boldsymbol{\xi}^\star \in \arg\max_{\boldsymbol{\xi} \in \{0,1\}^{|\mathcal{C}|}} \min_{\boldsymbol{x} \in P'} \boldsymbol{c}^\intercal \boldsymbol{x} - \varepsilon \sum_{\chi \in \mathcal{C}} \xi_\chi$, this ensures that the bound improvement from the selected cuts is at least $\varepsilon \sum_{\chi \in \mathcal{C}} \xi_\chi^\star$. Larger ε will tend to select fewer cutting planes, while smaller ε will tend to select more cuts. Solving the minimax problem above may be more expensive than repeatedly using *Lookahead* on the same cutpool $\mathcal{C}$ to select multiple cuts, but could result in improved selection.

Overall, any model for multi-cut selection will have a significantly lower in-service overhead than *NeuralCut* and it is plausible that this may facilitate improvements in overall solver performance. In the next chapter, we will pursue such overall performance improvements, albeit from a different angle, namely by considering heuristics methods to improve the primalbound.

7

LEARNING TO DIVE IN BRANCH AND BOUND

SYNOPSIS In this chapter, we propose a primal heuristic method to find feasible solutions for mixed integer linear programs. As with cutting planes, feasible solutions provide bounds on the optimal solution value that facilitate branch and bound. But while cutting planes may tighten the dualbound, feasible solutions strengthen the primalbound (chapter 5). Our method is based on diving heuristics that are a prominent group of primal heuristics in branch and bound. Diving heuristics iteratively modify and resolve linear programs to conduct a depth-first search from any node in the search tree. Existing divers rely on generic decision rules that fail to exploit structural commonality between similar problem instances that often arise in practice. Therefore, we propose *L2Dive* to learn application-specific diving heuristics with graph neural networks: We train generative models to predict variable assignments and leverage the duality of linear programs to make diving decisions based on the model's predictions. Experimentally, we demonstrate that *L2Dive* outperforms standard divers to find better feasible solutions on a range of combinatorial optimization problems. Within branch and bound, *L2Dive* improves overall solver performance. For real-world applications from server load balancing and neural network verification, *L2Dive* improves the primal-dual integral by up to 7% (35%) on average over a tuned (default) solver baseline and reduces average solving time by 20% (29%).

ATTRIBUTION This chapter is largely based on the following preprint that was authored jointly with Andreas Krause.

- Paulus, M. B. & Krause, A. *Learning To Dive In Branch and Bound* in *Under submission.* (2023)

7.1 DIVING HEURISTICS IN BRANCH AND BOUND

7.1.1 *Generic Diving*

Diving heuristics are a prominent group of primal heuristics (chapter 5). They are based on the linear program relaxation that was given in (5.2) for

the root node and attempt to drive the LP solution $\boldsymbol{x}^*$ towards integrality. Diving heuristics conduct a depth-first search in the branch and bound tree to explore a single root-leaf path. They iteratively modify and solve linear programs to find feasible solutions. Algorithm 3 illustrates a generic diving heuristic. A dive can be initiated from any node in the branch and bound tree and will return a (possibly empty) set of feasible solutions $\mathcal{X}_{\text{div}}$ for the original mixed integer linear program in (5.1). Typically, diving heuristics alternate between tightening the bound of a single candidate variable x_j with $j \in \mathcal{D} \subseteq \mathcal{I}$ (line 5) and resolving the modified linear program (line 7), possibly after propagating domain changes [230]. The resultant solution may be integral ($x_j \in \mathbb{Z},\ \forall j \in \mathcal{I}$) or admit an integral solution via rounding [line 9, 230]. Eventually, the procedure is guaranteed to result in at least one primal-feasible solution (line 10) or an infeasible program (line 6). However, in practice solvers may prematurely abort a dive to curb its computational costs, for example by imposing a maximal diving depth (line 2), an iteration limit on the LP solver or a bound on the objective value. Diving heuristics may also leverage sophisticated rounding methods in each iteration which makes them more likely to be successful.

Diving heuristics feature prominently in integer programming. In contrast to methods for large neighborhood search, they only require repeatedly solving linear programs (in complexity class P) instead of small integer programs (NP-hard). As a result, they tend to be considerably faster and more applicable in practice. Diving heuristics are an integral part of modern branch and bound solvers.

7.1.2 *Diving heuristics*

Various diving heuristics have been proposed and Berthold [184] gives a good overview. We list and briefly describe the most common ones in Table 7.1[1]. Most diving heuristics only differ in how they select variables for bound tightening in line 3:

$$j^\star = \arg\max_{j \in \mathcal{D}} \hat{y}_j \tag{7.1}$$

where $\hat{\boldsymbol{y}}$ heuristically scores variables for diving with examples in Table 7.1. The set $\mathcal{D}$ is the set of *divable* variables that typically excludes integer

1 SCIP's diving ensemble includes a few additional divers which we did not directly compare against in Section 7.4.1, because they either choose from the other heuristics (*adaptive*), were ineffective (*conflict*), require a feasible solution (*guided*) or do not use the generic diving algorithm (*objective pseudocost*). However, these divers are active for the baselines in Section 7.4.2.

Algorithm 3: Generic Diving Heuristic

Input: An integer program as in (5.1) and a LP relaxation as in (5.2), maximal depth $d_{\max}$

Output: $\mathcal{X}_{\text{div}} \subseteq \mathcal{X}$, a (possibly empty) set of feasible solutions

Require: $\hat{y}$, a heuristic score to select variables for bound tightening

$d = 1$
while $d \leq d_{\max}$ **do**
  $j^\star = \arg\max_{j \in \mathcal{D}} \hat{y}_j$
  Either $\underline{\pi}_{j^\star} \leftarrow \lceil x^*_{j^\star} \rceil$ or $\overline{\pi}_{j^\star} \leftarrow \lfloor x^*_{j^\star} \rfloor$
  $P \leftarrow P \cap \{\underline{\pi}_{j^\star} \leq x_{j^\star} \leq \overline{\pi}_{j^\star}\}$
  if *P is infeasible* **then** `break`;
  $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x} \in P} \boldsymbol{c}^\intercal \boldsymbol{x}$
  if *$\boldsymbol{x}^*$ is roundable* **then**
    $\boldsymbol{x} = \texttt{round}(\boldsymbol{x}^*)$
    $\mathcal{X}_{\text{div}} \leftarrow \mathcal{X}_{\text{div}} \cup \{\boldsymbol{x}\}$
  end
  $d \leftarrow d + 1$
  Possibly update candidates $\mathcal{D}$
end

variables that are *slack variables* [231, Chapter 1.1] or that have already been fixed ($\underline{\pi}_j = \overline{\pi}_j$). The heuristic scores standard divers use are generic and fail to exploit problem-specific characteristics. For example, *Fractional* diving simply selects the variable with the lowest fractionality in the LP solution, i.e., $\hat{y}_j = -|x_j^* - \lfloor x_j^* + 0.5 \rfloor|$. While standard divers generally applicable to any mixed integer program, they are not universally effective. For example, we found *Farkas* diving to deliver good results on a class of facility location problems, but to be ineffective for instances of maximum independent set (Section 7.4.1). This non-specificity is particularly severe in applications, where similar problem instances are solved repeatedly and structural commonality exists. As a result, modern solvers use an array of different diving heuristics during branch and bound. Selectively tuning this diving ensemble can be effective in improving solver performance on particular applications (Section 7.4). In contrast, our approach that we discuss in the next section is to directly learn problem-specific diving heuristics for particular applications. We demonstrate in our experiments that this yields improvements over using a (tuned) ensemble of divers as is common practice.

7.2 LEARNING TO DIVE

We propose *L2Dive* to learn application-specific diving heuristics with graph neural networks. *L2Dive* uses a generative model to predict an assignment for the integer variables of a given mixed integer linear program. This model is learnt from a distribution of good feasible solutions collected initially for a set of training instances of a particular application. The model is a graph neural network closely related to the model in we described in chapter 5. It is trained to minimize a variational objective. At test time, *L2Dive* leverages insights from the duality of linear programs to select variables and tighten their bounds based on the model's predictions. We fully integrate *L2Dive* into the open-source solver SCIP.

Diver	Description
Coefficient	Selects the variable with the minimal number of (positive) up-locks or down-locks and bounds it in the corresponding direction; ties are broken using *Fractional* diving.
Distribution	Selects a variable based on the solution density following [232].
Farkas	Attempts to construct a Farkas proof, *Farkas* diving bounds a variable in the direction that improves the objective value and selects the variable for which the improvement in the objective is largest.
Fractional	Selects the variable with the lowest fractionality $\lvert x_j^* - \lfloor x_j^* + 0.5 \rfloor \rvert$ in the current LP solution x^* and bounds it in the corresponding direction.
Linesearch	Considers the ray originating at the LP solution of the root and passing through the current node's LP solution x^*, selects the variable j whose coordinate hyperplane $x_j = \lfloor x_j^* \rfloor$ or $x_j = \lceil x_j^* \rceil$ is first intersected by the ray.
Pseudocost	Selects and bounds a variable based on its branching pseudocosts, its fractionality and the LP solutions at the root and the current node.
Vectorlength	Inspired by set partition constraints, selects a variable for which the quotient between the objective cost from bounding it and the number of constraint it appears in is smallest.

TABLE 7.1: Overview of standard diving heuristics.

7.2.1 *Learning from feasible solutions*

We propose to learn a generative model to predict good feasible solutions for any given integer program. For this purpose, we first pose a conditional probability distribution over the variables $\boldsymbol{x}$:

$$\log \tilde{p}_\tau(\boldsymbol{x} \mid \boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}}) :\propto \begin{cases} -\boldsymbol{c}^\intercal \boldsymbol{x}/\tau & \text{if } \boldsymbol{x} \in \mathcal{X}' \\ -\infty & \text{else} \end{cases} \tag{7.2}$$

The distribution $\tilde{p}_\tau(\boldsymbol{x}|\cdot)$ is conditioned on a given instance as in equation (5.1) with $(\boldsymbol{c}, \boldsymbol{A}, \boldsymbol{b}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ and defined with respect to a set of good feasible solutions $\mathcal{X}' \subseteq \mathcal{X}$ for this instance. Solutions $\boldsymbol{x} \in \mathcal{X}'$ with a better objective are given larger mass as regulated by the temperature τ, while solutions that are not known to be feasible or are not good ($\boldsymbol{x} \notin \mathcal{X}'$) are given no probability mass. In practice, a model for diving will only need to make predictions on the divable variables $\boldsymbol{x}_\mathcal{D}$ that are integral, non-fixed and not *slack* (Section 7.1). Hence, our model will target the marginal distribution $p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot) := \sum_{\boldsymbol{x}' \in \mathcal{X}'} \mathbb{I}(\boldsymbol{x}_\mathcal{D} = \boldsymbol{x}'_\mathcal{D})\, \tilde{p}_\tau(\boldsymbol{x}'|\cdot)$.

Our goal is to learn a generative model $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$ that closely approximates the distribution $p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot)$. The model $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$ will be used to make predictions on unseen test instances for which $p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot)$ and $\mathcal{X}'$ are unknown. To learn a good generative model, our objective is to minimize the Kullback-Leibler divergence between $p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot)$ and $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$,

$$L(p_\tau, \hat{p}) := \mathrm{KL}(p_\tau || \hat{p}) = \sum_{\boldsymbol{x}_\mathcal{D}} p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot) \log \left(\frac{p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot)}{\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)} \right) \tag{7.3}$$

jointly over all training instances. The sum in (7.3) can be evaluated exactly, because the number of good feasible solutions in $\mathcal{X}' \ni \boldsymbol{x}_\mathcal{D}$ (for which $p_\tau(\boldsymbol{x}_\mathcal{D}|\cdot) > 0$) tends to be small. Our model predicts a probability $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$ for any assignment of the divable variables $\boldsymbol{x}_\mathcal{D}$ based on the integer program and the current solver state, but we suppress the dependence of $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$ on $\boldsymbol{x}_\mathrm{sup}$ and $\boldsymbol{\theta}_\mathbf{sup}$ for notational convenience as before. We choose to make conditionally independent predictions. For each variable, we choose an appropriate probability distribution and our model outputs its parameters. For binary variables, we choose a Bernoulli distribution, i.e. $x_j \sim \mathrm{Bern}(\mu_j)$ and the model outputs its mean parameter $\mu_j = \mu_j(\boldsymbol{x}_\mathrm{sup}, \boldsymbol{\theta}_\mathbf{sup})$. Binary variables are the most common variables in integer programs and all the instances we considered feature exclusively binary divable variables. Nevertheless, for general integer variables we suggest to consider the bitwise representation

of the integer domain. For example, the domain of a variable that can assume no more than eight unique integer values can be represented using at most four bits. Each of these bits can be parameterized with its own Bernoulli distribution and the model outputs a mean for each. When the maximum size of the domain of any divable variable is known a priori, the model can output an array of fixed size for each variable and superfluous bits may be ignored for variables with smaller domains. This is more likely to be the case for diving than for other applications, because *slack variables* from cutting plane constraints (whose domains are not known initially and may be large) are not divable. In cases where outputting a fixed-size array of Bernoulli parameters is not applicable, a variable length array could be outputted by using a recurrent layer, such as an LSTM [233], as is proposed in Nair *et al.* [197][2]. Although conditionally independent predictions for each variable limit our model to unimodal distributions, this design choice delivered strong empirical performance in our experiments (Section 7.4). It is possible to choose more delicate models for $\hat{p}(\boldsymbol{x}_{\mathcal{D}}|\cdot)$, such as autoregressive models, but those will typically impose a larger cost for evaluations. Our model is a variant of the graph neural network described in chapter 5. As in chapter 6, we use an extended feature set (Appendix B.6.1) and batch normalization instead of layer normalization. We adapt the output layer to parameterize a probability distribution over variable assignments as described above.

SOLUTION AUGMENTATION At train time, we rely on the availability a set of good feasible solutions $\mathcal{X}'$ for each instance. This is required to evaluate $p_\tau(\boldsymbol{x}_{\mathcal{D}}|\cdot)$ and the objective in (7.3). Several choices for $\mathcal{X}'$ are possible, and the effectiveness of our approach may depend on them. For example, if $\mathcal{X}'$ only contains trivial solutions, we cannot reasonably hope to learn any non-trivial integer variable assignments. The most obvious choice perhaps is to let $\mathcal{X}' = \{x^\star\}$, where $x^\star$ is the best solution the solver finds within a given time limit T. However, solvers are typically configured to not only store $x^\star$, but also a set number of its predecessors. Thus, alternatively some or all of the solutions in store could be used to define $\mathcal{X}'$ at no additional expense. Lastly, many instances of combinatorial optimization (e.g., set cover, independent set) exhibit strong symmetries and multiple

2 Alternatively, a fixed size array could still be used with an additional bit to indicate integer overflow that must be handled appropriately. For example in diving, variables for which the model predicts overflow may be ignored. Further, the frequency with which overflow would be encountered in practice could plausibly be reduced by making predictions relative to the current solution $\lfloor x_j^* \rfloor$ rather than with respect to the lower variable bound $\underline{\pi}_j$.

solutions with the same objective value may exist as a result. These can be identified by using a standard solver to enumerate the solutions of an additional auxiliary mixed integer linear program,

$$\min_{\boldsymbol{x} \in \mathcal{X}^\star} \boldsymbol{c}^\intercal \boldsymbol{x}, \qquad \mathcal{X}^\star = \mathcal{X} \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c}^\intercal \boldsymbol{x} = z^\star\} \tag{7.4}$$

where the optimal objective $z^\star$ is identified from a previous call to the solver on the program in (5.1). Standard solvers can enumerate the set $\mathcal{X}^\star$ in (7.4) with the aid of a constraint handler that is used whenever a new feasible integer variable assignment is found to continue the solving process. We used this method to augment solutions and chose $\mathcal{X}' = \mathcal{X}^\star$ for some of our experiments. This technique may be of independent interest, because the problem of handling symmetries is ubiquitous in machine learning for combinatorial optimization [234]. While it can be expensive to collect feasible solutions for each training instance regardless of the choice of $\mathcal{X}'$, this cost may be curbed, because we do not require $\mathcal{X}'$ to contain the optimal solution. Moreover, the cost is incurred only once and ahead of training, such that all solver calls are embarassingly parallelizable across instances. In some cases, a practitioner may be able to draw on previous solver logs and not be required to expend additional budget for data collection. Finally, any training expense will be ultimately amortized in test time service.

7.2.2 *Using a generative model for diving*

At test time, we use our generative model $\hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$ to predict an assignment for the divable integer variables $\boldsymbol{x}_\mathcal{D}$ of a given instance. We choose $\hat{\boldsymbol{x}}_\mathcal{D} = \arg\max \hat{p}(\boldsymbol{x}_\mathcal{D}|\cdot)$, but assignments could also be sampled from the model, for example if multiple dives are attempted in parallel. To use the prediction for diving, we need to decide which variable to select (line 3 in Algorithm 3) and how to tighten its bounds (line 4). Ideally, our decision rules will admit a feasible solution at shallow depths, i.e., only a few bounds must be tightened to result in an integral or roundable solution to the diving linear program. Which variables should we tighten for this purpose and how? Compellingly, the theory of linear programming affords some insight:

Proposition 5. *Let $\boldsymbol{x}' \in \mathcal{X}$ be a feasible solution for a mixed integer linear program as in* (5.1) *with relaxation as in* (5.2)*. For* (5.2)*, the dual linear program is as defined in* (A.28) *in Appendix A.5. Let $\boldsymbol{v}^* := (\boldsymbol{v}_b^*, \boldsymbol{v}_{\underline{\boldsymbol{\pi}}}^*, \boldsymbol{v}_{\overline{\boldsymbol{\pi}}}^*)$ be an optimal*

solution for this dual program. Let $\underline{\mathcal{J}}(\boldsymbol{x}')$ *and* $\overline{\mathcal{J}}(\boldsymbol{x}')$ *index the set of variables that violate complementary slackness* (A.29) *for* $\boldsymbol{x}'$ *and* $\boldsymbol{v}^*$*, such that*

$$\underline{\mathcal{J}}(\boldsymbol{x}') := \{j \mid \left(x'_j - \underline{\pi}_j\right) v^*_{\underline{\pi}_j} > 0\}$$
$$\overline{\mathcal{J}}(\boldsymbol{x}') := \{j \mid \left(x'_j - \overline{\pi}_j\right) v^*_{\overline{\pi}_j} > 0\}$$

and define $\mathcal{J}(\boldsymbol{x}') := \underline{\mathcal{J}}(\boldsymbol{x}') \cup \overline{\mathcal{J}}(\boldsymbol{x}')$*. Now consider the linear program*

$$\arg \min_{\boldsymbol{x} \in P_{\mathcal{J}(\boldsymbol{x}')}} \boldsymbol{c}^\intercal \boldsymbol{x} \tag{7.5}$$

where the bounds of all variables indexed by $\mathcal{J}(\boldsymbol{x})$ *are tightened, such that*

$$P_{\mathcal{J}(\boldsymbol{x}')} = P \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid x_j \geq x'_j \; \forall j \in \underline{\mathcal{J}}(\boldsymbol{x}'),\; x_j \leq x'_j \; \forall j \in \overline{\mathcal{J}}(\boldsymbol{x}')\}$$

Then, $\boldsymbol{x}' \in \arg\min_{\boldsymbol{x} \in P_{\mathcal{J}(\boldsymbol{x}')}} \boldsymbol{c}^\intercal \boldsymbol{x}$*.*

Proof. $\boldsymbol{x}'$ is clearly a feasible solution for the program in (7.5). $\boldsymbol{v}^*$ is a feasible solution for the dual linear program of (7.5), because it is feasible for the dual linear program of (5.2). The additional bound changes in (7.5) only change the objective of the dual. $\boldsymbol{x}'$ and $\boldsymbol{v}^*$ satisfy complementary slackness, hence $\boldsymbol{x}'$ is optimal for (7.5). □

This suggests that for a prediction $\hat{\boldsymbol{x}}_{\mathcal{D}}$, variables in $\mathcal{J}(\hat{\boldsymbol{x}}_{\mathcal{D}})$ should be tightened. If the integer variable assignment $\hat{\boldsymbol{x}}_{\mathcal{D}}$ is feasible and the candidate set includes all integer variables, this will yield a diving linear program for which the assignment is optimal and that may be detected by the LP solver. Unfortunately, this is not guaranteed in the presence of *slack* variables (where typically $\mathcal{D} \subset \mathcal{I}$) or if multiple optimal solutions exist (some of which may not be integer feasible). In practice, it may thus be necessary to tighten additional variables in $\mathcal{D}$ and we propose the following rule to select a variable $j^\star \in \mathcal{D}$ for tightening

$$j^\star = \arg\max_{j \in \mathcal{D}} \hat{y}_j := \hat{p}(\hat{x}_j|\cdot) + \mathbb{I}(j \in \mathcal{J}(\hat{\boldsymbol{x}}_{\mathcal{D}})) \tag{7.6}$$

This rule will select any variables in $\mathcal{J}(\hat{\boldsymbol{x}}_{\mathcal{D}})$ before considering other variables for tightening and favors those variables in whose predictions the model is most confident in. Conveniently, the set $\mathcal{J}(\hat{\boldsymbol{x}})$ can be easily computed on the fly from the dual values $\boldsymbol{v}^*$, which standard LP solvers readily emit on every call at no additional expense.

7.2.3 *Deployment*

We fully integrate *L2Dive* into the open-source solver SCIP 7.0.2 [193]. The solver exposes a plug-in for diving heuristics that implements an optimized version of Algorithm 3 in the programming language C. We extend the solver's Python interface [235] to include this plug-in and use it to realize *L2Dive*. This integration facilitates direct comparison to all standard divers implemented in SCIP (Section 7.4.1) and makes it easy to include *L2Dive* in SCIP for use in branch and bound (Section 7.4.2). Importantly, we call our generative model only once at the initiation of a dive to predict a variable assignment. While predictions may potentially improve with additional calls at deeper depths, this limits the in-service overhead of our method. It also simplifies the collection of training data and produced good results in our experiments (Section 7.4).

7.3 RELATED WORK

Nair *et al.* [197] propose a method that learns to tighten a subset of the variable bounds. It spawns a smaller sub-integer program which is then solved with an off-the-shelf branch and bound solver to find feasible solutions for the original program. Sonnerat *et al.* [213] improve this approach using imitation learning. Others explore reinforcement learning [211] or hybrids [210], but only focus on improving primal performance. All of these methods are variants of large neighborhood search [236–238], where a neighborhood for local search is not proposed heuristically, but learnt instead. In contrast, our approach *L2Dive* does not propose a fixed neighborhood and it does not require access to a branch and bound solver to run. Instead, we use our model's predictions to iteratively modify and solve linear programs instead of sub-integer programs. In practice, linear programs tend to solve significantly faster which makes *L2Dive* more applicable.

7.4 EXPERIMENTS

We evaluated the effectiveness of *L2Dive* in two different experiments and on a total of six datasets. The first set of experiments (Section 7.4.1) was designed to study the diving performance of *L2Dive* in isolation and compare it against existing diving heuristics. On a benchmark of four synthetic combinatorial optimization problems from previous work [200], we performed single dives with each diver and measured the average

primal gap. We found that *L2Dive* outperformed all existing divers on every dataset and produced the best solutions amongst all divers. The second set of experiments (Section 7.4.2) directly included *L2Dive* into the branch and bound process of the open-source solver SCIP. The solver called *L2Dive* in place of existing diving heuristics and our goal was to improve overall performance on real-world mixed integer linear programs. We considered instances from neural network verification [197] and server load balancing in distributed computing [215]. We measured performance with *L2Dive* against the default configuration and a highly challenging baseline that tuned the solver's diving ensemble. We found that *L2Dive* improved the average primal-dual integral by 7% (35%) on load balancing and improved average solving time by 20% (29%) on neural network verification over the tuned (default) solver.

We collected data for training and validation: In all experiments, we extracted the bipartite graph input representation described in chapter 5 with the features listed in Appendix B.6.1 at each instance's root node. On all but two datasets, we chose $\mathcal{X}' = \{x^\star\}$ where $x^\star$ is the best solution the solver finds within a given time limit T. For set cover and maximum independent set only, we observed strong symmetries and used the solution augmentation described in section 7.2. We trained separate models for each dataset. We trained each model with ADAM [227] for 100 epochs in the first set of experiments and for 10 epochs in the second set of experiments. We individually tuned the learning rate from a grid of $[10^{-2}, 10^{-3}, 10^{-4}]$. For each dataset, we chose the batch size to exhaust the memory of a single NVIDIA GeForce GTX 1080 Ti device. We validated after every epoch and chose the model that achieved the best validation loss. In all experiments, we use the mode prediction of the generative model and only perform a single dive from a given node. We do not attempt multiple dives in parallel and did not use any accelerators at test time. In all experiments, we only call *L2Dive*'s generative model once at the beginning of a dive to limit the in-service overhead from serving the graph neural network.

7.4.1 *Diving with* L2Dive

With this first set of experiments, we studied the diving performance of *L2Dive* and compared it against existing diving heuristics. We used the same benchmark as previous work [200]. This benchmark consists of four different classes of combinatorial optimization problems, including set covering, combinatorial auctions, capacitated facility location and maximum

	Set Cover	Comb. Auction	Fac. Location	Ind. Set
L2Dive	**55** (3)	**222** (7)	**160** (10)	**5** (1)
Best heuristic	*95* (3)	*256* (8)	*484* (7)	*18* (2)
Coefficient	3,700 (55)	671 (11)	762 (9)	246 (4)
Distributional	3,900 (50)	1,504 (12)	760 (9)	196 (3)
Farkas	105 (3)	476 (9)	*484* (7)	-
Fractional	3,726 (57)	672 (10)	1,058 (11)	232 (4)
Linesearch	1,269 (24)	467 (10)	1,036 (15)	77 (1)
Pseudocost	195 (9)	*256* (8)	505 (11)	32 (2)
Vectorlength	*95* (3)	832 (20)	840 (19)	*18* (1)
Random	416 (13)	704 (12)	902 (14)	78 (2)
Lower	2,918 (63)	1,587 (11)	623 (8)	171 (5)
Upper	239 (6)	611 (11)	828 (14)	62 (2)

TABLE 7.3: *L2Dive* finds better feasible solution on all four problem classes than existing diving heuristics. Average primal gap with standard error on test set. Best diver is in **bold** and best heuristic is in *italics*.

independent sets. For each class, we trained on 1000 instances and validated and tested on 500 instances. We presolved all instances before diving, but did not branch and disabled cutting planes and other primal heuristics as our interest is solely in diving. We compared *L2Dive* against all other standard diving heuristics in Table 7.1. This includes *Coefficient*, *Fractional*, *Linesearch*, *Pseudocost*, *Distributional*, *Vectorlength* [184] and *Farkas* diving [239]. For all of these divers, we use the optimized implementation made available in the SCIP solver. In addition, we considered three trivial divers that respectively fix integer variables to their lower (*Lower*) or upper limit (*Upper*) or either with equal probability (*Random*). All divers ran with the same diving budget ($d_{max} = 100$) and their execution was enforced after resolving the root node. We ignore the few test instance that were solved directly at root by SCIP before we could initiate a dive.

L2Dive outperformed all other standard divers (Table 7.3). It achieved the lowest average test primal gap on each of the four problem classes. The improvements over the *best heuristic* diver ranged from roughly 15% for combinatorial auctions to more than 70% for independent sets. The trivial

divers only found solutions that are significantly worse, which indicates that *L2Dive* learnt to exploit more subtle patterns in the problem instances to find better feasible solutions. Some baseline divers (e.g., *Linesearch*, *Distributional*) failed to consistently outperform the trivial divers across all problem classes and *best heuristic* diver varied (*Pseudocost* diving for combinatorial auctions, *Farkas* diving for facility location, *Vectorlength* diving for set cover, independent set). This confirms that in practice most diving heuristics while generic tend to only work well for some problem classes. *L2Dive* is a generic recipe to design effective divers for any specific application. Finally, we found that the mode predictions of our learnt models were rarely feasible (e.g., set cover, combinatorial auctions) or yielded poor solutions (e.g., independent set). This highlights that learning a generative model for diving may be a more promising approach than trying to predict feasible solutions directly.

7.4.2 L2Dive *in branch and bound*

With this second set of experiments, our goal was to use *L2Dive* within branch and bound to improve solver performance on real-world mixed integer linear programs. To this end, we included *L2Dive* into the open-source solver SCIP. We disabled all other diving heuristics in SCIP and dive with *L2Dive* from the root node. We found this to work well, but results for *L2Dive* may improve further with a more subtle schedule for *L2Dive* or by integrating *L2Dive* into the solver's diving ensemble.

We considered two strongly contrasting applications from previous work. The first application deals with safely balancing workloads across servers in a large-scale distributed compute cluster. This problem is an instance of bin packing with apportionment and can be formulated as a mixed integer linear program. We used the dataset from Gasse *et al.* [215] which contains 9900 instances for training and 100 instances for validation and testing respectively. Solving these instances to optimality is prohibitively hard[3] and we therefore set a time limit of $T_{\text{limit}} = 900$ seconds for data collection and test time evaluations. The second application deals with verifying the robustness of neural networks as in chapter 6. We use the same instances as previously and set a limit of $T_{\text{limit}} = 3600$ seconds for data collection and test time evaluations.

3 Using a Xeon Gold 5118 CPU processor with 2.3 GHz and 8 GB of RAM, none of the instances could be solved with SCIP 7.0.2 at default settings within an hour.

To assess solver performance, we measure solving time T for neural network verification and the primal-dual integral $\Gamma_{pd}(T)$ for server load balancing. Both measures fully account for the entire in-service overhead of *L2Dive* (e.g., computing the bipartite graph representation from the tree node, forward-propagating the generative model, diving etc.), because the *L2Dive* diver is directly included into SCIP and called by the solver during the branch and bound process. Our experiments were conducted on a shared distributed compute cluster. To reduce measurement variability, we ran all test time evaluations repeatedly on machines equipped with the same Intel Xeon Gold 5118 CPU 2.3 GHz processors for three different seedings of the solver. We batched evaluations randomly across instances and methods to be processed sequentially on the same machine. We report test set means and standard errors over the three different random seeds.

	Load Balancing		Neural Network Verif.	
	Primal-dual Integral	Wins	Solving Time	Wins
SCIP				
Default	4,407 (34)	0 (0)	55.8 (2.3)	54 (5)
No diving	4,221 (21)	0 (0)	53.7 (0.6)	40 (4)
Tuned	3,067 (10)	7 (3)	49.9 (2.8)	164 (3)
L2Dive	**2,863 (13)**	**93 (3)**	**39.8 (2.3)**	**287 (5)**

TABLE 7.4: *L2Dive* improves the performance of the branch and bound solver SCIP on real-world applications. When using *L2Dive* instead of standard divers, the average primal-dual integral for load balancing improves by 7% (35%) and solving time on neural network verification shrinks by 20% (29%) against the tuned (default) solver.

L2Dive improved solver performance ad-hoc (Table 7.4, *L2Dive*). On load balancing, *L2Dive* improved the average primal-dual integral by over 30% from the solver at default settings (Table 7.4, *Default*). On neural network verification, *L2Dive* reduced the average solving time from approximately 56 seconds to less than 40 seconds (35%). As a control, we also ran SCIP without any diving and surprisingly found small improvements on both datasets (Table 7.4, *No diving*). The solver's default setting are calibrated on a general purpose set of mixed integer programs and are typically a challenging baseline to beat. However, our results suggests that SCIP's

divers are either ineffective or may be poorly calibrated for these two applications. For this reason, we decided to tune the diving heuristics of the solver to seek an even more challenging baseline for comparison. We leveraged expert knowledge and random search to find strong diving ensembles in the the vicinity of the default configuration. Then, we selected the best tuned solver configuration on a validation set using the same budget of solver calls that *L2Dive* expended for data collection. Details are in Appendix B.6.3. Our tuned solver baseline (Table 7.4, *Tuned*) significantly improved performance over *Default*, but was still outperformed by *L2Dive*. This highlights that our approach to learn specific divers may be more promising than fitting ensembles of generic diving heuristics to a particular application. Overall, *L2Dive* achieved the best average performance on 93 (out of 100) test instances for load balancing, and achieved the best average performance on 287 (out of 545) test instances for neural network verification, more than the three SCIP configurations collectively.

7.5 DISCUSSION

In this chapter we presented *L2Dive* to learn application-specific diving heuristics in branch and bound. Our approach combines ideas from generative modeling and relational learning with a profound understanding of integer programs and their solvers. We tested *L2Dive* on a range of applications including combinatorial optimization problems, workload apportionment and neural network verification. It found better feasible solutions than existing diving heuristics and facilitated improvements in overall solver performance. We view our work as yet another example that demonstrates the fruitful symbiosis of learning and search to design powerful algorithms.

There are limitations in learning diving heuristics for specific applications. For example, in some cases the set of training instances may be small or collecting feasible solutions could be prohibitively expensive. In such cases, it may be desirable to transfer models from other applications or to utilize self-supervised representations that require fewer labelled examples for training [44]. This is a natural direction to explore in the future, for this and other work at the intersection of machine learning and integer programming.

8

CONCLUSION

8.1 SUMMARY

We presented methods for learning *with* and *for* discrete optimization. Our contributions are two-fold. In learning *with* discrete optimization, we focussed on gradient estimation for models in non-supervised learning that involve discrete variables. Such models appear in unsupervised, self-supervised and reinforcement learning and discrete variables provide benefits in terms of regularization, interpretability, model design and algorithmic integration. Relying on efficient methods in discrete optimization, we designed new gradient estimators for non-supervised machine learning via relaxations. We demonstrated experimentally that these facilitate learning models that are more performant, useful and efficient. In learning *for* discrete optimization, we focussed on improving the performance of branch and bound solvers with machine learning. We identified opportunities to replace existing heuristic subroutines designed by domain experts with learnt models that are tailored to specific applications. In particular, we developed methods to learn models for cutting plane selection and diving. Our methods combine ideas from imitation learning and generative modeling on relational data with a profound understanding of integer programs and modern branch and bound solvers. Experimentally, we demonstrated that our methods outperform existing heuristics and competing machine learning approaches and can facilitate overall improvements in solver performance.

8.2 OUTLOOK

There are opportunities for future research in this thesis. In the preceding chapters, we discussed some immediate ideas to extend the methods presented there. Beyond this, there are other exciting directions. With regard to our work on learning with discrete optimization, these include the use of approximate methods to compute relaxations and applications in reinforcement learning.

RELAXATIONS VIA APPROXIMATIONS Our gradient estimators for non-supervised learning with discrete variables require methods to compute the solution of the relaxation exactly. As we demonstrated in chapter 3, this is possible for a variety of problems. However, in some cases methods that approximate the solution instead of computing it exactly could be easier to implement or computationally more efficient. For example, our gradient estimator for correlated k-subsets is computed via dynamic programming which scales unfavorably on accelerators and we could consider heuristics to reduce the size of the program or approximate its solution. Thus, it may be desirable if we could rely on approximations to design relaxed gradient estimators instead. This raises interesting theoretical and practical questions. What approximations admit a valid reparameterization gradient? How can it be computed (or approximated)? Do estimators that use approximations suffer from larger bias and variance or can they improve gradient estimation in some cases? Do they suffer from a larger integrality gap? Can we retain control over the faithfulness of the relaxation when we use approximations?

REINFORCEMENT LEARNING Relaxed gradient estimators require that the loss function can be evaluated on and differentiated with respect to the solution of the relaxation. This is typically not the case in reinforcement learning, if the model learns a policy solely from interactions with a black-box environment whose dynamics are unknown and unmodelled. However, in cases where a model for the environment is known or learnt and differentiable, there are opportunities for the use of relaxed gradient estimation. For example, Ha & Schmidhuber [240] learn generative models of reinforcement learning environments and use these to train a policy via evolutionary strategies. Instead, relaxed gradient estimators could be considered. In addition, several authors recently presented differentiable simulators for rigid body physics [see e.g., 241, 242] that are suitable for relaxed gradient estimation in some cases. Finally, Christiano *et al.* [243] align large language models with the aid of a learnt reward model. They use traditional algorithms from reinforcement learning, but the reward model is differentiable and relaxed gradient estimation may be a promising alternative. As in other applications, this offers the opportunity to leverage the differentiability of the loss function and may more effectively trade off bias and variance to learn powerful models.

With regard to our work on learning for discrete optimization, exciting directions for future research include the use of machine learning for other

components of branch and bound solvers and learning models that are not application-specific but general-purpose.

NEURAL BRANCH AND BOUND In this thesis, we presented methods to learn models for cutting plane selection and diving in branch and bound. Our work complements existing work that has considered variable selection, node selection or primal heuristics and that we surveyed in chapter 5. Yet, there are many components in modern branch and bound solvers that remain for machine learning to explore. For example, with respect to cutting planes, we focussed on the problem of cutting plane selection, i.e., choosing a cutting plane from a pool of available candidates. However, cutting plane management in branch and bound also entails numerous other decisions, among them choosing how many cutting planes to add or which ones to remove and scheduling separation rounds or individual separators. All of these components present opportunities for machine learning to improve the performance of branch and bound solvers. Based on the results of this thesis, we believe that the most promising methods are those that impose a limited computational overhead from model evaluations and can learn from the rich data that solvers produce and integer programming facilitates. Ultimately, the integration of an increasingly larger number of learnt components may result in *neural branch and bound solvers*. Today, with the exception of [197], there is little work that combines multiple learnt components to improve solver performance, but this is a promising direction for future research.

GENERAL-PURPOSE COMPONENTS Most previous work, including the methods we presented, considers learning application-specific models for branch and bound. As we argued, this is relevant in practice, because numerous applications involve solving many similar integer programs. However, a case can be made for learning general-purpose components for use in branch and bound that could outperform existing heuristics broadly on diverse sets of integer programs. This could make machine learning for integer programming universally available or allow solving problems of unprecedented scale. From a learning perspective, learning general-purpose components instead of application-specific ones is more challenging for multiple reasons. When instances are more diverse, it is likely that larger data sets and models with larger capacity are required, while signal-noise ratios may be poorer. Moreover, instances that vary widely in size present unique challenges when graph neural networks are used as is common.

Many of these scale unfavorably with the size of the input graph and require careful management. There is hope to address these challenges in the future. With regard to data, the MIPLIB 2017 collection [222] is a benchmark of diverse integer programs, unfortunately it only contains 1065 instances. But, integer programs can easily be synthetically generated, and solvers can be used to collect labelled data. Thus, larger data sets may be facilitated. With regard to modeling, alternative architectures are subject to ongoing research in machine learning for integer programming [201]. While previous research focused on computational efficiency with respect to available hardware, model scalability with respect to the program instance is an equally important consideration.

A

PROOFS AND DEFINITIONS

A.1 PROPERTIES OF EXPONENTIALS AND GUMBELS

The properties of *Exponential* and *Gumbel* random variables are central to SMTs that have simple descriptions for the marginal $q_{\boldsymbol{\theta}}$. We review the important ones here. These are not new; many have been used for more elaborate algorithms that manipulate Gumbels [e.g., 104].

A Gumbel random variable $\boldsymbol{U} \sim \text{Gumbel}(\boldsymbol{\theta})$ for $\boldsymbol{\theta} \in \mathbb{R}^n$ is a location family distribution, which can be simulated from $\boldsymbol{U}_0 \sim \text{Uniform}(\mathbf{0}, \mathbf{1})$ using the identity

$$\boldsymbol{U} \stackrel{d}{=} \boldsymbol{\theta} - \log(-\log \boldsymbol{U}_0) \tag{A.1}$$

An exponential random variable $\boldsymbol{U} \sim \text{Exp}(\boldsymbol{\theta})$ for rate $\boldsymbol{\theta} > 0$ can be simulated from $\boldsymbol{U}_0 \sim \text{Uniform}(\mathbf{0}, \mathbf{1})$ using the identity

$$\boldsymbol{U} \stackrel{d}{=} -(1/\boldsymbol{\theta}) \log \boldsymbol{U}_0 \tag{A.2}$$

Any result for Exponentials immediately becomes a result for Gumbels, because they are monotonically related:

Proposition 6. *If* $\boldsymbol{U} \sim \text{Exp}(\boldsymbol{\theta})$, *then* $-\log \boldsymbol{U} \sim \text{Gumbel}(\log \boldsymbol{\theta})$.

Proof. Let $\boldsymbol{U} \stackrel{d}{=} -(1/\boldsymbol{\theta}) \log \boldsymbol{U}_0$. Now,

$$\begin{aligned} -\log \boldsymbol{U} &= -\log(-1/\boldsymbol{\theta} \log \boldsymbol{U}_0) \\ &= -(\log(1/\boldsymbol{\theta}) + \log(-\log \boldsymbol{U}_0)) \\ &= \log \boldsymbol{\theta} - \log(-\log \boldsymbol{U}_0) \sim \text{Gumbel}(\log \boldsymbol{\theta}) \end{aligned}$$

□

Although we prove results for Exponentials, using their monotonic relationship, all of these results have analogs from Gumbels.

The properties of Exponentials are summarized in the following proposition.

Proposition 7. *For* $U_i \sim \text{Exp}(\theta_i)$ *independent for* $\theta_i > 0$ *and* $i \in \{1, \ldots, n\}$, *let* $j^\star = \arg\min_i U_i$ *and* $U_{j^\star} = \min_i U_i$, *then*

1. $\mathbb{P}(i = j^\star) \propto \theta_i$
2. $U_{j^\star} \sim \text{Exp}(\sum_{i=1}^n \theta_i)$
3. *$j^\star$ and $U_{j^\star}$ are independent*
4. *Given $j^\star$ and $U_{j^\star}$, U_i for $i \neq j^\star$ are conditionally, mutually independent; exponentially distributed with rates θ_i; and truncated to be larger than $U_{j^\star}$.*

Proof. We can manipulate the joint density of $j^\star = \arg\min_i U_i$ and $U_{j^\star}$ as follows

$$\begin{aligned}
&\prod_{i=1}^n \theta_i \exp(-\theta_i u_{j^\star}) \, \mathbb{I}(u_i \geq u_{j^\star}) \\
&\quad = \theta_{j^\star} \exp(-\theta_{j^\star} u_{j^\star}) \prod_{i \neq j^\star} \theta_i \exp(-\theta_i u_i) \, \mathbb{I}(u_i \geq u_{j^\star}) \\
&\quad = \frac{\theta_{j^\star}}{\sum_{i=1}^n \theta_i} \left(\sum_{i=1}^n \theta_i \right) \exp(-\theta_{j^\star} u_{j^\star}) \prod_{i \neq j^\star} \theta_i \exp(-\theta_i u_i) \, \mathbb{I}(u_i \geq u_{j^\star}) \\
&\quad = \left[\frac{\theta_{j^\star}}{\sum_{i=1}^n \theta_i} \right] \left[\left(\sum_{i=1}^n \theta_i \right) \exp\left(- \sum_{i=1}^n \theta_i u_{j^\star} \right) \right] \left[\prod_{i \neq j^\star} \frac{\theta_i \exp(-\theta_i u_i)}{\exp(-\theta_i u_{j^\star})} \, \mathbb{I}(u_i \geq u_{j^\star}) \right]
\end{aligned} \tag{A.3}$$

While hard to parse, this manipulation reveals the all of the assertions of the proposition. □

Proposition 7 has a couple of corollaries. First, subtracting the minimum Exponential from a collection only affects the distribution of the minimum, leaving the distribution of the other Exponentials unchanged.

Corollary 1. *If $U_i \sim \text{Exp}(\theta_i)$ independent for $\theta_i > 0$ and $i \in \{1, \ldots, n\}$, then $U_i - U_{j^\star}$ are mutually independent and*

$$U_i - U_{j^\star} \sim \begin{cases} \text{Exp}(\theta_i) & i \neq j^\star \\ 0 & i = j^\star \end{cases}, \tag{A.4}$$

where $j^\star = \arg\min_i U_i$ as before.

Proof. Consider the change of variables $u_i' = u_i - u_{j^\star}$ in the joint (A.3). Each of the terms in the right hand product over $i \neq j^\star$ of (A.3) are transformed in the following way

$$\frac{\theta_i \exp(-\theta_i (u_i' + u_{j^\star}))}{\exp(-\theta_i u_{j^\star})} \, \mathbb{I}(u_i' + u_{j^\star} \geq u_{j^\star}) \longrightarrow \theta_i \exp(-\theta_i u_i') \tag{A.5}$$

This is essentially the memoryless property of Exponentials. Thus, the $U_i' = U_i - U_{j^\star}$ for $i \neq j^\star$ are distributed as Exponentials with rate θ_i and mutually independent. $U_{j^\star}'$ is the constant 0, which is independent of any random variable. Our result follows. □

Second, the process of sorting the collection U_i is equivalent to sampling from $\{1, \ldots, n\}$ without replacement with probabilities proportional to θ_i.

Corollary 2. *Let $U_i \sim \mathrm{Exp}(\theta_i)$ independent for $\theta_i > 0$ and $i \in \{1, \ldots, n\}$. Let $\mathrm{argsort}_{\boldsymbol{x}}$ be the argsort permutation of $\boldsymbol{x} \in \mathbb{R}^n$, i.e., the permutation such that x_{σ_i} where $\boldsymbol{\sigma} = \mathrm{argsort}_{\boldsymbol{x}}$ is in non-decreasing order. We have*

$$\mathbb{P}(\mathrm{argsort}_{\boldsymbol{U}} = \boldsymbol{\sigma}) = \prod_{i=1}^{n} \frac{\theta_{\sigma_i}}{\sum_{j=i}^{n} \theta_{\sigma_j}} \tag{A.6}$$

Given $\mathrm{argsort}_{\boldsymbol{U}} = \boldsymbol{\sigma}$, the sorted vector $\boldsymbol{U}_{\boldsymbol{\sigma}} = (\boldsymbol{U}_{\sigma_i})_{i=1}^n$ has the following distribution,

$$\begin{aligned} U_{\sigma_1} &\sim \mathrm{Exp}\left(\textstyle\sum_{j=1}^{n} \theta_{\sigma_j}\right) \\ U_{\sigma_i} - U_{\sigma_{i-1}} &\sim \mathrm{Exp}\left(\textstyle\sum_{j=i}^{n} \theta_{\sigma_j}\right) \end{aligned} \tag{A.7}$$

Proof. This follows after repeated, interleaved uses of Corollary 1 and Proposition 7. □

A.2 CONVEX CONJUGATE

Lemma 1. *Let $\mathcal{X} \subseteq \mathbb{R}^n$ be a finite, non-empty set, $P := \mathrm{conv}(\mathcal{X})$, and $\boldsymbol{u} \in \mathbb{R}^n$. We have,*

$$\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x} = \max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x} = \sup_{\boldsymbol{x} \in \mathrm{relint}(P)} \boldsymbol{u}^\intercal \boldsymbol{x} \tag{A.8}$$

If $\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ has a unique solution $\boldsymbol{x}^\star$, then $\boldsymbol{x}^\star$ is also the unique solution of $\max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$.

Proof. Assume w.l.o.g. that $\mathcal{X} = \{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^m\}$. Let $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$.

First, let us consider the linear program over $\mathcal{X}$ vs. P. Clearly, $\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x} \leq \max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$. In the other direction, for any $\boldsymbol{x}' \in P$, we can write $\boldsymbol{x}' = \sum_i \lambda_i \boldsymbol{x}^i$ for $\lambda_i \geq 0$ such that $\sum_i \lambda_i = 1$, and

$$\boldsymbol{u}^\intercal \boldsymbol{x}^\star = \sum_i \lambda_i \boldsymbol{u}^\intercal \boldsymbol{x}^\star \geq \sum_i \lambda_i \boldsymbol{u}^\intercal \boldsymbol{x}_i = \boldsymbol{u}^\intercal \boldsymbol{x}' \tag{A.9}$$

Hence $\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x} \geq \max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$. Thus $\boldsymbol{x}^\star \in \arg\max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$.

Second, let us consider the linear program over P vs. relint(P). The cases $\boldsymbol{x}^\star \in \text{relint}(P)$ or $\boldsymbol{u} = \mathbf{0}$ are trivial, so assume otherwise. Since $\boldsymbol{u}^\intercal \boldsymbol{x}^\star \geq \boldsymbol{u}^\intercal \boldsymbol{x}$ for $\boldsymbol{x} \in \text{relint}(P)$, it suffices to show that for all $\epsilon > 0$ there exists $\boldsymbol{x}_\epsilon \in \text{relint}(P)$ such that $\boldsymbol{u}^\intercal \boldsymbol{x}_\epsilon > \boldsymbol{u}^\intercal \boldsymbol{x}^\star - \epsilon$. To that end, take $\boldsymbol{x} \in \text{relint}(P)$ and $0 < \lambda < \min(\epsilon, \|\boldsymbol{u}\| \|\boldsymbol{x} - \boldsymbol{x}^\star\|)$, and define

$$x_\epsilon := \boldsymbol{x}^\star + \frac{\lambda}{\|u\| \|x - \boldsymbol{x}^\star\|}(x - \boldsymbol{x}^\star) \tag{A.10}$$

$x_\epsilon \in \text{relint}(P)$ by [112, Thm 6.1]. Thus, we get

$$\boldsymbol{u}^\intercal \boldsymbol{x}_\epsilon = \boldsymbol{u}^\intercal \boldsymbol{x}^\star + \lambda \frac{\boldsymbol{u}^\intercal (\boldsymbol{x} - \boldsymbol{x}^\star)}{\|\boldsymbol{u}\| \|\boldsymbol{x} - \boldsymbol{x}^\star\|} > \boldsymbol{u}^\intercal \boldsymbol{x}^\star - \epsilon \tag{A.11}$$

Finally, suppose that $\boldsymbol{x}^\star = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ is unique, but $\arg\max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$ contains more than just $\boldsymbol{x}^\star$. We will show this implies a contradiction. Let $i^\star$ be the index $i \in \{1, \ldots, m\}$ such that $\boldsymbol{x}^\star = x_{i^\star}$. Let $\boldsymbol{x}' \in \arg\max_{\boldsymbol{x} \in P} \boldsymbol{u}^\intercal \boldsymbol{x}$ be such that $y \neq \boldsymbol{x}^\star$. Then we may write $\boldsymbol{x}' = \sum_i \lambda_i \boldsymbol{x}^i$ for $\lambda_i \geq 0$ such that $\sum_i \lambda_i = 1$. But this leads to a contradiction,

$$\boldsymbol{u}^\intercal \boldsymbol{x}^\star = \sum_{i \neq i^\star} \frac{\lambda_i}{1 - \lambda_{i^\star}} \boldsymbol{u}^\intercal \boldsymbol{x}_i < \sum_{i \neq i^\star} \frac{\lambda_i}{1 - \lambda_{i^\star}} \boldsymbol{u}^\intercal \boldsymbol{x}^\star = \boldsymbol{u}^\intercal \boldsymbol{x}^\star \tag{A.12}$$

□

Lemma 2. *Let $P \subseteq \mathbb{R}^n$ be a non-empty convex polytope and $\boldsymbol{x}'$ an extreme point of P. Define the set,*

$$\mathcal{U}(\boldsymbol{x}) = \left\{ \boldsymbol{u} \in \mathbb{R}^n : \boldsymbol{u}^\intercal \boldsymbol{x} > \boldsymbol{u}^\intercal \boldsymbol{x},\ \forall \boldsymbol{x} \in P \setminus \{\boldsymbol{x}'\} \right\} \tag{A.13}$$

This is the set of utility vectors of a linear program over P whose argmax is the minimal face $\{\boldsymbol{x}'\} \subseteq P$. Then, for all $\boldsymbol{u} \in \mathcal{U}(\boldsymbol{x})$, there exists an open set $O \subseteq \mathcal{U}(\boldsymbol{x})$ containing $\boldsymbol{u}$.

Proof. Let $\boldsymbol{u} \in \mathcal{U}(\boldsymbol{x})$. Let $\{\boldsymbol{x}^1, \ldots, \boldsymbol{x}^m\} \subseteq P$ be the set of extreme points (there are finitely many), and assume w.l.o.g. that $x' = \boldsymbol{x}^m$. For each $\boldsymbol{x}^i \neq \boldsymbol{x}^m$ there exists $\epsilon_i > 0$ such that $\boldsymbol{u}^\intercal (\boldsymbol{x}^m - \boldsymbol{x}^i) > \epsilon_i$. Thus, for all $\boldsymbol{u}'$ in the open ball $B_{r_i}(\boldsymbol{u})$ of radius $r_i = \epsilon_i / \|\boldsymbol{x}^m - \boldsymbol{x}^i\|$ centered at $\boldsymbol{u}$, we have

$$\boldsymbol{u}'^\intercal (\boldsymbol{x}^m - \boldsymbol{x}^i) = \boldsymbol{u}^\intercal (\boldsymbol{x}^m - \boldsymbol{x}^i) + (\boldsymbol{u}' - \boldsymbol{u})^\intercal (\boldsymbol{x}^m - \boldsymbol{x}^i) > \boldsymbol{u}^\intercal (\boldsymbol{x}^m - \boldsymbol{x}^i) - \epsilon_i > 0 \tag{A.14}$$

Define $O = \cap_{i=1}^{m-1} B_{r_i}(\boldsymbol{u})$. Note, $\boldsymbol{u}'^\intercal \boldsymbol{x}^m > \boldsymbol{u}'^\intercal \boldsymbol{x}^i$ for all $\boldsymbol{u}' \in O, \boldsymbol{x}^i \neq \boldsymbol{x}^m$. Now, let $\boldsymbol{x}'' \in P \setminus \{\boldsymbol{x}^m\}$. Because P is the convex hull of the $\boldsymbol{x}^i$ [231, Thm. 2.9], we must have

$$\boldsymbol{x}'' = \sum_{i=1}^{m} \lambda_i \boldsymbol{x}^i \tag{A.15}$$

for $\lambda_i \geq 0$, $\sum_{i=1}^{m} \lambda_i = 1$ with at least one $\lambda_i > 0$ for $i < m$. Thus, for all $\boldsymbol{u}' \in O$

$$\boldsymbol{u}'^\intercal \boldsymbol{x}^m = \sum_{i=1}^{m} \lambda_i \boldsymbol{u}'^\intercal \boldsymbol{x}^m > \sum_{i=1}^{m} \lambda_i \boldsymbol{u}'^\intercal \boldsymbol{x}^i = \boldsymbol{u}'^\intercal \boldsymbol{x}'' \tag{A.16}$$

This implies that $O \subseteq \mathcal{U}(\boldsymbol{x}^m)$, which concludes the proof, as O is open, convex, and contains $\boldsymbol{u}$. □

Lemma 3 (Convex Conjugate)**.** *Given a non-empty, finite set $\mathcal{X} \subseteq \mathbb{R}^n$ and a proper, closed, strongly convex function $R : \mathbb{R}^n \to \{\mathbb{R}, \infty\}$ whose domain contains the relative interior of $P := \operatorname{conv}(\mathcal{X})$, let $R^\star = \min_{\boldsymbol{x} \in \mathbb{R}^n} R(\boldsymbol{x})$, and $\delta_P(\boldsymbol{x})$ be the indicator function of the polytope P,*

$$\delta_P(\boldsymbol{x}) = \begin{cases} 0 & \boldsymbol{x} \in P \\ \infty & \boldsymbol{x} \notin P \end{cases} \tag{A.17}$$

For $\tau \geq 0$, define

$$H_\tau(\boldsymbol{x}) := \tau(R(\boldsymbol{x}) - R^\star) + \delta_P(\boldsymbol{x}), \tag{A.18}$$

$$H_\tau^*(\boldsymbol{u}) := \sup_{\boldsymbol{x} \in \mathbb{R}^n} \boldsymbol{u}^\intercal \boldsymbol{x} - H_\tau(\boldsymbol{x}). \tag{A.19}$$

The following are true for $\tau > 0$,

1. (A.19) *has a unique solution, H_τ^* is continuously differentiable, twice differentiable a.e., and*

$$\nabla H_\tau^*(\boldsymbol{u}) = \arg\max_{\boldsymbol{x} \in \mathbb{R}^n} \boldsymbol{u}^\intercal \boldsymbol{x} - H_\tau(\boldsymbol{x}) \tag{A.20}$$

2. *If $\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}$ has a unique solution, then*

$$\lim_{\tau \to 0^+} \nabla H_\tau^*(\boldsymbol{u}) = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x} \tag{A.21}$$

Proof. Note, $\operatorname{relint}(P) \subseteq \operatorname{dom}(H_\tau) \subseteq P$.

1. Since H_τ is strongly convex [244, Lem. 5.20], (A.19) has a unique maximum [244, Thm. 5.25]. Moreover, H_τ^* is differentiable everywhere in $\mathbb{R}^n$ and its gradient ∇H_τ^* is Lipschitz continuous [244, Thm. 5.26]. By [112, Thm 25.5] ∇H_τ^* is a continuous function on $\mathbb{R}^n$. By Rademacher's theorem, ∇H_τ^* is a.e. differentiable. (A.20) follows by standard properties of the convex conjugate [112, Thm. 23.5, Thm. 25.1].

2. First, by Lemma 1,

$$H_0^*(\boldsymbol{u}) = \max_{\boldsymbol{x}\in P} \boldsymbol{u}^\intercal \boldsymbol{x} = \sup_{\boldsymbol{x}\in \operatorname{relint}(P)} \boldsymbol{u}^\intercal \boldsymbol{x} = \max_{\boldsymbol{x}\in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x}. \tag{A.22}$$

Since $\boldsymbol{u}$ is such that $\boldsymbol{u}^\intercal \boldsymbol{x}$ is uniquely maximized over P, H_0^* is differentiable at $\boldsymbol{u}$ by [112, Thm. 23.5, Thm. 25.1]. Again by Lemma 1 we have

$$\nabla H_0^*(\boldsymbol{u}) = \arg\max_{\boldsymbol{x}\in P} \boldsymbol{u}^\intercal \boldsymbol{x} = \arg\max_{\boldsymbol{x}\in \mathcal{X}} \boldsymbol{u}^\intercal \boldsymbol{x} \tag{A.23}$$

Hence, our aim is to show $\lim_{\tau\to 0^+} \nabla H_\tau^*(\boldsymbol{u}) = \nabla H_0^*(\boldsymbol{u})$. This is equivalent to showing that $\lim_{i\to\infty} \nabla H_{\tau_i}^*(\boldsymbol{u}) = \nabla H_0^*(\boldsymbol{u})$ for any $\tau_i > 0$ such that $\tau_i \to 0$. Let τ_i be such a sequence.

We will first show that $H_{\tau_i}^*(\boldsymbol{u}) \to H_0^*(\boldsymbol{u})$. For any $\boldsymbol{x}' \in \operatorname{relint}(P)$,

$$\begin{aligned}
\liminf_{i\to\infty} H_{\tau_i}^*(\boldsymbol{u}) &= \liminf_{i\to\infty} \sup_{j\geq i} \sup_{\boldsymbol{x}\in\mathbb{R}^n} \boldsymbol{u}^\intercal \boldsymbol{x} - H_{\tau_j}(\boldsymbol{x}) \\
&\geq \liminf_{i\to\infty} \sup_{j\geq i} \boldsymbol{u}^\intercal \boldsymbol{x}' - H_{\tau_j}(\boldsymbol{x}') \\
&= \boldsymbol{u}^\intercal \boldsymbol{x}'
\end{aligned}$$

Thus,

$$\liminf_{i\to\infty} H_{\tau_i}^*(\boldsymbol{u}) \geq \sup_{\boldsymbol{x}'\in\operatorname{relint}(P)} \boldsymbol{u}^\intercal \boldsymbol{x}' = H_0^*(\boldsymbol{u})$$

Since $\tau(R(\boldsymbol{x}) - R^\star) \geq 0$ for all $\boldsymbol{x} \in \mathbb{R}^n$, we also have

$$\begin{aligned}
\limsup_{i\to\infty} H_{\tau_i}^*(\boldsymbol{u}) &= \limsup_{i\to\infty} \sup_{\boldsymbol{x}\in\mathbb{R}^n} \boldsymbol{u}^\intercal \boldsymbol{x} - H_{\tau_i}(\boldsymbol{x}) \\
&\leq \limsup_{i\to\infty} \sup_{\boldsymbol{x}\in P} \boldsymbol{u}^\intercal \boldsymbol{x} = H_0^*(\boldsymbol{u})
\end{aligned}$$

Thus $\lim_{i\to\infty} H_{\tau_i}^*(\boldsymbol{u}) = H_0^*(\boldsymbol{u})$.

By Lemma 2, there exists an open convex set O containing $\boldsymbol{u}$ such that for all $\boldsymbol{u}' \in O$, $\nabla H_0^*(\boldsymbol{u}) = \arg\max_{\boldsymbol{x}\in P} \boldsymbol{u}'^\intercal x$. Again, H_0^* is differentiable

on O [112, Thm. 23.5, Thm. 25.1]. Using this and the fact that $H^*_{\tau_i}(\boldsymbol{u}) \to H^*_0(\boldsymbol{u})$, we get $\nabla H^*_{\tau_i}(\boldsymbol{u}) \to \nabla H^*_0(\boldsymbol{u})$ [112, Thm. 25.7].

□

A.3 CONVEX POSITION

Proposition 8. *Let $\mathcal{X} \subseteq \mathbb{R}^n$ be a non-empty finite set. If $\mathcal{X}$ is convex independent, i.e., for all $\boldsymbol{x} \in \mathcal{X}$, $\boldsymbol{x} \notin \operatorname{conv}(\mathcal{X} \setminus \{\boldsymbol{x}\})$, then $\mathcal{X}$ is the set of extreme points of $\operatorname{conv}(\mathcal{X})$. In particular, any non-empty set of binary vectors $\mathcal{X} \subseteq \{0,1\}^n$ is convex independent and thus the set of extreme points of $\operatorname{conv}(\mathcal{X})$.*

Proof. Let $\mathcal{X} = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^m\}$. The fact that the extreme points of $\operatorname{conv}(\mathcal{X})$ are in $\mathcal{X}$ is trivial. In the other direction, it is enough to show that $\boldsymbol{x}^m$ is an extreme point. Assume $\boldsymbol{x}^m \in \mathcal{X}$ is not an extreme point of $\operatorname{conv}(\mathcal{X})$. Then by definition, we can write $\boldsymbol{x}^m = \lambda \boldsymbol{x}' + (1-\lambda)\boldsymbol{x}''$ for $\boldsymbol{x}', \boldsymbol{x}'' \in \operatorname{conv}(\mathcal{X})$, $\lambda \in (0,1)$ with $\boldsymbol{x}' \neq \boldsymbol{x}^m$ and $\boldsymbol{x}'' \neq \boldsymbol{x}^m$. Then, we have that

$$\boldsymbol{x}^m = \sum_{i=1}^{m-1} \frac{\lambda\lambda'_i + (1-\lambda)\lambda''_i}{1 - \lambda\lambda'_m - (1-\lambda)\lambda''_m} \boldsymbol{x}^i \tag{A.24}$$

for some sequences $\lambda'_i, \lambda''_i \geq 0$ such that $\sum_{i=1}^m \lambda'_i = \sum_{i=1}^m \lambda''_i = 1$ and $\lambda'_m, \lambda''_m < 1$. This is clearly a contradiction of our assumption that $\boldsymbol{x}^m \notin \operatorname{conv}(\mathcal{X} \setminus \{\boldsymbol{x}^m\})$, since the weights in the summation (A.24) sum to unity. This implies that $\mathcal{X}$ are the extreme points of $\operatorname{conv}(\mathcal{X})$.

Let $\mathcal{X} \subseteq \{0,1\}^n$. It is enough to show that $\boldsymbol{x}^m \notin \operatorname{conv}(\{\boldsymbol{x}^1, \dots, \boldsymbol{x}^{m-1}\})$. Assume this is not the case. Let $\boldsymbol{c} = \boldsymbol{x}^m - (0.5)_{i=1}^n \in \mathbb{R}^n$, and note that $\boldsymbol{c}^\intercal \boldsymbol{x}^i < \boldsymbol{c}^\intercal \boldsymbol{x}^m$ for all $i \neq m$ when $\boldsymbol{x}^i$ are distinct binary vectors. But, this leads to a contradiction. By assumption we can express $\boldsymbol{x}^m$ as a convex combination of $\boldsymbol{x}^1, \dots, \boldsymbol{x}^{m-1}$. Thus, there exists $\lambda_i \geq 0$ such that $\sum_{i=1}^{m-1} \lambda_i = 1$, and

$$\boldsymbol{c}^\intercal \boldsymbol{x}^m = \sum_{i=1}^{m-1} \lambda_i \boldsymbol{c}^\intercal \boldsymbol{x}^i < \sum_{i=1}^{m-1} \lambda_i \boldsymbol{c}^\intercal \boldsymbol{x}^m = \boldsymbol{c}^\intercal \boldsymbol{x}^m \tag{A.25}$$

□

A.4 VARIANCE DECOMPOSITION FOR GRADIENT ESTIMATOR IN MINI-BATCHES

Proposition 4. *Let* $\hat{\boldsymbol{\eta}}_{\text{STGS}}$, $\hat{\boldsymbol{\eta}}_{\text{GR}}$ *and* $\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}$ *be the estimators defined in* (4.2), (4.3) *and* (4.8). *We have*

$$\begin{aligned}\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}\right] = &\frac{\mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}}|\boldsymbol{X}, \boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \\ &+ \frac{\mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{GR}}|\boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} + \frac{\mathbb{V}\left[\mathbb{E}\left[\hat{\boldsymbol{\eta}}_{\text{GR}}|\boldsymbol{x}_{\text{sup}}\right]\right]}{N_{\boldsymbol{x}_{\text{sup}}}}\end{aligned} \tag{4.9}$$

where $\mathbb{V}$ *is the trace of the covariance matrix.*

Proof. Let $\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}$ be defined as in 4.8. By the law of total variance, we have

$$\begin{aligned}\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{B-GRMC}}\right] = &\mathbb{E}\left[\mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \sum_{k=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right]\right] \\ &+ \mathbb{V}\left[\mathbb{E}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \sum_{k=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right]\right]\end{aligned} \tag{A.26}$$

The inner variance in the first term of (A.26) is over $\boldsymbol{U}^{ijk} \mid \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}$ which are all identically and independently distributed given $\boldsymbol{X}^{ij}$ and $\boldsymbol{x}_{\text{sup}}^{i}$. The outer expectation is over $\left(\boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right)$. Hence, this term simplifies to

$$\begin{aligned}&\mathbb{E}\left[\mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \sum_{k=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right]\right] \\ &\mathbb{E}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}}^2 N_{\boldsymbol{X}}^2} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \mathbb{V}\left[\frac{1}{N_{\boldsymbol{U}}} \sum_{k=1}^{N_{\boldsymbol{U}}} \hat{\boldsymbol{\eta}}_{\text{STGS}}^{ijk} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right]\right] \\ &= \frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{U}}} \mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{STGS}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right]\right]\end{aligned}$$

In the second term of (A.26), the inner expectation is again over the utilities and thus simplifies which the Gumbel-Rao estimator for $\boldsymbol{X}^{ij}$ and $\boldsymbol{x}^i_{\text{sup}}$. Accordingly, the second term simplifies to

$$\begin{aligned}
&\mathbb{V}\left[\mathbb{E}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}} N_{\boldsymbol{u}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \sum_{k=1}^{N_{\boldsymbol{u}}} \hat{\boldsymbol{\eta}}^{ijk}_{\text{STGS}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&= \mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \mathbb{E}\left[\frac{1}{N_{\boldsymbol{u}}} \sum_{k=1}^{N_{\boldsymbol{u}}} \hat{\boldsymbol{\eta}}^{ijk}_{\text{STGS}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&= \mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right]\right]
\end{aligned}$$

where the variance is over $\left(\boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right)$ and can be decomposed again by another application of the law of total variance:

$$\begin{aligned}
&\mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&= \mathbb{E}\left[\mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right] \,\middle|\, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&+ \mathbb{V}\left[\mathbb{E}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right] \,\middle|\, \boldsymbol{x}^i_{\text{sup}}\right]\right]
\end{aligned} \tag{A.27}$$

The inner variance in the first term of (A.27) is over $\boldsymbol{X}^{ij} \mid \boldsymbol{x}^i_{\text{sup}}$ which are all identically and independently distributed given $\boldsymbol{x}^i_{\text{sup}}$. The outer expectation is over $\boldsymbol{x}^i_{\text{sup}}$. Hence, this term simplifies to

$$\begin{aligned}
&\mathbb{E}\left[\mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right] \,\middle|\, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&= \mathbb{E}\left[\frac{1}{N^2_{\boldsymbol{x}_{\text{sup}}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \mathbb{V}\left[\frac{1}{N_{\boldsymbol{X}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}^{ij}_{\text{GR}} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}^i_{\text{sup}}\right] \,\middle|\, \boldsymbol{x}^i_{\text{sup}}\right]\right] \\
&= \frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \mathbb{E}\left[\mathbb{V}\left[\hat{\boldsymbol{\eta}}_{\text{GR}} \,\middle|\, \boldsymbol{x}_{\text{sup}}\right]\right]
\end{aligned}$$

In the second term of (A.27), the inner expectation is again over the categorical variables and accordingly

$$\begin{aligned}
&\mathbb{V}\left[\mathbb{E}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}} N_{\boldsymbol{X}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}_{\text{GR}}^{ij} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right] \,\middle|\, \boldsymbol{x}_{\text{sup}}^{i}\right]\right] \\
&= \mathbb{V}\left[\frac{1}{N_{\boldsymbol{x}_{\text{sup}}}} \sum_{i=1}^{N_{\boldsymbol{x}_{\text{sup}}}} \mathbb{E}\left[\frac{1}{N_{\boldsymbol{X}}} \sum_{j=1}^{N_{\boldsymbol{X}}} \left[\hat{\boldsymbol{\eta}}_{\text{GR}}^{ij} \,\middle|\, \boldsymbol{X}^{ij}, \boldsymbol{x}_{\text{sup}}^{i}\right] \,\middle|\, \boldsymbol{x}_{\text{sup}}^{i}\right]\right] \\
&= \frac{1}{N_{\boldsymbol{x}_{\text{sup}}}} \mathbb{V}\left[\mathbb{E}\left[\hat{\boldsymbol{\eta}}_{\text{GR}} \,\middle|\, \boldsymbol{x}_{\text{sup}}^{i}\right]\right]
\end{aligned}$$

This completes the proof. □

A.5 DUAL LINEAR PROGRAM AND COMPLEMENTARY SLACKNESS

Definition 3 (Dual linear program)**.** *The dual linear program of the program given in 5.2 is*

$$\max_{\boldsymbol{v}_{\boldsymbol{b}}, \boldsymbol{v}_{\underline{\boldsymbol{\pi}}}, \boldsymbol{v}_{\overline{\boldsymbol{\pi}}}} \boldsymbol{v}_{\boldsymbol{b}}^{\intercal}\boldsymbol{b} + \boldsymbol{v}_{\underline{\boldsymbol{\pi}}}^{\intercal}\underline{\boldsymbol{\pi}} + \boldsymbol{v}_{\overline{\boldsymbol{\pi}}}^{\intercal}\overline{\boldsymbol{\pi}} \quad \text{subject to } \boldsymbol{A}^{\intercal}\boldsymbol{v}_{\boldsymbol{b}} + \boldsymbol{v}_{\underline{\boldsymbol{\pi}}} + \boldsymbol{v}_{\overline{\boldsymbol{\pi}}} = \boldsymbol{c},\ \boldsymbol{v}_{\underline{\boldsymbol{\pi}}} \geq 0, \boldsymbol{v}_{\overline{\boldsymbol{\pi}}} \leq 0, \tag{A.28}$$

where $\boldsymbol{v}_{\boldsymbol{b}} \in \mathbb{R}^m$, $\boldsymbol{v}_{\underline{\boldsymbol{\pi}}} \in \mathbb{R}^n$ *and* $\boldsymbol{v}_{\overline{\boldsymbol{\pi}}} \in \mathbb{R}^n$.

Theorem 1 (Complementary Slackness)**.** *Let* $\boldsymbol{x}$ *be a feasible solution to the linear program in* (5.2) *and let* $\boldsymbol{v} := (\boldsymbol{v}_{\boldsymbol{b}}, \boldsymbol{v}_{\underline{\boldsymbol{\pi}}}, \boldsymbol{v}_{\overline{\boldsymbol{\pi}}})$ *be a feasible solution to its corresponding dual linear program given in* (A.28)*. Then,* $\boldsymbol{x}$ *and* $\boldsymbol{v}$ *are optimal solutions for the two respective problems if and only if*

$$(\boldsymbol{x} - \underline{\boldsymbol{\pi}}) \odot \boldsymbol{v}_{\underline{\boldsymbol{\pi}}} = \boldsymbol{0} \tag{A.29}$$

$$(\boldsymbol{x} - \overline{\boldsymbol{\pi}}) \odot \boldsymbol{v}_{\overline{\boldsymbol{\pi}}} = \boldsymbol{0} \tag{A.30}$$

where $\odot$ *denotes the Hadamard product. The additional conditions* $\boldsymbol{v}_{\boldsymbol{b}} \odot (\boldsymbol{A}^{\intercal}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{0}$ *and* $(\boldsymbol{c} - \boldsymbol{A}^{\intercal}\boldsymbol{v}_{\boldsymbol{b}} - \boldsymbol{v}_{\underline{\boldsymbol{\pi}}} - \boldsymbol{v}_{\overline{\boldsymbol{\pi}}}) \odot \boldsymbol{x} = \boldsymbol{0}$ *are satisfied because both* $\boldsymbol{x}$ *and* $\boldsymbol{v}$ *are feasible.*

Proof. The conditions are derived from the definition of the dual program and strong duality in linear programming [see e.g., 231]. □

B

EXPERIMENTAL DETAILS

B.1 NEURAL RELATIONAL INFERENCE (NRI) FOR GRAPH LAYOUT

B.1.1 *Data*

Our dataset consisted of latent prior spanning trees over 10 vertices. Latent spanning trees were sampled by applying Kruskal's algorithm [32] to $U \sim \text{Gumbel}(0)$ for a fully-connected graph. Note that this does not result in a uniform distribution over spanning trees. Initial vertex locations were sampled from $\text{Normal}(0, I)$ in $\mathbb{R}^2$. Given initial locations and the latent tree, dynamical observations were obtained by applying a force-directed algorithm for graph layout [153] for $T \in \{10, 20\}$ iterations. We then discarded the initial vertex positions, because the first iteration of the layout algorithm typically results in large relocations. This renders the initial vertex positions an outlier which is hard to model. Hence, the final dataset used for training consisted of 10 respectively 20 location observations in $\mathbb{R}^2$ for each of the 10 vertices. By this procedure, we generated a training set of size 50,000 and validation and test sets of size 10,000.

B.1.2 *Model*

The NRI model consists of encoder and decoder graph neural networks. Our encoder and decoder architectures were identical to the MLP encoder and MLP decoder architectures, respectively, in [77].

ENCODER The encoder GNN passes messages over the fully connected directed graph with $n = 10$ nodes. We took the final edge representation of the GNN to use as θ. The final edge representation was in $\mathbb{R}^{90 \times m}$, where $m = 2$ for *Indep. Directed Edges* and $m = 1$ for *E.F. Ent. Top* $|V| - 1$ and *Spanning Tree*, both over undirected edges (90 because we considered all directed edges excluding self-connections). We had $m = 2$ for *Indep. Directed Edges*, because we followed [77] and applied the Gumbel-Max trick independently to each edge. This is equivalent to using $U \sim \text{Logistic}(\theta)$, where $\theta \in \mathbb{R}^{90}$. Both *E.F. Ent. Top* $|V| - 1$ and *Spanning Tree* require undirected

graphs, therefore, we "symmetrized" θ such that $\theta_{ij} = \theta_{ji}$ by taking the average of the edge representations for both directions. Therefore, in this case, $\theta \in \mathbb{R}^{45}$.

DECODER Given previous timestep data, the decoder GNN passes messages over the sampled graph adjacency matrix X and predicts future node positions. As in [77], we used teacher-forcing every 10 timesteps. $X \in \mathbb{R}^{n \times n}$ in this case was a directed adjacency matrix over the graph $G = (V, E)$ where V were the nodes. $X_{ij} = 1$ is interpreted as there being an edge from $i \to j$ and 0 for no edge. For the SMTs over undirected edges (*E.F. Ent. Top* $|V| - 1$ and *Spanning Tree*) X was the symmetric, directed adjacency matrix with edges in both directions for each undirected edge. The decoder passed messages between both connected and not-connected nodes. When considering a message from node $i \to j$, it used one network for the edges with $X_{ij} = 1$ and another network for the edges with $X_{ij} = 0$, such that we could differentiate the two edge "types". For the SST relaxation, both messages were passed, weighted by $(X_t)_{ij}$ and $1 - (X_t)_{ij}$, respectively. Because of the parameterization of our model, during evaluation, it is ambiguous whether the sampled hard graph is in the correct representation (adjacency matrix where 1 is the existence of an edge, and 0 is the non-existence of an edge). Therefore, when measuring precision and recall for structure discovery, we selected whichever graph (the sampled graph versus the graph with adjacency matrix of one minus that of the sampled graph) that yielded the highest precision, and reported precision and recall measurements for that graph.

OBJECTIVE Our ELBO objective consisted of the reconstruction error and KL divergence. The reconstruction error was the Gaussian log likelihood of the predicted node positions generated from the decoder given ground truth node positions. As mentioned in 3.3, we computed the KL divergence with respect to $\boldsymbol{U}$ instead of the sampled graph for all methods, because computing the probability of a *Spanning Tree*, or *Top k* sample is not computationally efficient. We chose our prior to be Gumbel(0). The KL divergence between a Gumbel distribution with location θ and a Gumbel distribution with location 0, is $\boldsymbol{\theta} + \exp(-\boldsymbol{\theta}) - 1$.

B.1.3 *Training*

All graph layout experiments were run with batch size 128 for 50000 steps. We evaluated the model on the validation set every 500 training steps, and saved the model that achieved the best average validation ELBO. We used the Adam optimizer with a constant learning rate, and $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. We tuned hypermarameters using random uniform search over a hypercube-shaped search space with 20 trials. We tuned the constant learning rate, and temperature t for all methods. For *E.F. Ent. Top k*, we additionally tuned ϵ, which is used when computing the gradients for the backward-pass using finite-differences. The ranges for hyperparameter values were chosen such that optimal hyperparameter values (corresponding to the best validation ELBO) were not close to the boundaries of the search space.

B.2 UNSUPERVISED PARSING ON LISTOPS

B.2.1 *Data*

We considered a simplified variant of the ListOps dataset [154]. Specifically, we used the same data generation process as [154] but excluded the `summod` operator and used rejection sampling to ensure that the lengths of all sequences in the dataset ranged only from 10 to 50 and that our dataset contained the same number of sequences of depths $d \in \{1, 2, 3, 4, 5\}$. Depth was measured with respect to the ground truth parse tree. For each sequence, the ground truth parse tree was defined by directed edges from all operators to their respective operands. We generated 100,000 samples for the training set (20,000 for each depth), and 10,000 for the validation and test set (2,000 for each depth).

B.2.2 *Model*

We used an embedding dimension of 60, and all neural networks had 60 hidden units.

LSTM We used a single-layered LSTM going from left to right on the input embedding matrix. The LSTM had hidden size 60 and includes dropout with probability 0.1. The output of the LSTM was flattened and fed into a

single linear layer to bring the dimension to 60. The output of the linear layer was fed into an MLP with one hidden layer and ReLU activations.

GNN ON LATENT (DI)GRAPH Our models had two main parts: an LSTM encoder that produced a graph adjacency matrix sample (X or X_t), and a GNN that passed messages over the sampled graph.

The LSTM encoder consisted of two LSTMs– one representing the "head" tokens, and the other for "modifier" tokens. Both LSTMs were single-layered, left-to-right, with hidden size 60, and include dropout with probability 0.1. Each LSTM outputted a single real valued vector for each token i of n tokens with dimension 60. To obtain $\theta \in \mathbb{R}^{n \times n}$, we defined $\theta_{ij} = v_i^{\text{head}T} v_j^{\text{mod}}$, where v_i^{head} is the vector outputted by the head LSTM for word i and v_i^{mod} is the vector outputted by the modifier LSTM for word j. As in the graph layout experiments, we symmetrized θ for the SSTs that require undirected edges (*Indep. Undirected Edges*, and *Spanning Tree*). For exponential $U \sim \text{Exp}(\theta)$, θ was parameterized as the softplus function of the $\mathbb{R}^{n \times n}$ matrix output of the encoder. We used the Torch-struct library [245] to obtain soft samples for *arborescence*.

$X \in \mathbb{R}^{n \times n}$ in this case was a directed adjacency matrix over the graph $G = (V, E)$ where V were the tokens. $X_{ij} = 1$ is interpreted as there being an edge from $i \to j$ and 0 for no edge. For the SMTs over undirected edges (*Indep. Undirected Edges* and *Spanning Tree*) X was the symmetric, directed adjacency matrix with edges in both directions for each undirected edge. For *Arborescence*, we assumed the first token is the root node of the arborescence.

Given X, the GNN ran 5 message passing steps over the adjacency matrix, with the initial node embeddings being the input embedding. The GNN architecture was identical to the GNN decoder in the graph layout experiments, except we did not pass messages on edges with $X_{ij} = 0$ and we did not include the last MLP after every messaging step. For the SST, we simply weighted each message from $i \to j$ by $(X_t)_{ij}$. We used dropout with probability 0.1 in the MLPs. We used a recurrent connection after every message passing step. The LSTM encoder and the GNN each had their own embedding lookup table for the input. We fed the node embedding of the first token to an MLP with one hidden layer and ReLU activations.

B.2.3 *Training*

All ListOps experiments were run with batch size 100 for 50 epochs. We evaluated the model on the validation set every 800 training steps, and saved the model that achieved the best average validation task accuracy. We used the AdamW optimizer with a constant learning rate, and $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. We tuned hypermarameters using random uniform search over a hypercube-shaped search space with 20 trials. We tuned the constant learning rate, temperature t, and weight decay for all methods. The ranges for hyperparameter values were chosen such that optimal hyperparameter values (corresponding to the best validation accuracy) were not close to the boundaries of the search space.

B.3 LEARNING TO EXPLAIN (L2X) ASPECT RATINGS

B.3.1 *Data*

We used the BeerAdvocate dataset [155], which contains reviews comprised of free-text feedback and ratings for multiple aspects, including appearance, aroma, palate, and taste. For each aspect, we used the same de-correlated subsets of the original dataset as [156]. The training set for the aspect appearance contained 80k reviews and for all other aspects 70k reviews. Unfortunately, [156] do not provide separate validation and test sets. Therefore for each aspect, we split their heldout set into two evenly sized validation and test sets containing 5k reviews each. We used pre-trained word embeddings of dimension 200 from [156] to initialize all models. Each review was padded/ cut to 350 words. For all aspects, subset precision was measured on the same subset of 993 annotated reviews from [155]. The aspect ratings were normalized to the unit interval $[0, 1]$ and MSE is reported on the normalized scale.

B.3.2 *Model*

Our model used convolutional neural networks to parameterize both the subset distribution and to make a prediction from the masked embeddings. For parameterizing the masks, we considered a simple and (a more) complex architecture. The simple architecture consisted of a Dropout layer (with $p = 0.1$) and a convolutional layer (with one filter and a kernel size of one) to parameterize $\theta_i \in \mathbb{R}$ for each word i, producing a the vector $\theta \in \mathbb{R}^n$. For

Corr. Top k, $\theta \in \mathbb{R}^{2n-1}$. The first n dimensions correspond to each word i and are parameterized as above. For dimensions $i \in \{n+1, \ldots, 2n-1\}$, θ_i represents a coupling between words i and $i+1$, so we denote this $\theta_{i,i+1}$. It was parameterized as the sum of three terms, $\theta_{i,i+1} = \phi_i + \phi'_i + \phi_{i,i+1}$: $\phi_i \in \mathbb{R}$ computed using a seperate convolutional layer of the same kind as described above, $\phi'_i \in \mathbb{R}$ computed using yet another convolutional layer of the same kind as described above, and $\phi_{i,i+1} \in \mathbb{R}$ obtained from a third convolutional layer with one filter and a kernel size of two. In total, we used four separate convolutional layers to parameterize the simple encoder. For the complex architecture, we used two additional convolutional layers, each with 100 filters, kernels of size three, ReLU activations to compute the initial word embeddings. This was padded to maintain the length of a review.

X was a k-hot binary vector in n-dimensions with each dimension corresponding to a word. For *Corr. Top K*, we ignored dimensions $i \in \{n+1, \ldots, 2n-1\}$, which correspond to the pairwise indicators. Predictions were made from the masked embeddings, using X_i to mask the embedding of word i. Our model applied a soft (at training) or hard (at evaluation) subset mask to the word embeddings of a given review. Our model then used two convolutional layers over these masked embeddings, each with 100 filters, kernels of size three and ReLU activations. The resulting output was max-pooled over all feature vectors. Our model then made predictions using a Dropout layer (with $p = 0.1$), a fully connected layer (with output dimension 100, ReLU activation) and a fully connected layer (with output dimension one, sigmoid activation) to predict the rating of a given aspect.

B.3.3 *Training.*

We trained all models for ten epochs at minibatches of size 100. We used the Adam optimizer [246] and a linear learning rate schedule. Hyperparameters included the initial learning rate, its final decay factor, the number of epochs over which to decay the learning rate, weight decay and the temperature of the relaxed gradient estimator. Hyperparameters were optimized for each model using random search over 25 independent runs. The learning rate and its associated hyperparameters were sampled from $\{1, 3, 5, 10, 30, 50, 100\} \times 10^{-4}$, $\{1, 10, 100, 1000\} \times 10^{-4}$ and $\{5, 6, \ldots, 10\}$ respectively. Weight decay was sampled from $\{0, 1, 10, 100\} \times 10^{-6}$ and the temperature was sampled from $[0.1, 2]$. For a given run, models were evaluated on the validation set

at the end of each epoch and the best validated model was was retained. For *E.F. Ent. Top k* and *Corr. Top k*, we trained these methods with $\epsilon \in \{1, 10, 100, 1000\} \times 10^{-3}$ and selected the best ϵ on the validation set. We believe that it may be possible to improve on the results we report with an efficient exact implementation of the backward pass for these two methods. We found the overhead created by automatic differentiation software to differentiate through the unrolled dynamic program was prohibitively large in this experiment.

B.4 GENERATIVE MODELLING WITH VARIATIONAL AUTOENCODERS

We trained variational autoencoders with n-ary discrete random variables with values on the corners of the hypercube $\{-1, 1\}^{\log_2(n)}$. The model with arity $\{2, 4, 8, 16\}$ included $\{240, 120, 80, 60\}$ random variables respectively.

All models were optimized using stochastic gradient descent with momentum for 50000 steps on minibatches of size 20 and 200 respectively. Hyperparameters were randomly sampled and the best setting was selected from twenty independent runs. Learning rate and momentum were randomly sampled from $\{5, 6, \ldots 50\} \times 10^{-4}$ and $(0, 1)$ respectively. We did not anneal the learning rate during training. For regularising the network, we used weight-decay, which was randomly sampled from $\{0, 10^{-1}, 10^{-2} \ldots, 10^{-6}\}$. The temperature was randomly sampled from $[0.1, 1.0]$ and not annealed throughout training.

All models were evaluated on the validation and test set using the importance-weighted bound on the log-likelihood described by Burda, Grosse & Salakhutdinov [173] with 5000 samples.

To estimate the variance of a gradient estimator in the VAE experiment we used 5000 randomly sampled minibatches of size 20, for each of which we performed 100 independent forward passes and then computed the associated gradient for the parameters of the inference network. We then summed the variance to get a singe scalar measurement.

To estimate the bias of a gradient estimator in the VAE experiment, we proceeded as above to approximate the expectation for a gradient estimator. We approximated the true gradient by following this procedure for the REINFORCE algorithm.

To assess training speed, we measured the average number of iterations needed to achieve a prespecified loss threshold on the validation set. In particular, we ran multiple independent runs under the same experimental conditions for all gradient estimators. Among only runs that achieved

the threshold within the total budget, we report the average number of iterations taken to cross the threshold.

B.5 LEARNING TO CUT IN BRANCH AND BOUND

B.5.1 *Data*

B.5.1.1 *Features*

B.6 LEARNING TO DIVE IN BRANCH AND BOUND

B.6.1 *Data*

B.6.1.1 *Features*

B.6.2 *Data*

B.6.2.1 *Features*

B.6.3 *Baselines*

We leveraged expert knowledge and used random search to optimize the use of diving heuristics in SCIP 7.0.2 for our baseline *Tuned*. The most important parameters to control the standard divers in SCIP are *freq* and *freqofs*. For each diving heuristic, these parameters control the depth at which the heuristic may be called or not called. By varying these parameters, diverse diving ensembles can be realized that call different heuristics at different stages of the branch and bound search. We randomly sample solver configurations by setting either $freq = -1$ (no diving), or $freq = \lfloor 0.5 \times freq_{\text{default}} \rfloor$ (double frequency) or $freq = freq_{\text{default}}$ (leave frequency at default) or $freq = \lfloor 2 \times freq_{\text{default}} \rfloor$ (halve frequency) with equal probability and setting either $freqofs = 0$ or $freqofs = freqofs_{\text{default}}$ with equal probability independently for each diving heuristic. We run the solver with the usual time limits for each configuration and each validation instance and pick the configuration with the lowest primal-dual integral for server load balancing and with the lowest solving time for neural network verification. For server load balancing, since the original validation dataset was relatively small, we created a new validation dataset of 625 instances from the original training and validation sets. We optimized over 16 random configurations and thus used a budget of 10,000 solver calls for load balancing. For neural network verification, we used the original validation dataset

of 505 validation instances. We considered six random configurations and thus used a budget of 3,030 solver calls which is slightly more than what *L2Dive* used for data collection (2903).

We did not tune any parameters to optimize the use of *L2Dive*, but this might improve performance.

Nodes	Feature	Description
Variables	norm_coef	Objective coefficient, normalized by objective norm
	type	Type (binary, integer, impl. integer, continuous) one-hot
	has_lb	Lower bound indicator
	has_ub	Upper bound indicator
	norm_redcost	Reduced cost, normalized by objective norm
	solval	Solution value
	solfrac	Solution value fractionality
	sol_is_at_lb	Solution value equals lower bound
	sol_is_at_ub	Solution value equals upper bound
	norm_age	LP age, normalized by total number of solved LPs
	basestat	Simplex basis status (lower, basic, upper, zero) one-hot
Constraints, Cuts	is_cut	Indicator to differentiate cut vs. constraint
	type	Separator type, one-hot
	rank	Rank of a row
	norm_nnzrs	Fraction of nonzero entries
	bias	Unshifted side normalized by row norm
	row_is_at_lhs	Row value equals left hand side
	row_is_at_rhs	Row value equals right hand side
	dualsol	Dual LP solution of a row, normalized by row and objective norm
	basestat	Basis status of a row in the LP solution, one-hot
	norm_age	Age of row, normalized by total number of solved LPs
	norm_nlp_creation	LPs since the row has been created, normalized
	norm_intcols	Fraction of integral columns in the row
	is_integral	Activity of the row is always integral in a feasible solution
	is_removable	Row is removable from the LP
	is_in_lp	Row is member of current LP
	violation	Violation score of a row
	rel_violation	Relative violation score of a row
	obj_par	Objective parallelism score of a row
	exp_improv	Expected improvement score of a row
	supp_score	Support score of a row
	int_support	Integral support score of a row
	scip_score	SCIP score of a row for cut selection

TABLE B.1: We use these features in our experiments with *NeuralCut* in chapter 6.

Nodes	Feature	Description
Variables	norm_coef	Objective coefficient, normalized by objective norm
	type	Type (binary, integer, impl. integer, continuous) one-hot
	has_lb	Lower bound indicator
	has_ub	Upper bound indicator
	norm_redcost	Reduced cost, normalized by objective norm
	solval	Solution value
	solfrac	Solution value fractionality
	sol_is_at_lb	Solution value equals lower bound
	sol_is_at_ub	Solution value equals upper bound
	norm_age	LP age, normalized by total number of solved LPs
	basestat	Simplex basis status (lower, basic, upper, zero) one-hot
Constraints, Cuts	is_cut	Indicator to differentiate cut vs. constraint
	rank	Rank of a row
	norm_nnzrs	Fraction of nonzero entries
	bias	Unshifted side normalized by row norm
	row_is_at_lhs	Row value equals left hand side
	row_is_at_rhs	Row value equals right hand side
	dualsol	Dual LP solution of a row, normalized by row and objective norm
	basestat	Basis status of a row in the LP solution, one-hot
	norm_age	Age of row, normalized by total number of solved LPs
	norm_nlp_creation	LPs since the row has been created, normalized
	norm_intcols	Fraction of integral columns in the row
	is_integral	Activity of the row is always integral in a feasible solution
	is_removable	Row is removable from the LP
	is_in_lp	Row is member of current LP

TABLE B.2: We use these features in our experiments with *L2Dive* in chapter 7.

C

ADDITIONAL RESULTS

C.1 GRADIENT ESTIMATION WITH STOCHASTIC SOFTMAX TRICKS

C.1.1 *Neural Relational Inference with the Score Function*

We experimented with 3 variants of REINFORCE estimators, each with a different baseline. The *EMA* baseline is an exponential moving average of the ELBO. The *Batch* baseline is the mean ELBO of the current batch. Finally, the *Multi-sample* baseline is the mean ELBO over k multiple samples, which is a local baseline for each sample (See section 3.1 of [247]). For NVIL, the input-dependent baseline was a one hidden-layer MLP with ReLU activations, attached to the GNN encoder, just before the final fully connected layer. We did not do variance normalization. We used weight decay on the encoder parameters, including the input-dependent baseline parameters. We tuned weight decay and the exponential moving average constant, in addition to the learning rate. For *Multi-sample* REINFORCE, we additionally tuned $k = \{2, 4, 8\}$, and following [247], we divided the batch size by k in order to keep the number of total samples constant.

We used $\boldsymbol{U}$ as the "action" for all edge distributions, and therefore, computed the log probability over $\boldsymbol{U}$. We also computed the KL divergence with respect to $\boldsymbol{U}$ as in the rest of the graph layout experiments. This was because computing the probability of $\boldsymbol{X}$ is not computationally efficient for *Top* $|V| - 1$ and *Spanning Tree*. In particular, the marginal of $\boldsymbol{X}$ in these cases is not in the exponential family. We emphasize that using $\boldsymbol{U}$ as the "action" for REINFORCE is atypical.

We found that both NVIL and REINFORCE with *Indep. Directed Edges* and *Top* $|V| - 1$ perform similarly to their SST counterparts, struggling to learn the underlying structure. This is also the case for REINFORCE with *Spanning Tree*. On the other hand, NVIL with *Spanning Tree*, is able to learn some structure, although worse and higher variance than its SST counterpart.

	REINFORCE (*EMA*)			NVIL		
Edge Distribution	ELBO	Edge Prec.	Edge Rec.	ELBO	Edge Prec.	Edge Rec.
Indep. Directed Edges	-1730 ± 60	41 ± 4	92 ± 7	-1550 ± 20	44 ± 1	94 ± 1
Top $\lvert V \rvert - 1$	-2170 ± 10	42 ± 1	42 ± 1	-2110 ± 10	42 ± 2	42 ± 2
Spanning Tree	-2250 ± 20	40 ± 7	40 ± 7	-1570 ± 300 I	83 ± 20	83 ± 20
	REINFORCE (*Batch*)			REINFORCE (*Multi-sample*)		
Edge Distribution	ELBO	Edge Prec.	Edge Rec.	ELBO	Edge Prec.	Edge Rec.
Indep. Directed Edges	-1780 ± 20	39 ± 3	90 ± 6	-1710 ± 30	38 ± 3	88 ± 6
Top $\lvert V \rvert - 1$	-2180 ± 0	39 ± 1	39 ± 1	-2150 ± 10	40 ± 0	40 ± 0
Spanning Tree	-2260 ± 0	41 ± 1	41 ± 1	-2230 ± 20	42 ± 1	42 ± 1

TABLE C.1: NVIL and REINFORCE struggle to learn the underlying structure wherever their SST counterparts struggle. NVIL with *Spanning Tree* is able to learn some structure, but it is still worse and higher variance than its SST counterpart. This is for $T = 10$.

C.1.2 *Learning To Explain Other Aspect Ratings*

C.2 GUMBEL-RAO

C.3 LEARNING TO CUT IN BRANCH AND BOUND

C.3.1 *Model Ablation*

In Table C.6, we perform model ablations to better assess the effectiveness of our modelling choices. We begin with a simple GNN model that is based on Gasse *et al.* [200], but adapted to cut selection by using cuts in place of constraints in the original model formulation of Gasse *et al.* [200] and switching the order in which half-convolutions are applied. Second, we use the features of NeuralCut (+ our features) for both cuts and variables instead of the features of Gasse *et al.* [200]. Third, we additionally make architectural choices, i.e. we replace layer normalization with batch normalization and add the attention module to the model. Finally, we leverage the tri-partite graph formulation to model constraints explicitly in addition to the cuts and variables, which gives the full NeuralCut model.

Model	Relaxation	$k = 5$		$k = 10$		$k = 15$	
		MSE	Subs. Prec.	MSE	Subs. Prec.	MSE	Subs. Prec.
Simple	*L2X* [63]	3.1 ± 0.1	48.7 ± 0.6	2.6 ± 0.1	41.9 ± 0.6	2.5 ± 0.1	38.6 ± 1.5
	SoftSub [131]	3.2 ± 0.1	43.9 ± 1.1	2.7 ± 0.1	41.9 ± 2.1	2.5 ± 0.1	38.0 ± 2.4
	Euclid. Top k	3.0 ± 0.1	49.4 ± 1.7	2.6 ± 0.1	48.8 ± 1.2	2.4 ± 0.1	42.9 ± 1.0
	Cat. Ent. Top k	3.0 ± 0.1	53.2 ± 1.7	2.6 ± 0.1	46.3 ± 1.9	2.4 ± 0.1	41.3 ± 0.8
	Bin. Ent. Top k	3.0 ± 0.1	54.5 ± 5.6	2.6 ± 0.1	48.9 ± 1.7	2.4 ± 0.1	43.1 ± 0.6
	E.F. Ent. Top k	3.0 ± 0.1	53.2 ± 0.9	2.5 ± 0.1	50.6 ± 2.1	2.4 ± 0.1	43.3 ± 0.3
	Corr. Top k	$\mathbf{2.7 \pm 0.1}$	$\mathbf{71.6 \pm 1.1}$	$\mathbf{2.4 \pm 0.1}$	$\mathbf{69.7 \pm 1.7}$	$\mathbf{2.3 \pm 0.1}$	$\mathbf{66.7 \pm 1.7}$
Complex	*L2X* [63]	2.6 ± 0.1	76.6 ± 0.4	2.4 ± 0.1	69.3 ± 0.9	2.4 ± 0.1	62.6 ± 3.0
	SoftSub [131]	2.6 ± 0.1	79.4 ± 1.1	2.5 ± 0.1	69.5 ± 2.0	2.4 ± 0.1	60.2 ± 7.0
	Euclid. Top k	2.6 ± 0.1	81.6 ± 0.9	2.4 ± 0.1	76.9 ± 1.7	2.3 ± 0.1	69.7 ± 2.2
	Cat. Ent. Top k	$\mathbf{2.5 \pm 0.1}$	83.7 ± 0.8	2.4 ± 0.1	76.5 ± 0.9	2.2 ± 0.1	65.9 ± 1.4
	Bin. Ent. Top k	2.6 ± 0.1	81.9 ± 0.7	2.4 ± 0.1	75.7 ± 1.2	$\mathbf{2.2 \pm 0.1}$	65.7 ± 1.1
	E.F. Ent. Top k	2.6 ± 0.1	82.3 ± 1.4	2.4 ± 0.1	72.9 ± 0.7	2.3 ± 0.1	65.8 ± 1.3
	Corr. Top k	$\mathbf{2.5 \pm 0.1}$	$\mathbf{85.1 \pm 2.4}$	$\mathbf{2.3 \pm 0.1}$	$\mathbf{77.8 \pm 1.3}$	$\mathbf{2.2 \pm 0.1}$	$\mathbf{74.5 \pm 1.5}$

TABLE C.2: For k-subset selection on appearance aspect, SSTs select subsets with high precision and outperform baseline relaxations. Test set MSE ($\times 10^{-2}$) and subset precision (%) is shown for models selected on valid. MSE.

C.3.2 *Loss Ablation*

In Table C.7, we experimented with different loss functions on the bound fulfillment surrogate. In addition to the binary entropy loss (which was used to train all NeuralCut models), we explored the effectiveness of mean squared error (MSE), both with linear and sigmoid activations as well as a cross entropy loss function.

C.3.3 *Visuals for Table 6.3*

In Figure C.3, we visualize the IGC trajectories of Lookahead, NeuralCut and common heuristics for cut selection on the four benchmarks Maximum Cut, Packing, Binary Packing and Planning. The NeuralCut models correspond to those in Table 6.3. All models and heuristics were evaluated on the 500 test instances of the respective domain. 30 separation rounds were performed in each a single cut was added.

		$k = 5$		$k = 10$		$k = 15$	
Model	Relaxation	MSE	Subs. Prec.	MSE	Subs. Prec.	MSE	Subs. Prec.
Simple	*L2X* [63]	3.5 ± 0.1	27.8 ± 3.7	3.2 ± 0.1	21.0 ± 1.8	3.0 ± 0.1	20.5 ± 0.7
	SoftSub [131]	3.7 ± 0.1	23.9 ± 1.4	3.3 ± 0.1	23.5 ± 3.7	3.1 ± 0.1	20.0 ± 1.7
	Euclid. Top k	3.5 ± 0.1	36.0 ± 5.7	3.2 ± 0.1	27.1 ± 0.7	3.0 ± 0.1	23.7 ± 0.8
	Cat. Ent. Top k	3.6 ± 0.1	25.4 ± 3.6	3.0 ± 0.1	28.5 ± 2.9	3.0 ± 0.1	21.7 ± 0.4
	Bin. Ent. Top k	3.6 ± 0.1	25.2 ± 1.7	3.2 ± 0.1	27.2 ± 2.6	3.0 ± 0.1	23.4 ± 1.7
	E.F. Ent. Top k	3.6 ± 0.1	26.0 ± 3.0	3.1 ± 0.1	27.0 ± 1.6	2.9 ± 0.1	23.4 ± 0.6
	Corr. Top k	$\mathbf{3.2 \pm 0.1}$	$\mathbf{54.3 \pm 1.0}$	$\mathbf{2.8 \pm 0.1}$	$\mathbf{50.0 \pm 1.7}$	$\mathbf{2.7 \pm 0.1}$	$\mathbf{46.0 \pm 2.0}$
Complex	*L2X* [63]	3.1 ± 0.1	47.4 ± 1.7	2.8 ± 0.1	40.8 ± 0.6	2.7 ± 0.1	34.8 ± 0.8
	SoftSub [131]	3.1 ± 0.1	44.4 ± 1.1	2.8 ± 0.1	44.2 ± 2.0	2.8 ± 0.1	38.7 ± 1.0
	Euclid. Top k	2.9 ± 0.1	56.2 ± 0.7	2.7 ± 0.1	43.9 ± 1.7	$\mathbf{2.6 \pm 0.1}$	38.0 ± 1.1
	Cat. Ent. Top k	2.9 ± 0.1	55.1 ± 0.7	2.7 ± 0.1	45.2 ± 0.8	$\mathbf{2.6 \pm 0.1}$	40.2 ± 0.9
	Bin. Ent. Top k	2.9 ± 0.1	55.6 ± 0.8	2.7 ± 0.1	47.6 ± 1.0	2.7 ± 0.1	39.1 ± 1.0
	E.F. Ent. Top k	2.9 ± 0.1	56.3 ± 0.3	2.7 ± 0.1	48.1 ± 1.3	$\mathbf{2.6 \pm 0.1}$	40.3 ± 1.0
	Corr. Top k	$\mathbf{2.8 \pm 0.1}$	$\mathbf{60.4 \pm 1.5}$	$\mathbf{2.6 \pm 0.1}$	$\mathbf{53.5 \pm 2.9}$	$\mathbf{2.6 \pm 0.1}$	$\mathbf{46.8 \pm 1.5}$

TABLE C.3: For k-subset selection on palate aspect, SSTs tend to outperform baseline relaxations. Test set MSE ($\times 10^{-2}$) and subset precision (%) is shown for models selected on valid. MSE.

Model	Relaxation	$k = 5$		$k = 10$		$k = 15$	
		MSE	Subs. Prec.	MSE	Subs. Prec.	MSE	Subs. Prec.
Simple	*L2X* [63]	3.1 ± 0.1	28.5 ± 0.6	2.9 ± 0.1	24.1 ± 1.3	2.7 ± 0.1	26.8 ± 0.8
	SoftSub [131]	3.1 ± 0.1	29.9 ± 0.8	2.9 ± 0.1	27.7 ± 0.7	2.7 ± 0.1	27.8 ± 1.9
	Euclid. Top k	3.0 ± 0.1	30.2 ± 0.4	2.7 ± 0.1	28.0 ± 0.4	2.6 ± 0.1	26.5 ± 0.5
	Cat. Ent. Top k	3.1 ± 0.1	28.5 ± 0.6	2.8 ± 0.1	28.9 ± 0.6	2.6 ± 0.1	30.5 ± 1.6
	Bin. Ent. Top k	3.0 ± 0.1	29.2 ± 0.4	2.9 ± 0.1	24.6 ± 1.7	2.6 ± 0.1	27.9 ± 0.9
	E.F. Ent. Top k	3.0 ± 0.1	29.7 ± 0.3	2.7 ± 0.1	29.0 ± 1.5	2.6 ± 0.1	26.5 ± 0.5
	Corr. Top k	$\mathbf{2.8 \pm 0.1}$	$\mathbf{31.7 \pm 0.5}$	$\mathbf{2.5 \pm 0.1}$	$\mathbf{37.7 \pm 1.6}$	$\mathbf{2.4 \pm 0.1}$	$\mathbf{37.8 \pm 0.5}$
Complex	*L2X* [63]	2.5 ± 0.1	40.3 ± 0.7	2.4 ± 0.1	42.4 ± 2.0	2.4 ± 0.1	39.7 ± 1.1
	SoftSub [131]	2.5 ± 0.1	43.3 ± 0.9	2.4 ± 0.1	41.3 ± 0.5	2.3 ± 0.1	40.5 ± 0.7
	Euclid. Top k	$\mathbf{2.4 \pm 0.1}$	43.8 ± 0.7	2.3 ± 0.1	43.1 ± 0.6	2.2 ± 0.1	42.2 ± 0.8
	Cat. Ent. Top k	$\mathbf{2.4 \pm 0.1}$	$\mathbf{46.5 \pm 0.6}$	2.3 ± 0.1	44.6 ± 0.3	2.2 ± 0.1	45.5 ± 1.1
	Bin. Ent. Top k	$\mathbf{2.4 \pm 0.1}$	40.9 ± 1.3	2.3 ± 0.1	46.3 ± 0.9	2.2 ± 0.1	44.7 ± 0.5
	E.F. Ent. Top k	$\mathbf{2.4 \pm 0.1}$	45.3 ± 0.6	$\mathbf{2.2 \pm 0.1}$	46.1 ± 0.8	2.2 ± 0.1	$\mathbf{46.6 \pm 1.1}$
	Corr. Top k	$\mathbf{2.4 \pm 0.1}$	45.9 ± 1.3	$\mathbf{2.2 \pm 0.1}$	$\mathbf{47.3 \pm 0.6}$	$\mathbf{2.1 \pm 0.1}$	45.1 ± 2.0

TABLE C.4: For k-subset selection on taste aspect, MSE and subset precision tend to be lower for all methods. This is because the taste rating is highly correlated with other ratings making it difficult to identify subsets with high precision. SSTs achieve small improvements. Test set MSE ($\times 10^{-2}$) and subset precision (%) is shown for models selected on valid. MSE.

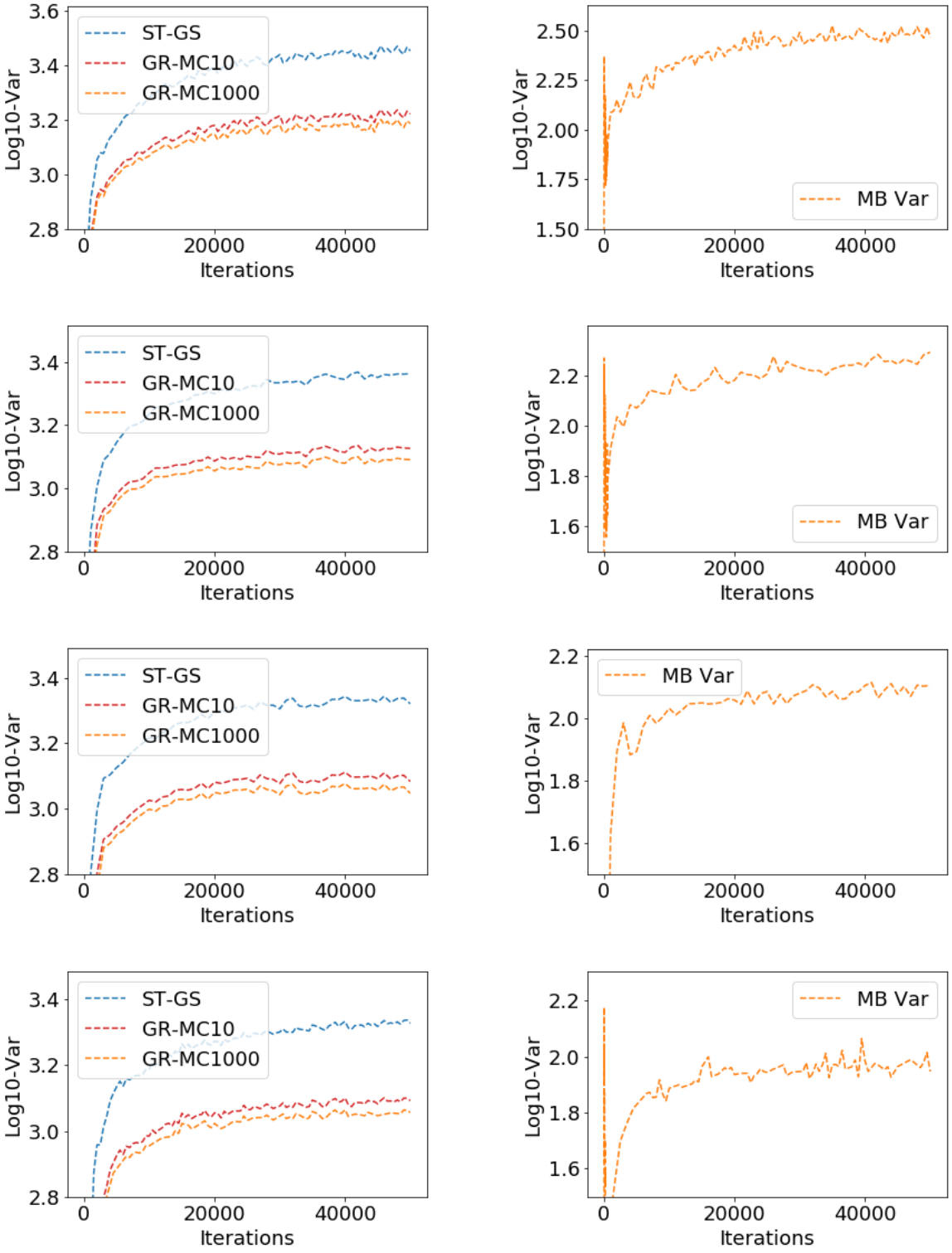


FIGURE C.1: Our estimator (GR-MC*K*) effectively reduces the variance over the entire training trajectory at all arities. The variance reduction compares favorable to the minibatch variance. Columns correspond to arities, i.e. (a) binary, (b) 4-ary, (c) 8-ary, (d) 16-ary. First row, log10-trace of MC covariance matrix for various gradient estimators over iterations. Second row, log10-trace of MB covariance matrix over iterations (same for all gradient estimators).

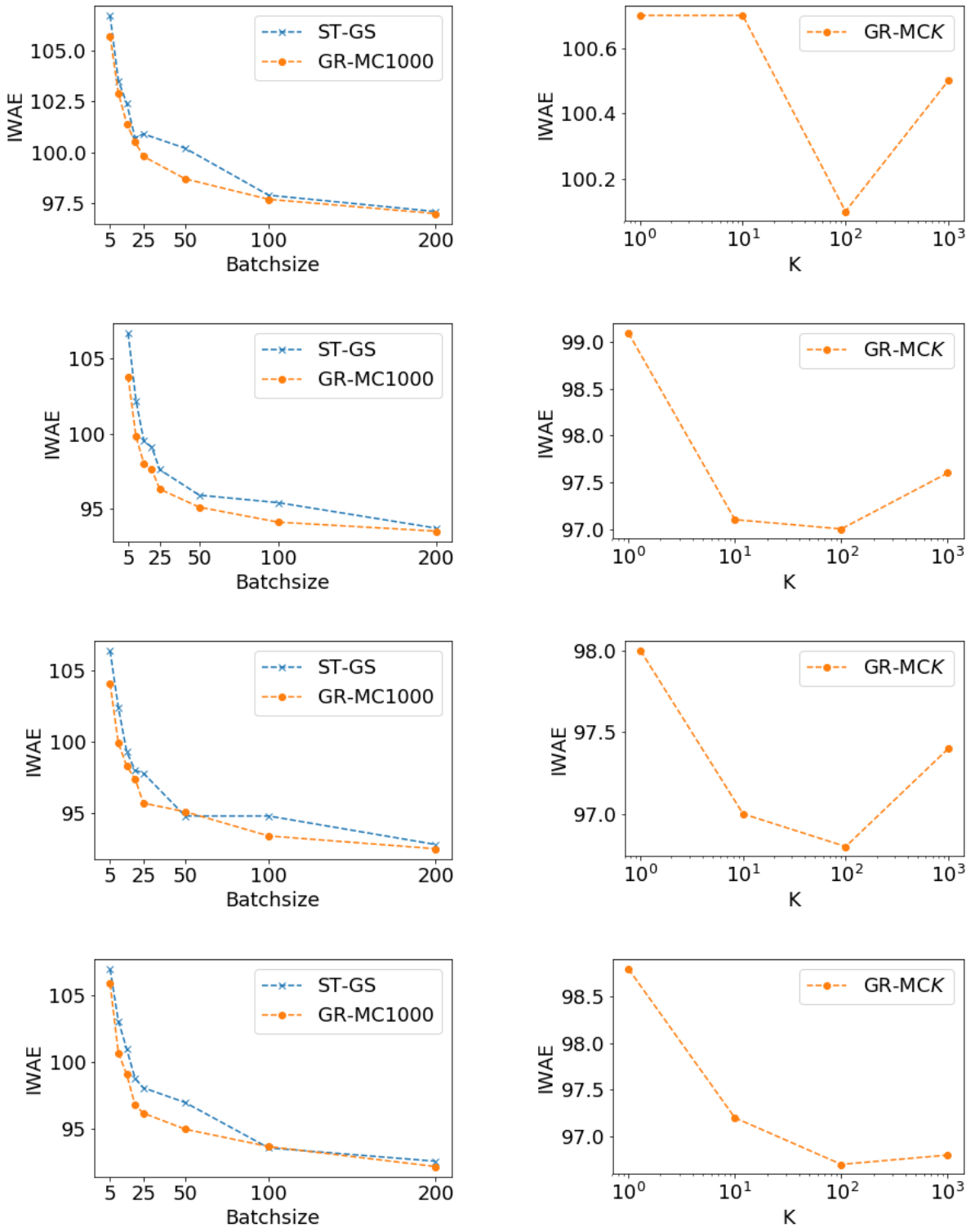


FIGURE C.2: Increasing the number of Monte Carlo samples K to reduce variance in gradient estimation tends to improve performance. The performance difference tends to be larger at smaller batch sizes. Row correspond to arities, i.e. binary, 4-ary, 8-ary, 16-ary. First column, IWAE on test set for best validated model trained at various batch sizes. Second column, IWAE on test set for best validated model trained at various K at batch size 20.

		binary		4-ary		8-ary		16-ary	
	Estimator	Valid.	Test	Valid.	Test	Valid.	Test	Valid.	Test
batch-size 5	ST-GS	107.7	106.7	107.8	106.7	107.5	106.4	108.1	107.0
	GR-MC1000	**106.7**	**105.7**	**104.7**	**103.8**	**105.1**	**104.1**	**107.0**	**105.9**
batch-size 10	ST-GS	104.4	103.5	103.2	102.2	103.5	102.4	104.1	103.1
	GR-MC1000	**103.7**	**102.9**	**100.8**	**99.8**	**100.9**	**99.9**	**101.8**	**100.7**
batch-size 15	ST-GS	103.4	102.4	100.4	99.5	100.3	99.3	101.9	101.0
	GR-MC1000	**102.3**	**101.4**	**99.0**	**98.0**	**99.2**	**98.3**	**100.2**	**99.1**
batch-size 20	ST-GS	101.5	100.7	100.0	99.1	99.0	98.0	99.8	98.8
	GR-MC1000	**101.3**	**100.5**	**98.4**	**97.6**	**97.5**	**96.5**	**97.8**	**96.8**
batch-size 25	ST-GS	101.7	100.9	98.6	97.6	98.8	97.8	99.0	98.1
	GR-MC1000	**100.7**	**99.8**	**97.2**	**96.3**	**96.6**	**95.7**	**97.1**	**96.2**
batch-size 50	ST-GS	101.2	100.2	96.7	95.9	**95.7**	**94.8**	98.0	97.0
	GR-MC1000	**99.5**	**98.7**	**96.0**	**95.1**	95.9	95.1	**95.9**	**95.0**
batch-size 100	ST-GS	98.8	97.9	96.3	95.4	95.7	94.8	**94.4**	**93.6**
	GR-MC1000	**98.5**	**97.7**	**95.0**	**94.1**	**94.3**	**93.4**	94.6	93.7
batch-size 200	ST-GS	97.9	97.1	94.5	93.7	93.6	92.8	93.4	92.6
	GR-MC1000	**97.8**	**97.0**	**94.3**	**93.5**	**93.2**	**92.5**	**93.1**	**92.2**

TABLE C.5: Our estimator, GR-MC*K*, consistently achieves better performance across arities and batchsizes. The outperformance tends to be larger at smaller batchsizes. Best bound on the negative log-likelihood selected on the validation set from 20 independent runs at randomly searched hyperparameters.

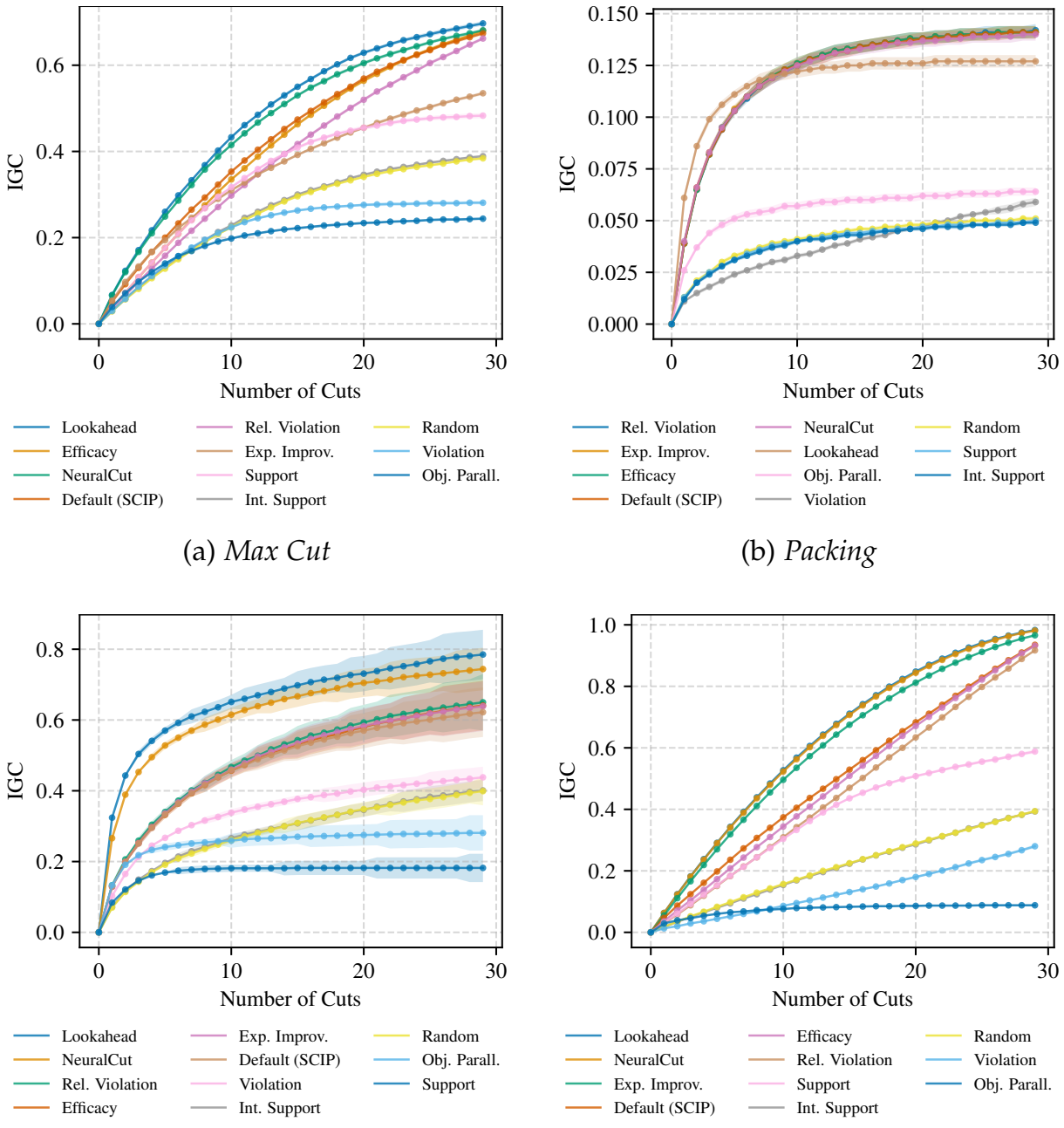
	Bound fulfillment (↑) on test samples, mean			
	Max. Cut	Packing	Bin. Packing	Planning
Gasse *et al.* [200]	0.90	0.42	0.54	0.95
+ our features	0.93	0.26	0.66	**0.99**
+ our architecture	0.93	**0.62**	0.76	**1.00**
NeuralCut (+ our graph)	**0.96**	**0.61**	**0.78**	**1.00**

	Reversed IGC integral (↓) on test instances, mean (ste)			
	Max. Cut	Packing	Bin. Packing	Planning
Gasse *et al.* [200]	16.68 (0.10)	27.30 (0.07)	16.09 (0.31)	11.26 (0.06)
+ our features	**15.54 (0.09)**	28.54 (0.04)	12.77 (0.32)	10.66 (0.04)
+ our architecture	**15.72 (0.09)**	**26.44 (0.07)**	**10.73 (0.33)**	10.55 (0.04)
NeuralCut (+ our graph)	**15.55 (0.09)**	**26.30 (0.08)**	**10.96 (0.33)**	**10.42 (0.04)**

TABLE C.6: Our model choices tend to improve both bound fulfillment (left) and test reversed IGC integral (right) on all four benchmarks. The improvements tend to be clearer for bound fulfillment and binary packing. Gasse *et al.* [200] is the model and features described in [200] and adapted to cut selection by replacing constraints with cuts in the bipartite graph. For the purpose of this ablation, this base model is first augmented with the features of *NeuralCut*. Second, structural changes are made to the model by adding the attention module and using batch normalization. Finally, constraints are explicitly modelled in the tripartite graph giving the full *NeuralCut* model.

	Bound fulfillment (↑) on test samples, mean			
	Max. Cut	Packing	Bin. Packing	Planning
Binary Ent. (*NeuralCut*)	**0.96**	**0.61**	**0.78**	**1.00**
MSE (linear act.)	**0.96**	**0.60**	0.70	**0.99**
MSE (sigmoid act.)	**0.97**	**0.61**	0.69	**1.00**
Cross Entropy	**0.96**	0.30	0.73	**1.00**
	Reversed IGC integral (↓) on test instances, mean (ste)			
	Max. Cut	Packing	Bin. Packing	Planning
Binary Ent. (*NeuralCut*)	**15.55 (0.09)**	**26.30 (0.08)**	**10.96 (0.33)**	**10.42 (0.04)**
MSE (linear act.)	**15.59 (0.09)**	**26.37 (0.08)**	11.66 (0.33)	10.54 (0.05)
MSE (sigmoid act.)	**15.63 (0.09)**	**26.30 (0.08)**	12.65 (0.32)	**10.47 (0.04)**
Cross Entropy	**15.71 (0.09)**	28.73 (0.04)	**11.36 (0.34)**	10.59 (0.04)

TABLE C.7: Different choices for the training objective (where bound fulfillment is used as surrogate) tend to produce models with comparable test performance. Notable exception are binary packing and the choice of cross entropy for packing. Binary entropy (as was used to train *NeuralCut*) tends to be a good choice across all four benchmarks. Best are bold-faced.



(a) *Max Cut* (b) *Packing*

(c) *Binary Packing* (d) *Planning*

FIGURE C.3: Mean test IGC curves for *NeuralCut* models trained on *Max Cut*, *Packing*, *Binary Packing* and *Planning*.

BIBLIOGRAPHY

1. Redmon, J., Divvala, S., Girshick, R. & Farhadi, A. *You only look once: Unified, real-time object detection* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 779.
2. Girshick, R. *Fast r-cnn* in *Proceedings of the IEEE international conference on computer vision* (2015), 1440.
3. Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A. & Zagoruyko, S. *End-to-end object detection with transformers* in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16* (2020), 213.
4. Wang, N. & Yeung, D.-Y. Learning a deep compact image representation for visual tracking. *Advances in neural information processing systems* **26** (2013).
5. Song, Y., Ma, C., Wu, X., Gong, L., Bao, L., Zuo, W., Shen, C., Lau, R. W. & Yang, M.-H. *Vital: Visual tracking via adversarial learning* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 8990.
6. Long, J., Shelhamer, E. & Darrell, T. *Fully convolutional networks for semantic segmentation* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), 3431.
7. Noh, H., Hong, S. & Han, B. *Learning deconvolution network for semantic segmentation* in *Proceedings of the IEEE international conference on computer vision* (2015), 1520.
8. Dong, C., Loy, C. C., He, K. & Tang, X. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence* **38**, 295 (2015).
9. Zhang, L. & Agrawala, M. Adding Conditional Control to Text-to-Image Diffusion Models. *arXiv preprint arXiv:2302.05543* (2023).
10. Ramesh, A., Dhariwal, P., Nichol, A., Chu, C. & Chen, M. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125* (2022).
11. Rombach, R., Blattmann, A., Lorenz, D., Esser, P. & Ommer, B. *High-Resolution Image Synthesis with Latent Diffusion Models* 2021.

12. Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y. & Potts, C. *Recursive deep models for semantic compositionality over a sentiment treebank* in *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), 1631.

13. Williams, A., Nangia, N. & Bowman, S. R. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426* (2017).

14. Levesque, H. J., Davis, E. & Morgenstern, L. The Winograd schema challenge. *KR* **2012**, 13th (2012).

15. Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M. & Le, Q. V. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

16. Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D. & Christiano, P. F. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* **33**, 3008 (2020).

17. Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., *et al.* Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (2022).

18. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484 (2016).

19. Brown, N. & Sandholm, T. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science* **359**, 418 (2018).

20. Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596**, 583 (2021).

21. Davies, A., Veličković, P., Buesing, L., Blackwell, S., Zheng, D., Tomašev, N., Tanburn, R., Battaglia, P., Blundell, C., Juhász, A., *et al.* Advancing mathematics by guiding human intuition with AI. *Nature* **600**, 70 (2021).

22. Schaefer, A. J., Johnson, E. L., Kleywegt, A. J. & Nemhauser, G. L. Airline crew scheduling under uncertainty. *Transportation science* **39**, 340 (2005).

23. Barahona, F., Grötschel, M., Jünger, M. & Reinelt, G. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research* **36**, 493 (1988).

24. Reményi, A., Schöler, H. R. & Wilmanns, M. Combinatorial control of gene expression. *Nature structural & molecular biology* **11**, 812 (2004).

25. Hendrik Kappes, J., Speth, M., Reinelt, G. & Schnorr, C. *Towards efficient and exact MAP-inference for large scale discrete computer vision problems via combinatorial optimization* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2013), 1752.

26. Djolonga, J. & Krause, A. From map to marginals: Variational inference in bayesian submodular models. *Advances in Neural Information Processing Systems* **27** (2014).

27. Cheng, C.-H., Nührenberg, G. & Ruess, H. *Maximum resilience of artificial neural networks* in *International Symposium on Automated Technology for Verification and Analysis* (2017), 251.

28. Tjeng, V., Xiao, K. Y. & Tedrake, R. *Evaluating Robustness of Neural Networks with Mixed Integer Programming* in *International Conference on Learning Representations* (2018).

29. Torrey, L. & Shavlik, J. in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques* 242 (IGI global, 2010).

30. Ford, E., Carroll, J. A., Smith, H. E., Scott, D. & Cassell, J. A. Extracting information from the text of electronic medical records to improve case detection: a systematic review. *Journal of the American Medical Informatics Association* **23**, 1007 (2016).

31. Pereira, S., Pinto, A., Alves, V. & Silva, C. A. Brain tumor segmentation using convolutional neural networks in MRI images. *IEEE transactions on medical imaging* **35**, 1240 (2016).

32. Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* **7**, 48 (1956).

33. Kuhn, H. W. The Hungarian method for the assignment problem. *Naval research logistics quarterly* **2**, 83 (1955).

34. Land, A. H. & Doig, A. G. in *50 Years of Integer Programming 1958-2008* 105 (Springer, 2010).

35. Fujishige, S. *Submodular functions and optimization* (Elsevier, 2005).

36. Hazan, T., Papandreou, G. & Tarlow, D. *Perturbations, optimization, and statistics* (MIT Press, 2016).

37. Paulus, M. B., Choi, D., Tarlow, D., Krause, A. & Maddison, C. J. *Gradient Estimation with Stochastic Softmax Tricks* in *Advances in Neural Information Processing Systems* (2020).

38. Paulus, M. B., Maddison, C. J. & Krause, A. *Rao-Blackwellizing the Straight-Through Gumbel-Softmax Gradient Estimator* in *International Conference on Learning Representations* (2021).

39. Paulus, M. B., Zarpellon, G., Krause, A., Charlin, L. & Maddison, C. *Learning to Cut by Looking Ahead: Cutting Plane Selection via Imitation Learning* in *Proceedings of the 39th International Conference on Machine Learning* (2022).

40. Paulus, M. B. & Krause, A. *Learning To Dive In Branch and Bound* in *Under submission.* (2023).

41. Valentin, R., Ferrari, C., Scheurer, J., Amrollahi, A., Wendler, C. & Paulus, M. B. Instance-wise algorithm configuration with graph neural networks. *NeurIPS Machine Learning for Combinatorial Optimization Competition* (2021).

42. Huijben, I. A., Kool, W., Paulus, M. B. & Van Sloun, R. J. *A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning* in (2022).

43. Miladinović, Đ., Shridhar, K., Jain, K., Paulus, M. B., Buhmann, J. M. & Allen, C. *Learning to Drop Out: An Adversarial Approach to Training Sequence VAEs* in *Advances in Neural Information Processing Systems* (2022).

44. Duan, H., Vaezipoor, P., Paulus, M. B., Ruan, Y. & Maddison, C. J. *Augment with Care: Contrastive Learning for Combinatorial Problems* in *Proceedings of the 39th International Conference on Machine Learning* (2022).

45. He, K., Zhang, X., Ren, S. & Sun, J. *Deep Residual Learning for Image Recognition* 2015.

46. Krizhevsky, A., Sutskever, I. & Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM* **60**, 84 (2017).

47. Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M. & Le, Q. V. *Finetuned Language Models Are Zero-Shot Learners* 2022.

48. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. & Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* **21**, 5485 (2020).

49. Kingma, D. P. & Welling, M. Auto-Encoding Variational Bayes. *arXiv e-prints*, arXiv:1312.6114 (2013).

50. Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction* (MIT press, 2018).

51. Chen, T., Kornblith, S., Norouzi, M. & Hinton, G. *A simple framework for contrastive learning of visual representations* in *International conference on machine learning* (2020), 1597.

52. Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M. & Sutskever, I. *Zero-shot text-to-image generation* in *Proceedings of the Intern. Conf. on Mach. Learn. (ICML)* (2021), 8821.

53. Hastie, T., Tibshirani, R. & Wainwright, M. Statistical learning with sparsity. *Monographs on statistics and applied probability* **143**, 143 (2015).

54. Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* **58**, 267 (1996).

55. Bach, F., Jenatton, R., Mairal, J., Obozinski, G., *et al.* Convex optimization with sparsity-inducing norms. *Optimization for Machine Learning* **5**, 19 (2011).

56. Huang, J., Zhang, T. & Metaxas, D. *Learning with structured sparsity* in *Proceedings of the 26th Annual International Conference on Machine Learning* (2009), 417.

57. Xu, D., Wang, W., Tang, H., Liu, H., Sebe, N. & Ricci, E. *Structured attention guided convolutional neural fields for monocular depth estimation* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 3917.

58. Kim, Y., Denton, C., Hoang, L. & Rush, A. M. Structured attention networks. *arXiv preprint arXiv:1702.00887* (2017).

59. Niculae, V. & Blondel, M. A regularized framework for sparse and structured neural attention. *Advances in neural information processing systems* **30** (2017).

60. Kim, S. & Xing, E. P. Tree-guided group lasso for multi-response regression with structured sparsity, with an application to eQTL mapping (2012).

61. Kim, B. *Interactive and interpretable machine learning models for human machine collaboration* PhD thesis (Massachusetts Institute of Technology, 2015).

62. Ribeiro, M., Singh, S. & Guestrin, C. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier* in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations* (Association for Computational Linguistics, San Diego, California, 2016), 97.

63. Chen, J., Song, L., Wainwright, M. & Jordan, M. *Learning to Explain: An Information-Theoretic Perspective on Model Interpretation* in *International Conference on Machine Learning* (2018).

64. Dupont, E. *Learning Disentangled Joint Continuous and Discrete Representations* in *Proceedings of the Conf. on Neur. Inf. Process. Syst. (NIPS)* (2018).

65. Bengio, Y., Léonard, N. & Courville, A. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv e-prints*, arXiv:1308.3432 (2013).

66. Bengio, E., Bacon, P.-L., Pineau, J. & Precup, D. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297* (2015).

67. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G. & Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

68. Han, Y., Huang, G., Song, S., Yang, L., Wang, H. & Wang, Y. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **44**, 7436 (2021).

69. Rosenbaum, C., Klinger, T. & Riemer, M. Routing networks: Adaptive selection of non-linear functions for multi-task learning. *arXiv preprint arXiv:1711.01239* (2017).

70. Yang, B., Bender, G., Le, Q. V. & Ngiam, J. Condconv: Conditionally parameterized convolutions for efficient inference. *Advances in Neural Information Processing Systems* **32** (2019).

71. Veit, A. & Belongie, S. *Convolutional networks with adaptive inference graphs* in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), 3.

72. Verelst, T. & Tuytelaars, T. *Dynamic convolutions: Exploiting spatial sparsity for faster inference* in *Proceedings of the ieee/cvf conference on computer vision and pattern recognition* (2020), 2320.

73. Choi, J., Yoo, K. M. & Lee, S. *Unsupervised Learning of Task-Specific Tree Structures with Tree-LSTMs* in *CoRR* (2017).

74. Wang, P.-W., Donti, P., Wilder, B. & Kolter, Z. *Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver* in *International Conference on Machine Learning* (2019), 6545.

75. Asai, M. & Fukunaga, A. *Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary* in *Proceedings of the Conf. on Artif. Intell. (AAAI)* **32** (2018).

76. Amos, B. & Kolter, J. Z. *Optnet: Differentiable optimization as a layer in neural networks* in *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), 136.

77. Kipf, T., Fetaya, E., Wang, K.-C., Welling, M. & Zemel, R. *Neural relational inference for interacting systems* in *International Conference on Machine Learning* (2018).

78. Lorberbom, G., Gane, A., Jaakkola, T. & Hazan, T. *Direct Optimization through argmax for Discrete Variational Auto-Encoder* in *Advances in Neural Information Processing Systems* (2019), 6200.

79. Petersen, F. Learning with Differentiable Algorithms. *arXiv preprint arXiv:2209.00616* (2022).

80. Berthet, Q., Blondel, M., Teboul, O., Cuturi, M., Vert, J.-P. & Bach, F. Learning with differentiable perturbed optimizers. *arXiv preprint arXiv:2002.08676* (2020).

81. Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S. & Kolter, Z. *Differentiable Convex Optimization Layers* in *Advances in Neural Information Processing Systems* (2019).

82. Agrawal, A., Barratt, S., Boyd, S., Busseti, E. & Moursi, W. M. Differentiating through a conic program. *arXiv preprint arXiv:1904.09043* (2019).

83. Djolonga, J. & Krause, A. *Differentiable learning of submodular models* in *Advances in Neural Information Processing Systems* (2017), 1013.

84. Glynn, P. W. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM* **33**, 75 (1990).

85. Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* **8**, 229 (1992).

86. Asmussen, S. & Glynn, P. W. *Stochastic simulation: algorithms and analysis* (Springer Science & Business Media, 2007).

87. Gregor, K., Danihelka, I., Mnih, A., Blundell, C. & Wierstra, D. *Deep autoregressive networks* in *International Conference on Machine Learning* (2014), 1242.

88. Gu, S., Levine, S., Sutskever, I. & Mnih, A. *Muprop: Unbiased back-propagation for stochastic neural networks* in *International Conference on Learning Representations* (2016).

89. Paisley, J., Blei, D. & Jordan, M. Variational Bayesian inference with stochastic search. *arXiv preprint arXiv:1206.6430* (2012).

90. Gregor, K., Danihelka, I., Graves, A., Rezende, D. J. & Wierstra, D. *DRAW: A recurrent neural network for image generation* in *International Conference on Machine Learning* (2015).

91. Tucker, G., Mnih, A., Maddison, C. J., Lawson, J. & Sohl-Dickstein, J. *Rebar: Low-variance, unbiased gradient estimates for discrete latent variable models* in *Advances in Neural Information Processing Systems* (2017), 2627.

92. Grathwohl, W., Choi, D., Wu, Y., Roeder, G. & Duvenaud, D. *Backpropagation through the Void: Optimizing control variates for black-box gradient estimation* in *International Conference on Learning Representations* (2018).

93. Geffner, T. & Domke, J. Using large ensembles of control variates for variational inference. *Advances in Neural Information Processing Systems* **31** (2018).

94. Mnih, A. & Rezende, D. J. *Variational inference for Monte Carlo objectives* in *International Conference on Machine Learning* (2016), 2188.

95. Rezende, D. J., Mohamed, S. & Wierstra, D. *Stochastic backpropagation and approximate inference in deep generative models* in *International Conference on Machine Learning* (2014), 1278.

96. Titsias, M. & Lázaro-gredilla, M. *Doubly Stochastic Variational Bayes for non-Conjugate Inference* in *International Conference on Machine Learning* (2014).

97. Glasserman, P. & Ho, Y.-C. *Gradient estimation via perturbation analysis* (Springer Science & Business Media, 1991).

98. Devroye, L. Nonuniform random variate generation. *Handbooks in operations research and management science* **13**, 83 (2006).

99. Fan, K., Wang, Z., Beck, J., Kwok, J. & Heller, K. A. Fast second order stochastic backpropagation for variational inference. *Advances in Neural Information Processing Systems* **28** (2015).

100. Mohamed, S., Rosca, M., Figurnov, M. & Mnih, A. Monte Carlo Gradient Estimation in Machine Learning. *J. Mach. Learn. Res.* **21**, 132:1 (2020).

101. Maddison, C. J., Mnih, A. & Teh, Y. W. *The concrete distribution: A continuous relaxation of discrete random variables* in *International Conference on Learning Representations* (2017).

102. Jang, E., Gu, S. & Poole, B. *Categorical reparameterization with gumbel-softmax* in *International Conference on Learning Representations* (2016).

103. Luce, R. D. *Individual Choice Behavior: A Theoretical Analysis* (New York: Wiley, 1959).

104. Maddison, C. J., Tarlow, D. & Minka, T. *A* Sampling* in *Advances in Neural Information Processing Systems* (2014).

105. Hinton, G. E. Distributed representations (1984).

106. Tarlow, D., Adams, R. & Zemel, R. *Randomized Optimum Models for Structured Prediction* in *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics* (eds Lawrence, N. D. & Girolami, M.) **22** (PMLR, La Palma, Canary Islands, 2012), 1221.

107. Gane, A., Hazan, T. & Jaakkola, T. *Learning with maximum a-posteriori perturbation models* in *Artificial Intelligence and Statistics* (2014), 247.

108. Schrijver, A. *Combinatorial optimization: polyhedra and efficiency* (Springer Science & Business Media, 2003).

109. Kolmogorov, V. Convergent tree-reweighted message passing for energy minimization. *IEEE transactions on pattern analysis and machine intelligence* **28**, 1568 (2006).

110. Koller, D. & Friedman, N. *Probabilistic graphical models: principles and techniques* (2009).

111. Hazan, T., Maji, S. & Jaakkola, T. *On Sampling from the Gibbs Distribution with Random Maximum A-Posteriori Perturbations* in *Advances in Neural Information Processing Systems* (2013).

112. Rockafellar, R. T. *Convex Analysis* (Princeton University Press, 1970).

113. Wainwright, M. J. & Jordan, M. I. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning* **1**, 1 (2008).

114. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. & Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv e-prints*, arXiv:1603.04467 (2016).

115. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. Automatic differentiation in PyTorch (2017).

116. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D. & Wanderman-Milne, S. *JAX: composable transformations of Python+NumPy programs* version 0.1.55. 2018.

117. Domke, J. in *Advances in Neural Information Processing Systems 23* (eds Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S. & Culotta, A.) 523 (Curran Associates, Inc., 2010).

118. Rockafellar, R. T. Second-order convex analysis. *J. Nonlinear Convex Anal* **1**, 84 (1999).

119. Amos, B., Koltun, V. & Zico Kolter, J. The Limited Multi-Label Projection Layer. *arXiv e-prints*, arXiv:1906.08707 (2019).

120. Martins, A. F. & Kreutzer, J. *Learning what's easy: Fully differentiable neural easy-first taggers* in *Proceedings of the 2017 conference on empirical methods in natural language processing* (2017), 349.

121. Wolfe, P. Finding the nearest point in a polytope. *Mathematical Programming* **11**, 128 (1976).

122. Duchi, J., Shalev-Shwartz, S., Singer, Y. & Chandra, T. *Efficient projections onto the l 1-ball for learning in high dimensions* in *Proceedings of the 25th international conference on Machine learning* (2008), 272.

123. Liu, J. & Ye, J. *Efficient Euclidean projections in linear time* in *Proceedings of the 26th Annual International Conference on Machine Learning* (2009), 657.

124. Blondel, M. *Structured Prediction with Projection Oracles* in *Advances in Neural Information Processing Systems* (2019), 12145.

125. Niculae, V., Martins, A. F., Blondel, M. & Cardie, C. Sparsemap: Differentiable sparse structured inference. *arXiv preprint arXiv:1802.04223* (2018).

126. Blondel, M., Martins, A. F. & Niculae, V. Learning with fenchel-young losses. *Journal of Machine Learning Research* **21**, 1 (2020).

127. Martins, A. & Astudillo, R. *From softmax to sparsemax: A sparse model of attention and multi-label classification* in *International Conference on Machine Learning* (2016), 1614.

128. Plackett, R. L. The analysis of permutations. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* **24**, 193 (1975).

129. Kool, W., van Hoof, H. & Welling, M. Ancestral Gumbel-Top-k Sampling for Sampling Without Replacement. *Journal of Machine Learning Research* **21**, 1 (2020).

130. Tarlow, D., Swersky, K., Zemel, R. S., Adams, R. P. & Frey, B. J. *Fast exact inference for recursive cardinality models* in *28th Conference on Uncertainty in Artificial Intelligence, UAI 2012* (2012), 825.

131. Xie, S. M. & Ermon, S. *Reparameterizable subset sampling via continuous relaxations* in *International Joint Conference on Artificial Intelligence* (2019).

132. Mezuman, E., Tarlow, D., Globerson, A. & Weiss, Y. *Tighter linear program relaxations for high order graphical models* in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence* (2013), 421.

133. Mena, G., Belanger, D., Linderman, S. & Snoek, J. *Learning Latent Permutations with Gumbel-Sinkhorn Networks* in *International Conference on Learning Representations* (2018).

134. Grover, A., Wang, E., Zweig, A. & Ermon, S. *Stochastic Optimization of Sorting Networks via Continuous Relaxations* in *International Conference on Learning Representations* (2019).

135. Sinkhorn, R. & Knopp, P. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics* **21**, 343 (1967).

136. Tutte, W. T. *Graph Theory* (Addison-Wesley, 1984).

137. Boyd, S. & Vandenberghe, L. *Convex Optimization* (Cambridge university press, 2004).

138. Chu, Y. & Liu, T. H. On the shortest arborescence of a directed graph. *Scientia Sinica* **14**, 1396 (1965).

139. Edmonds, J. Optimum Branchings". *Journal of Research of the National Bureau of Standards: Mathematics and mathematical physics. B* **71**, 233 (1967).

140. Kleinberg, J. & Tardos, E. *Algorithm Design* (Pearson Education, 2006).

141. Thurstone, L. L. A law of comparative judgment. *Psychological review* **34**, 273 (1927).

142. Papandreou, G. & Yuille, A. *Perturb-and-MAP Random Fields: Using Discrete Optimization to Learn and Sample from Energy Models* in *International Conference on Computer Vision* (2011).

143. Hazan, T. & Jaakkola, T. *On the partition function and random maximum a-posteriori perturbations* in *International Conference on Machine Learning* (2012).

144. Amos, B. *Differentiable optimization-based modeling for machine learning* PhD thesis (PhD thesis. Carnegie Mellon University, 2019).

145. Blondel, M., Teboul, O., Berthet, Q. & Djolonga, J. Fast Differentiable Sorting and Ranking. *arXiv preprint arXiv:2002.08871* (2020).

146. Adams, R. P. & Zemel, R. S. Ranking via sinkhorn propagation. *arXiv preprint arXiv:1106.1925* (2011).

147. Ross, S., Munoz, D., Hebert, M. & Bagnell, J. A. *Learning Message-Passing Inference Machines for Structured Prediction* in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2011).

148. Poon, H. & Domingos, P. *Sum-product networks: A new deep architecture* in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)* (2011), 689.

149. Domke, J. Learning graphical model parameters with approximate marginal inference. *IEEE transactions on pattern analysis and machine intelligence* **35**, 2454 (2013).

150. Swersky, K., Sutskever, I., Tarlow, D., Zemel, R. S., Salakhutdinov, R. R. & Adams, R. P. *Cardinality restricted boltzmann machines* in *Advances in neural information processing systems* (2012), 3293.

151. Berthet, Q., Blondel, M., Teboul, O., Cuturi, M., Vert, J.-P. & Bach, F. Learning with Differentiable Perturbed Optimizers. *arXiv e-prints*, arXiv:2002.08676 (2020).

152. Mnih, A. & Gregor, K. *Neural variational inference and learning in belief networks* in *Proceedings of the 31st International Conference on International Conference on Machine Learning-Volume 32* (2014), II.

153. Fruchterman, T. M. & Reingold, E. M. Graph drawing by force-directed placement. *Software: Practice and experience* **21**, 1129 (1991).

154. Nangia, N. & Bowman, S. R. Listops: A diagnostic dataset for latent tree learning. *arXiv preprint arXiv:1804.06028* (2018).

155. McAuley, J., Leskovec, J. & Jurafsky, D. *Learning attitudes and attributes from multi-aspect reviews* in *2012 IEEE 12th International Conference on Data Mining* (2012), 1020.

156. Lei, T., Barzilay, R. & Jaakkola, T. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155* (2016).

157. Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., *et al.* Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* **29**, 82 (2012).

158. Pervez, A., Cohen, T. & Gavves, E. *Low Bias Low Variance Gradient Estimates for Hierarchical Boolean Stochastic Networks* in *ICML* (2020).

159. Chung, J., Ahn, S. & Bengio, Y. *Hierarchical multiscale recurrent neural networks* in *International Conference on Learning Representations* (2017).

160. Jang, E., Gu, S. & Poole, B. *Categorical Reparametrization with Gumble-Softmax* in *International Conference on Learning Representations (ICLR 2017)* (2017).

161. LeCun, Y. & Cortes, C. MNIST handwritten digit database. *URL http://yann. lecun. com/exdb/mnist* (2010).

162. Blackwell, D. Conditional Expectation and Unbiased Sequential Estimation. *Ann. Math. Statist.* **18**, 105 (1947).

163. Rao, C. R. *Information and the Accuracy Attainable in the Estimation of Statistical Parameters* in *Breakthroughs in Statistics: Foundations and Basic Theory* (eds Kotz, S. & Johnson, N. L.) (Springer New York, New York, NY, 1992), 235.

164. Maddison, C. J., Tarlow, D. & Minka, T. *A* Sampling* in *Advances in Neural Information Processing Systems 27* (2014).

165. Maddison, C. J. in *Perturbation, Optimization, and Statistics* (eds Hazan, T., Papandreou, G. & Tarlow, D.) (MIT Press, 2016).

166. Kroese, D. P., Taimre, T. & Botev, Z. I. *Handbook of Monte Carlo methods* (John Wiley & Sons, 2013).

167. Liu, R., Regier, J., Tripuraneni, N., Jordan, M. I. & McAuliffe, J. Rao-blackwellized stochastic gradients for discrete distributions. *arXiv preprint arXiv:1810.04777* (2018).

168. Kool, W., van Hoof, H. & Welling, M. *Estimating Gradients for Discrete Random Variables by Sampling without Replacement* in *International Conference on Learning Representations* (2020).

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

170. Martins, A. F. T., Mihaylova, T., Nangia, N. & Niculae, V. *Latent Structure Models for Natural Language Processing* in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts* (Association for Computational Linguistics, Florence, Italy, 2019), 1.

171. Havrylov, S., Kruszewski, G. & Joulin, A. Cooperative learning of disjoint syntax and semantics. *arXiv preprint arXiv:1902.09393* (2019).

172. Salakhutdinov, R. & Murray, I. *On the quantitative analysis of deep belief networks* in *Proceedings of the 25th international conference on Machine learning* (2008), 872.

173. Burda, Y., Grosse, R. & Salakhutdinov, R. *Importance weighted autoencoders* in *International Conference on Learning Representations* (2016).

174. Maddison, C. J., Mnih, A. & Teh, Y. W. *The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables* in *International Conference on Learning Representations* (2017).

175. Bergstra, J. & Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research* **13**, 281 (2012).

176. Shekhovtsov, A. *Cold Rao-Blackwellized Straight-Through Gumbel-Softmax Gradient Estimator* 2023.

177. Fan, T.-H., Chi, T.-C., Rudnicky, A. I. & Ramadge, P. J. *Training Discrete Deep Generative Models via Gapped Straight-Through Estimator* in *International Conference on Machine Learning* (2022), 6059.

178. Kokkinos, I. Rapid deformable object detection using dual-tree branch-and-bound. *Advances in Neural Information Processing Systems* **24** (2011).

179. Komodakis, N., Paragios, N. & Tziritas, G. Clustering via lp-based stabilities. *Advances in neural information processing systems* **21** (2008).

180. Christofides, N. *Worst-case analysis of a new heuristic for the travelling salesman problem* tech. rep. (Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976).

181. Gurobi Optimization, L. *Gurobi optimizer reference manual* 2021.

182. Cplex, I. I. User Manual for CPLEX. *International Business Machines Corporation* **46**, 157 (2009).

183. Gomory, R. *An algorithm for the mixed integer problem* tech. rep. RM-2597 (The Rand Corporation, 1960).

184. Berthold, T. *Primal Heuristics for Mixed Integer Programs* in (2006).

185. Ong, H. L. & Moore, J. Worst-case analysis of two travelling salesman heuristics. *Operations Research Letters* **2**, 273 (1984).

186. Lin, S. & Kernighan, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations research* **21**, 498 (1973).

187. Wallace, C. ZI round, a MIP rounding heuristic. *Journal of Heuristics* **16**, 715 (2010).

188. Balas, E., Schmieta, S. & Wallace, C. Pivot and shift—a mixed integer programming heuristic. *Discrete Optimization* **1**, 3 (2004).

189. Fischetti, M. & Lodi, A. Local branching. *Mathematical programming* **98**, 23 (2003).

190. Danna, E., Rothberg, E. & Pape, C. L. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming* **102**, 71 (2005).

191. Berthold, T. RENS-relaxation enforced neighborhood search (2007).

192. Rothberg, E. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing* **19**, 534 (2007).

193. Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.-K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., *et al.* The SCIP optimization suite 7.0 (2020).

194. Hutter, F., Hoos, H. H. & Leyton-Brown, K. *Sequential model-based optimization for general algorithm configuration* in *International conference on learning and intelligent optimization* (2011), 507.

195. Hutter, F., Xu, L., Hoos, H. H. & Leyton-Brown, K. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* **206**, 79 (2014).

196. Biedenkapp, A., Bozkurt, H. F., Eimer, T., Hutter, F. & Lindauer, M. in *ECAI 2020* 427 (IOS Press, 2020).

197. Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., *et al.* Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349* (2020).

198. Khalil, E., Le Bodic, P., Song, L., Nemhauser, G. & Dilkina, B. *Learning to branch in mixed integer programming* in *Proceedings of the AAAI Conference on Artificial Intelligence* **30** (2016).

199. Alvarez, A. M., Louveaux, Q. & Wehenkel, L. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing* **29**, 185 (2017).

200. Gasse, M., Chételat, D., Ferroni, N., Charlin, L. & Lodi, A. *Exact Combinatorial Optimization with Graph Convolutional Neural Networks* in *Advances in Neural Information Processing Systems 32* (2019).

201. Gupta, P., Gasse, M., Khalil, E., Mudigonda, P., Lodi, A. & Bengio, Y. Hybrid models for learning to branch. *Advances in neural information processing systems* **33**, 18087 (2020).

202. Sun, H., Chen, W., Li, H. & Song, L. Improving learning to branch via reinforcement learning (2020).

203. Zarpellon, G., Jo, J., Lodi, A. & Bengio, Y. *Parameterizing branch-and-bound search trees to learn branching policies* in *Proceedings of the AAAI Conference on Artificial Intelligence* **35** (2021), 3931.

204. He, H., Daume III, H. & Eisner, J. M. Learning to search in branch and bound algorithms. *Advances in neural information processing systems* **27** (2014).

205. Yilmaz, K. & Yorke-Smith, N. Learning efficient search approximation in mixed integer branch and bound. *arXiv preprint arXiv:2007.03948* (2020).

206. Huang, Z., Wang, K., Liu, F., Zhen, H.-L., Zhang, W., Yuan, M., Hao, J., Yu, Y. & Wang, J. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition* **123**, 108353 (2022).

207. Tang, Y., Agrawal, S. & Faenza, Y. *Reinforcement Learning for Integer Programming: Learning to Cut* in *Proceedings of the 37th International Conference on Machine Learning* (eds III, H. D. & Singh, A.) **119** (PMLR, 2020), 9367.

208. Berthold, T., Francobaldi, M. & Hendel, G. Learning to use local cuts. *arXiv preprint arXiv:2206.11618* (2022).

209. Turner, M., Koch, T., Serrano, F. & Winkler, M. Adaptive Cut Selection in Mixed-Integer Linear Programming. *arXiv preprint arXiv:2202.10962* (2022).

210. Song, J., Yue, Y., Dilkina, B., *et al.* A general large neighborhood search framework for solving integer linear programs. *Advances in Neural Information Processing Systems* **33**, 20012 (2020).

211. Wu, Y., Song, W., Cao, Z. & Zhang, J. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems* **34**, 30075 (2021).

212. Ding, J.-Y., Zhang, C., Shen, L., Li, S., Wang, B., Xu, Y. & Song, L. *Accelerating primal solution findings for mixed integer programs based on solution prediction* in *Proceedings of the aaai conference on artificial intelligence* **34** (2020), 1452.

213. Sonnerat, N., Wang, P., Ktena, I., Bartunov, S. & Nair, V. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201* (2021).

214. Liu, D., Fischetti, M. & Lodi, A. *Learning to search in local branching* in *Proceedings of the AAAI Conference on Artificial Intelligence* **36** (2022), 3796.

215. Gasse, M., Bowly, S., Cappart, Q., Charfreitag, J., Charlin, L., Chételat, D., Chmiela, A., Dumouchelle, J., Gleixner, A., Kazachkov, A. M., *et al. The machine learning for combinatorial optimization competition (ml4co): Results and insights* in *NeurIPS 2021 Competitions and Demonstrations Track* (2022), 220.

216. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. & Philip, S. Y. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* **32**, 4 (2020).

217. Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O'Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., Addanki, R., Hapuarachchi, T., Keck, T., Keeling, J., Kohli, P., Ktena, I., Li, Y., Vinyals, O. & Zwols, Y. *Solving Mixed Integer Programs Using Neural Networks* 2020.

218. Marchand, H., Martin, A., Weismantel, R. & Wolsey, L. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics* **123**, 397 (2002).

219. Marchand, H. & Wolsey, L. A. Aggregation and mixed integer rounding to solve MIPs. *Operations research* **49**, 363 (2001).

220. Wesselmann, F. & Stuhl, U. *Implementing cutting plane management and selection techniques* tech. rep. (Technical report, University of Paderborn, 2012).

221. Dey, S. S. & Molinaro, M. Theoretical challenges towards cutting-plane selection. *Math. Program.* 237–266 (2018).

222. Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P., Jarck, K., Koch, T., Linderoth, J., *et al.* MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation* **13**, 443 (2021).

223. Ioffe, S. & Szegedy, C. *Batch normalization: Accelerating deep network training by reducing internal covariate shift* in *International conference on machine learning* (2015), 448.

224. Ba, J. L., Kiros, J. R. & Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

225. Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., *et al.* The SCIP optimization suite 8.0. *arXiv preprint arXiv:2112.08872* (2021).

226. Huang, Z., Wang, K., Liu, F., ling Zhen, H., Zhang, W., Yuan, M., Hao, J., Yu, Y. & Wang, J. *Learning to Select Cuts for Efficient Mixed-Integer Programming* arXiv preprint 2105.13645. 2021.

227. Kingma, D. P. & Ba, J. *Adam: A method for stochastic optimization* in *International Conference on Learning Representations (ICLR)* (2015).

228. Gurobi. *Gurobi Optimizer* `http://www.gurobi.com`.

229. Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* **11**, 37 (1985).

230. Achterberg, T., Berthold, T. & Hendel, G. in *Operations research proceedings 2011* 71 (Springer, 2012).

231. Bertsimas, D. & Tsitsiklis, J. N. *Introduction to linear optimization* (Athena Scientific Belmont, MA, 1997).

232. Pryor, J. & Chinneck, J. W. Faster integer-feasibility in mixed-integer linear programs by branching to force change. *Computers & Operations Research* **38**, 1143 (2011).

233. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9**, 1735 (1997).

234. Kotary, J., Fioretto, F. & Van Hentenryck, P. Learning hard optimization problems: A data generation perspective. *Advances in Neural Information Processing Systems* **34**, 24981 (2021).

235. Maher, S., Miltenberger, M., Pedroso, J. P., Rehfeldt, D., Schwarz, R. & Serrano, F. in *Mathematical Software – ICMS 2016* 301 (Springer International Publishing, 2016).

236. Shaw, P. *Using constraint programming and local search methods to solve vehicle routing problems* in *International conference on principles and practice of constraint programming* (1998), 417.

237. Ahuja, R. K., Orlin, J. B. & Sharma, D. New neighborhood search structures for the capacitated minimum spanning tree problem (1998).

238. Pisinger, D. & Ropke, S. in *Handbook of metaheuristics* 399 (Springer, 2010).

239. Witzig, J. & Gleixner, A. Conflict-driven heuristics for mixed integer programming. *INFORMS Journal on Computing* **33**, 706 (2021).

240. Ha, D. & Schmidhuber, J. World models. *arXiv preprint arXiv:1803.10122* (2018).

241. De Avila Belbute-Peres, F., Smith, K., Allen, K., Tenenbaum, J. & Kolter, J. Z. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems* **31** (2018).

242. Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I. & Bachem, O. Brax–A Differentiable Physics Engine for Large Scale Rigid Body Simulation. *arXiv preprint arXiv:2106.13281* (2021).

243. Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S. & Amodei, D. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* **30** (2017).

244. Beck, A. *First-Order Methods in Optimization* (SIAM, 2017).

245. Rush, A. M. Torch-Struct: Deep Structured Prediction Library. *arXiv preprint arXiv:2002.00876* (2020).

246. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *International Conference on Learning Representations* (2015).

247. Kool, W., van Hoof, H. & Welling, M. Buy 4 REINFORCE Samples, Get a Baseline for Free! (2019).

PUBLICATIONS

Conference contributions:

- Paulus, M. B., Choi, D., Tarlow, D., Krause, A. & Maddison, C. J. *Gradient Estimation with Stochastic Softmax Tricks* in *Advances in Neural Information Processing Systems* (2020)
- Paulus, M. B., Maddison, C. J. & Krause, A. *Rao-Blackwellizing the Straight-Through Gumbel-Softmax Gradient Estimator* in *International Conference on Learning Representations* (2021)
- Paulus, M. B., Zarpellon, G., Krause, A., Charlin, L. & Maddison, C. *Learning to Cut by Looking Ahead: Cutting Plane Selection via Imitation Learning* in *Proceedings of the 39th International Conference on Machine Learning* (2022)
- Duan, H., Vaezipoor, P., Paulus, M. B., Ruan, Y. & Maddison, C. J. *Augment with Care: Contrastive Learning for Combinatorial Problems* in *Proceedings of the 39th International Conference on Machine Learning* (2022)
- Miladinović, Đ., Shridhar, K., Jain, K., Paulus, M. B., Buhmann, J. M. & Allen, C. *Learning to Drop Out: An Adversarial Approach to Training Sequence VAEs* in *Advances in Neural Information Processing Systems* (2022)

Articles in peer-reviewed journals:

- Huijben, I. A., Kool, W., Paulus, M. B. & Van Sloun, R. J. *A Review of the Gumbel-max Trick and its Extensions for Discrete Stochasticity in Machine Learning* in (2022)

Other contributions:

- Valentin, R., Ferrari, C., Scheurer, J., Amrollahi, A., Wendler, C. & Paulus, M. B. Instance-wise algorithm configuration with graph neural networks. *NeurIPS Machine Learning for Combinatorial Optimization Competition* (2021)
- Paulus, M. B. & Krause, A. *Learning To Dive In Branch and Bound* in *Under submission.* (2023)