Optimization for and by Machine Learning

Desmaison Alban

Lady Margaret Hall

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2019

Acknowledgements

First and foremost, let me thank my supervisors, Prof M. Pawan Kumar, Prof Philip Torr and Dr Pushmeet Kohli, for their help and guidance through my graduate studies. Their contagious passion for research and problem solving made this work possible in many ways.

I would like to thank all the members of both the OVAL group and the TVG in Oxford. In particular my collaborators Rudy Bunel and Thalaiyasingam Ajanthan. I also want to thank Siddarth for his most needed help with writing.

I would like to thank all my friends in Oxford for their support and motivation. My years as a graduate student wouldn't have been the same without them. In particular, my house mates for their support, Leonard Berrada for the technical discussions and Derek for the music.

I would also like to acknowledge Microsoft who provided me with a graduate fellowship to support my DPhil.

Finally, I would like to thank my friends and family from France for their support and guidance during these years.

Abstract

Optimization and machine learning are both extremely active research topics. In this thesis, we explore problems at the intersection of the two fields. In particular, we will develop two main ideas.

First, optimization can be used to improve machine learning. We illustrate this idea by considering computer vision tasks that are modelled with dense conditional random fields. Existing solvers for these models are either slow or inaccurate. We show that, by introducing a specialized solver based on proximal minimization and fast filtering, these models can be solved both quickly and accurately. Similarly, we introduce a specialized linear programming solver for block bounded problems, a common class of problems encountered in machine learning. This solver is efficient, easy to tune and simple to integrate inside larger machine learning algorithms.

Second, machine learning can be used to improve optimization, in particular for NP-hard problems. For problems solved by using hand-tuned heuristics, machine learning can be used to discover and improve these heuristics. We show that, for the problem of super-optimization, a better heuristic to explore the space of programs can be learnt using reinforcement learning. For problems where no such heuristics exist, machine learning can be used to get an approximate solution of the original problem. We use this idea to tackle the problem of program synthesis by reformulating it as the problem of learning a program that performs the required task. We introduce a new differentiable formulation of the execution and show that the fastest programs can be recovered for simple tasks.

Contents

1 Introduction

Contents

Machine learning aims at making computers learn using large amount of data. This is achieved by extracting information from data either using statistical or computational methods. Thanks to the large increase of computational power and amount of available data, machine learning methods have been successfully used in the recent years to solve meaningful tasks. The large corpora of text that have been translated for very different tasks or encyclopedia available in different languages have enabled very effective automatic translation systems. The democratization of digital photographs and their broad availability lead to breakthroughs in Computer Vision tasks such as face recognition or scene understanding. The recent increase in computational power has enabled, by running a large amount of simulated environments, training Artificial Intelligence systems for games that match the best humans. At the core, all these machine learning tasks rely on solving an optimization problem. How such a problem is solved is crucial for the machine learning system to be both accurate and fast.

1.1 Overview of Optimization

In this thesis, we denote as optimization the task of minimizing an objective function, optionally under a set of constraints. Such a general formulation allows us to define a wide range of tasks as optimization problems.

Problems of Interest We introduce below multiple tasks that span a large class of optimization problems as we will see in the next paragraph. To begin with, many design tasks can be formulated as optimization problems. Indeed, a classical example is the task of building an energy distribution infrastructure, under the constraint of what can technically be built, such that the loss during transport is minimized. The task in biology of finding how a given protein folds, and thus interact with other proteins, can be modelled as an optimization problem as well. Indeed, a physical model will always move to the configuration of lowest energy. So if we find the configuration with minimum energy of the protein under the constraint of physics, we can know how the protein folds in nature. Another interesting task consists in finding the shortest length of road necessary to connect a set of cities together. It can be seen as minimizing the length of the roads under the constraint that all the cities will be connected together.

We also consider tasks arising from computer science. Indeed, we consider the task of program synthesis. It consists in finding the fastest program that can perform a given input/output mapping. It can be framed as an optimization problem where the runtime of the program is minimized under the constraint that it performs the correct mapping. Similarly, program super-optimization has the same goal but uses a reference program instead of an explicit definition of the mapping.

Finally, machine learning also leads to a large variety of optimization problems. The underlying optimization problems will vary both with the task at hand and the model used to solve it. We will focus here on computer vision tasks such as image segmentation and stereo matching. The first one aims at labelling each pixel of an image based on which object it belongs to and the second one aims at re-aligning two images taken from two cameras. Both these tasks are modelled using graphical models, namely Conditional Random Fields (CRF). The original task is solved by finding the most probable labelling according to the CRF model. This task is called Maximum a Posteriori (MAP) estimation. Note that we will solve both the MAP estimation and different relaxations of the MAP estimation.

Brief Introduction to Computational Complexity To work with the large diversity of optimization problems, the computational complexity theory has been developed. It is a large field in computer science and a good introduction can be found in [1]. In this theory, problems are classified based on their inherent difficulty, regardless of the algorithm used to solve them. The measure of difficulty we are interested in, in this thesis, are computational and resource complexity. Indeed, we will compare algorithm by measuring how fast they can be solved and how much memory is required.

Based on this theory, we can classify the problems considered above in increasing complexity. The task of finding the shortest length of road can be see as finding the minimum spanning tree in the graph formed by the cities. This problem can be solved exactly in linearithmic time using, for example, Kruskal's algorithm [2]. We will call such algorithms combinatorial algorithms.

Next we consider the relaxations of the MAP estimation problem for CRF. The Linear Programming (LP) relaxation has been studied in particular. Multiple approaches have been used to solve this problem. For example, [3] uses message passing and [4] uses a specialized combinatorial optimization algorithm. The complexity of such solvers is linear in the number of edges in the considered problem. This makes them unable to solve the particular problem of dense CRFs where every node of the graphical model is connected to every other and thus the number of edges is quadratic in the number of nodes. Another relaxation, based on the mean field theory, is introduced in [5] to solve dense CRFs problems.

More generally, the problem of solving LPs has been of particular interest for the optimization community and the energy infrastructure task presented above is a simple example of such LP. Generic LP solvers are based on the Simplex [6] or Interior Point methods which are described, for example, in [7]. To be effective in practice they also use extensive heuristics to find and exploit special structure of the problem at hand.

Finally, the original MAP estimation problem, the protein folding problem and the program synthesis problem are NP-hard problems. Such problems cannot be solved in polynomial time and the exact solution can only be found by enumerating all possible solutions. While considering all solutions is impossible even for problems of moderate size, methods such as Branch and Bound (BB) can be used to speed up the exhaustive search. Indeed, when branching on a variable that can take, for example, two value, two new problems are created with the variable fixed to each possible value. When bounding, the quality of these new problems is evaluated to detect if the optimal solution can still be attained. This technique allows to efficiently ignore the region of the solution space where the optimal solution cannot be.

1.2 The Challenges of Machine Learning

Even though many algorithms exist to solve the optimization problems coming from machine learning, many machine learning problems remain unsolved because their underlying optimization is intractable or cannot be solved fast enough to be used in practice. Coming back to the problem of MAP estimation for CRFs, we can see that the original problem is NP-hard and cannot be solved for large instances. We saw that different relaxations of the MAP estimation problems can be solved using specialized algorithms. Unfortunately, such algorithms are prohibitively slow on some instances, such as the ones involving dense CRFs, for which other specialized algorithms have been designed.

This example leads us to identify two main reasons for learning problems to be intractable in practice. First, when the underlying optimization problem is hard to solve, both generic and specialized solvers cannot solve the problem exactly. In practice, two options are possible. The first option is to solve a related and easier problem. For example, the classification problem in machine learning is never solved exactly, but continuous, convex approximations are introduced as they define problems that can be efficiently optimized. The second option is to solve the problem approximately. For example, in deep learning where the optimization problem is complex, first order methods are used to obtain an approximate solution.

Second, even for a feasible machine learning problem, the problem can become prohibitively expensive when the amount of data available increases or the required models become too complex. For example, the algorithms with quadratic time complexity used to solve CRFs need to be replaced by linear time algorithms to be able to solve dense CRFs.

1.3 The Challenges of Optimization

As we have seen above, the most challenging problems we consider for optimization are NP-hard problems. Indeed, for such problems, it is expected that no efficient algorithm exists, see [8] for more details on this question, and exhaustive search of all the possible solutions is required when a naive approach is used. Many heuristics are used to speed up the resolution of the problems that are of practical interest. Using these heuristics, some NP-hard problems can be solved almost optimally in a feasible amount of time.

Using these solvers to optimize many, possibly large, NP-hard problems is complex for two reasons. First, it is expected that no algorithm is able to solve all

instances of these problems in a polynomial time. This means that no generic solver can be built to solve this class of problems efficiently. Second, even though in practice, many problems can be solved, the design of specialised solvers and heuristics is expensive and starts almost from scratch for every new problem that is considered.

1.4 The Synergy of Optimization and Machine Learning

In this thesis, we will present our approaches to overcome these limitations both for optimization and machine learning algorithms. We separate them into three different main ideas.

First, we use continuous relaxation and specialized optimization algorithm that solve challenging machine learning problems. Indeed, considering the dense CRF problem, we see that the current solutions are either too slow or not accurate enough to solve the problem. Our work adds to the corpus of research that shows that specialized optimization algorithms are not only beneficial, but necessary for machine learning algorithms to be applicable to complex, real-life, situations.

Second, we replace the heuristics designed for specific optimization problems using machine learning. Indeed, we want the solver to be able to learn, from solving a large number of instances of a problem, how to solve this problem more efficiently. In particular, because the evaluation criterion for an algorithm is its speed at evaluation time, it is beneficial to perform a large number of operations beforehand if it improves the evaluation performance. This allows a single algorithm to perform better on a problem it repeatedly solves without the need for a human to study and design a heuristic for this problem. This approach shows that optimization solvers can leverage machine learning approaches to be able to improve their performances for problems they solve repeatedly.

Third, we illustrate the idea that some NP-hard problems can be approximated as machine learning problems. Indeed, we convert the minimization of an objective under a set of constraints into the problem of learning a system that is penalized both for large values of the objective and violating the constraints. This approach leads to a very different “relaxation”, compared to the classical ones from the optimization field. The latest machine learning techniques, such as deep learning, can be used to provide high quality approximate solutions for this new problem.

1.5 Summary of Contributions

We now outline the contributions in this thesis that use the different approaches listed above to meet the different challenges we consider. The main body of this thesis is a collection of some of the papers I published during my thesis.

We introduce a LP relaxation for dense CRFs and the associated specialized optimization algorithm. (Chapter 2) We introduce the LP relaxation for the MAP estimation problem for dense CRFs. This relaxation is provably the best relaxation for the MAP estimation problem for CRFs. We design a new optimization algorithm based on fast filtering methods developed in the field of Computer Graphics. We show that this algorithm is both faster and more precise than state-of-the-art, Mean Field-based solvers. This better solution leads to both qualitative and quantitative improvements in the Computer Vision tasks of Semantic Segmentation and Stereo Matching. This works shows that we can overcome the limited scaling of machine learning algorithms, namely algorithms designed for sparse CRFs and Mean Field-based solvers, by designing specialized optimization algorithms.

We formulate the problem of program synthesis as a machine learning problem and present an approach to solve it. (Chapter 3) We introduce a new differentiable relaxation of the execution of a program. It enables us to use approaches from deep learning to learn the best possible program. We show that such a system is able to find programs that perform the correct mapping for simple tasks. It is also able to find the fastest program for these tasks. This work shows that, even though no exact solver can be found, machine learning can be used to find approximate solutions to the problem.

We show that machine learning can be used to improve heuristics for the super-optimization problem. (Chapter 3) The super-optimization tasks aims at improving an existing program by making it faster while preserving the mapping it defines. We build on top of the state-of-the-art solver STOKE [9], a continuous research effort for the past years, by using machine learning to improve heuristics. Indeed, the solver performs stochastic search over a very large space of possible modifications of the source code. We show that a machine learning algorithm is able to speed up the search by using informations about past searches over a set of automatically generated programs. By using automatically generated programs, we enable the solver to learn without any human supervision or examples

and thus speed up the learning. This work shows that, the quality of heuristics can be improved with machine learning, leading to their manual design to be less critical for the quality of the optimization algorithm.

We introduce a new linear programming solver for block bounded problems. (Chapter 4) In this work, we focus on LPs that have special common properties. We consider problems where the constraints decomposes into blocks that work with small subsets of the variables and are easy to solve. These blocks should be similar to be able to be solved using a common algorithm. We also consider that the exact solution is not required and an accurate lower bound on the minimum value is sufficient and that we solve multiple related problems sequentially. These properties are common in many problems where the LP solver is used as subroutine, for example BB frameworks or neural network verification. Many classical optimization problems such as CRF optimization or combinatorial graph problems also exhibit such properties. We design a new solver that exploits these properties. It is built on top of classical approaches such as Lagrangean Decomposition (LD) and proximal minimization. We show that each proximal problem can be solved efficiently using the FW algorithms for which we derive an analytical optimal step-size. The careful implementation we provide is both simple and fast by using new hardware such as GPUs. We show that this solver outperforms both generic and specialized solvers in terms of runtime for both large problems and large batches of small problems.

1.6 Other Publications

In addition to the work described above, I had the chance to cooperate on the following works.

I contributed to the publication of [10]. It is the reference publication for the PyTorch project and discuss how the automatic differentiation is designed. I have been a contributor to PyTorch since its inception and have helped both with support and small features. Notable features include higher order derivatives for Convolutions, error and NaN detection during backward pass, work on in-place automatic differentiation and multiple bug fixes.

I contributed to the extension of the works presented in Chapter 2 in [11]. It introduces higher order terms to the CRF model in order to improve quality of the model. This contribution has mostly involved discussion about the fast filtering algorithm and its modifications.

I contributed to [12] for the technical details of its implementation and the experiments on Multi-MNIST

I contributed to [13] by supervising the student responsible for the project. I also implemented the low level code that extracts the training data and helped design of the overall pipeline.

2

Optimization For Machine Learning

Contents

2.1 Efficient Continuous Relaxation for Dense CRF

2.1.1 Preamble

The remaining of this section contains a paper published at ECCV 2016, namely [14]. This work has been done in collaboration with Bunel Rudy which worked on the quadratic programming relaxations of Section 2.1.5 and Section 2.1.6.

Dense conditional random fields (CRF) with Gaussian pairwise potentials have emerged as a popular framework for several computer vision applications such as stereo correspondence and semantic segmentation. By modeling long-range interactions, dense CRFs provide a more detailed labelling compared to their sparse counterparts. Variational inference in these dense models is performed using a filtering-based mean-field algorithm in order to obtain a fully-factorized distribution minimizing the Kullback-Leibler divergence to the true distribution. In contrast to the continuous relaxation-based energy minimization algorithms used for sparse CRFs, the mean-field algorithm fails to provide strong theoretical guarantees on the quality of its solutions. To address this deficiency, we show that it is possible to use the same filtering approach to speed-up the optimization of several continuous relaxations. Specifically, we solve a convex quadratic programming (QP) relaxation using the efficient Frank-Wolfe algorithm. This also allows us to solve difference-of-convex relaxations via the iterative concave-convex procedure where each iteration requires solving a convex QP. Finally, we develop a novel divide-and-conquer method to compute the subgradients of a linear programming relaxation that provides the best theoretical bounds for energy minimization. We demonstrate the advantage of continuous relaxations over the widely used mean-field algorithm on publicly available datasets.

2.1.2 Introduction

Discrete pairwise conditional random fields (CRFs) are a popular framework for modelling several problems in computer vision. In order to use them in practice, one requires an energy minimization algorithm that obtains the most likely output for a given input. The energy function consists of a sum of two types of terms: unary potentials that depend on the label for one random variable at a time and pairwise potentials that depend on the labels of two random variables.

Traditionally, computer vision methods have employed sparse connectivity structures, such as 4 or 8 connected grid CRFs. Their popularity lead to a considerable research effort in efficient energy minimization algorithms. One of the

biggest successes of this effort was the development of several accurate continuous relaxations of the underlying discrete optimization problem [15, 16]. An important advantage of such relaxations is that they lend themselves easily to analysis, which allows us to compare them theoretically [17], as well as establish bounds on the quality of their solutions [18].

Recently, the influential work of Krähenbühl and Koltun [5] has popularised the use of dense CRFs, where each pair of random variables is connected by an edge. Dense CRFs capture useful long-range interactions thereby providing finer details on the labelling. However, modeling long-range interactions comes at the cost of a significant increase in the complexity of energy minimization. In order to operationalise dense CRFs, Krähenbühl and Koltun [5] made two key observations. First, the pairwise potentials used in computer vision typically encourage smooth labelling. This enabled them to restrict themselves to the special case of Gaussian pairwise potentials introduced by Tappen et al. [19]. Second, for this special case, it is possible to obtain a labelling efficiently by using the mean-field algorithm [20]. Specifically, the message computation required at each iteration of mean-field can be carried out in $O(N)$ operations where N is the number of random variables (of the order of hundreds of thousands). This is in contrast to a naïve implementation that requires $O(N^2)$ operations. The significant speed-up is made possible by the fact that the messages can be computed using the filtering approach of Adams et al. [21].

While the mean-field algorithm does not provide any theoretical guarantees on the energy of the solutions, the use of a richer model, namely dense CRFs, still allows us to obtain a significant improvement in the accuracy of several computer vision applications compared to sparse CRFs [5]. However, this still leaves open the intriguing possibility that the same filtering approach that enabled the efficient mean-field algorithm can also be used to speed-up energy minimization algorithms based on continuous relaxations. In this work, we show that this is indeed possible.

In more detail, we make three contributions to the problem of energy minimization in dense CRFs. First, we show that the conditional gradient of a convex quadratic programming (QP) relaxation [15] can be computed in $O(N)$ complexity. Together with our observation that the optimal step-size of a descent direction can be computed analytically, this allows us to minimise the QP relaxation efficiently using the Frank-Wolfe algorithm [22]. Second, we show that difference-of-convex (DC) relaxations of the energy minimization problem can be optimised efficiently using an iterative concave-convex procedure (CCCP). Each iteration of CCCP requires solving a convex QP, for which we can once again employ the Frank-Wolfe algorithm. Third, we show that a linear programming (LP) relaxation [16]

of the energy minimization problem can be optimised efficiently via subgradient descent. Specifically, we design a novel divide-and-conquer method to compute the subgradient of the LP. Each subproblem of our method requires one call to the filtering approach. This results in an overall run-time of $O(N \log(N))$ per iteration as opposed to an $O(N^2)$ complexity of a naïve implementation. It is worth noting that the LP relaxation is known to provide the best theoretical bounds for energy minimization with metric pairwise potentials [16].

Using standard publicly available datasets, we demonstrate the efficacy of our continuous relaxations by comparing them to the widely used mean-field baseline for dense CRFs.

2.1.3 Related works

Krähenbühl and Koltun popularised the use of densely connected CRFs at the pixel level [5], resulting in significant improvements both in terms of the quantitative performance and in terms of the visual quality of their results. By restricting themselves to Gaussian edge potentials, they made the computation of the message in parallel mean-field feasible. This was achieved by formulating message computation as a convolution in a higher-dimensional space, which enabled the use of an efficient filter-based method [21].

While the original work [5] used a version of mean-field that is not guaranteed to converge, their follow-up paper [23] proposed a convergent mean-field algorithm for negative semi-definite label compatibility functions. Recently, Baqué et al. [24] presented a new algorithm that has convergence guarantees in the general case. Vineet et al. [25] extended the mean-field model to allow the addition of higher-order terms on top of the dense pairwise potentials, enabling the use of co-occurence potentials [26] and P^n-Potts models [27].

The success of the inference algorithms naturally lead to research in learning the parameters of dense CRFs. Combining them with Fully Convolutional Neural Networks [28] has resulted in high performance on semantic segmentation applications [29]. Several works [30, 31] showed independently how to jointly learn the parameters of the unary and pairwise potentials of the CRF. These methods led to significant improvements on various computer vision applications, by increasing the quality of the energy function to be minimised by mean-field.

Independently from the mean-field work, Zhang et al. [32] designed a different set of constraints that lends itself to a QP relaxation of the original problem. Their approach is similar to ours in that they use continuous relaxation to approximate the solution of the original problem but differ in the form of the pairwise potentials.

The algorithm they propose to solve the QP relaxation has linearithmic complexity while ours is linear. Furthermore, it is not clear whether their approach can be easily generalised to tighter relaxations such as the LP.

Wang et al. [33] derived a semi-definite programming relaxation of the energy minimization problem, allowing them to reach better energy than mean-field. Their approach has the advantage of not being restricted to Gaussian pairwise potentials. The inference is made feasible by performing low-rank approximation of the Gram matrix of the kernel, instead of using the filter-based method. However, while the complexity of their algorithm is the same as our QP or DC relaxation, the runtime is significantly higher. Furthermore, while the SDP relaxation has been shown to be accurate for repulsive pairwise potentials (encouraging neighbouring variables to take different labels) [34], our LP relaxation provides the best guarantees for attractive pairwise potentials [16].

In this paper, we use the same filter-based method [21] as the one employed in mean-field. We build on it to solve continuous relaxations of the original problem that have both convergence and quality guarantees. Our work can be viewed as a complementary direction to previous research trends in dense CRFs. While [23–25] improved mean-field and [30, 31] learnt the parameters, we focus on the energy minimization problem.

2.1.4 Preliminaries

Before describing our methods for energy minimization on dense CRF, we establish the necessary notation and background information.

2.1.4.0.1 Dense CRF Energy Function. We define a dense CRF on a set of N random variables $\mathcal{X} = \{X_1, \ldots, X_N\}$ each of which can take one label from a set of M labels $\mathcal{L} = \{l_1, \ldots l_M\}$. To describe a labelling, we use a vector $\mathbf{x}$ of size N such that its element x_a is the label taken by the random variable X_a. The energy associated with a given labelling is defined as:

$$E(\mathbf{x}) = \sum_{a=1}^{N} \phi_a(x_a) + \sum_{a=1}^{N} \sum_{\substack{b=1 \\ b \neq a}}^{N} \psi_{a,b}(x_a, x_b). \tag{2.1.4.1}$$

Here, $\phi_a(x_a)$ is called the *unary potential* for the random variable X_a taking the label x_a. The term $\psi_{a,b}(x_a, x_b)$ is called the *pairwise potential* for the random variables X_a and X_b taking the labels x_a and x_b respectively. The energy minimization problem on this CRF can be written as:

$$\mathbf{x}^\star = \operatorname*{argmin}_{\mathbf{x}} E(\mathbf{x}). \tag{2.1.4.2}$$

2.1.4.0.2 Gaussian Pairwise Potentials. Similar to previous work [5], we consider arbitrary unary potentials and Gaussian pairwise potentials. Specifically, the form of the pairwise potentials is given by:

$$\psi_{a,b}(i,j) = \mu(i,j) \sum_m w^{(m)} k(\mathbf{f}_a^{(m)}, \mathbf{f}_b^{(m)}), \tag{2.1.4.3}$$

$$k(\mathbf{f}_a, \mathbf{f}_b) = \exp\left(\frac{-\|\mathbf{f}_a - \mathbf{f}_b\|^2}{2}\right) \tag{2.1.4.4}$$

We refer to the term $\mu(i,j)$ as a *label compatibility* function between the labels i and j. An example of a label compatibility function is the Potts model, where $\mu_{\text{potts}}(i,j) = [i \neq j]$, that is $\mu_{potts}(i,j) = 1$ if $i \neq j$ and 0 otherwise. Note that the label compatibility does not depend on the image. The other term, called the *pixel compatibility* function, is a mixture of gaussian kernels $k(\cdot,\cdot)$. The coefficients of the mixture are the positive weights $w^{(m)}$. The $\mathbf{f}_a^{(m)}$ are the features describing the random variable X_a. Note that the pixel compatibility does not depend on the labelling. In practice, similar to [5], we use the position and RGB values of a pixel as features.

2.1.4.0.3 IP Formulation. We now introduce a formulation of the energy minimization problem that is more amenable to continuous relaxations. Specifically, we formulate it as an Integer Program (IP) and then relax it to obtain a continuous optimization problem. To this end, we define the vector $\mathbf{y}$ whose components $y_a(i)$ are indicator variables specifying whether or not the random variable X_a takes the label i. Using this notation, we can rewrite the energy minimization problem as an IP:

$$\begin{aligned} \min \quad & \sum_{a=1}^{N} \sum_{i \in \mathcal{L}} \phi_a(i) y_a(i) + \sum_{a=1}^{N} \sum_{\substack{b=1 \\ b \neq a}}^{N} \sum_{i,j \in \mathcal{L}} \psi_{a,b}(i,j) y_a(i) y_b(j), \\ \text{s.t.} \quad & \sum_{i \in \mathcal{L}} y_a(i) = 1 \quad \forall a \in [1,N], \\ & y_a(i) \in \{0,1\} \quad \forall a \in [1,N] \quad \forall i \in \mathcal{L}. \end{aligned} \tag{2.1.4.5}$$

The first set of constraints model the fact that each random variable has to be assigned exactly one label. The second set of constraints enforce the optimization variables $y_a(i)$ to be binary. Note that the objective function is equal to the energy of the labelling encoded by $\mathbf{y}$.

2.1.4.0.4 Filter-based Method. Similar to [5], a key component of our algorithms is the filter-based method of Adams et al. [21]. It computes the following operation:

$$\forall a \in [1, N], \quad v'_a = \sum_{b=1}^{N} k(\mathbf{f}_a, \mathbf{f}_b) v_b, \tag{2.1.4.6}$$

where $v'_a, v_b \in \mathbb{R}$ and $k(\cdot, \cdot)$ is a Gaussian kernel. Performing this operation the naïve way would result in computing a sum on N elements for each of the N terms that we want to compute. The resulting complexity would be $\mathcal{O}(N^2)$. The filter-based method allows us to perform it approximately with $\mathcal{O}(N)$ complexity. We refer the interested reader to [21] for details. The accuracy of the approximation made by the filter-based method is explored in the supplementary material.

2.1.5 Quadratic Programming Relaxation

We are now ready to demonstrate how the filter-based method [21] can be used to optimise our first continuous relaxation, namely the convex quadratic programming (QP) relaxation.

2.1.5.0.1 Notation. In order to concisely specify the QP relaxation, we require some additional notation. Similar to [23], we rewrite the objective function with linear algebra operations. The vector $\boldsymbol{\phi}$ contains the unary terms. The matrix $\boldsymbol{\mu}$ corresponds to the label compatibility function. The Gaussian kernels associated with the m-th features are represented by their Gram matrix $\mathbf{K}^{(m)}_{a,b} = k(\mathbf{f}^{(m)}_a, \mathbf{f}^{(m)}_b)$. The Kronecker product is denoted by $\otimes$. The matrix $\boldsymbol{\Psi}$ represents the pairwise terms and is defined as follows:

$$\boldsymbol{\Psi} = \boldsymbol{\mu} \otimes \sum_m w^{(m)} \left(\mathbf{K}^{(m)} - \mathbf{I}_N \right), \tag{2.1.5.1}$$

where $\mathbf{I}_N$ is the identity matrix. Under this notation, the IP (2.1.4.5) can be concisely written as

$$\begin{aligned} \min \quad & \boldsymbol{\phi}^T \mathbf{y} + \mathbf{y}^T \boldsymbol{\Psi} \mathbf{y}, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{I}, \end{aligned} \tag{2.1.5.2}$$

with $\mathcal{I}$ being the feasible set of integer solution, as defined in Eq. (2.1.4.5).

2.1.5.0.2 Relaxation. In general, IP such as (2.1.5.2) are NP-hard problems. Relaxing the integer constraint on the indicator variables to allow fractional values between 0 and 1 results in the QP formulation. Formally, the feasible set of our minimization problem becomes:

$$\mathcal{M} = \left\{ \mathbf{y} \quad \text{such that} \quad \begin{array}{ll} \sum_{i \in \mathcal{L}} y_a(i) = 1 & \forall a \in [1, N], \\ y_a(i) \geq 0 & \forall a \in [1, N], \forall i \in \mathcal{L} \end{array} \right\}. \tag{2.1.5.3}$$

Ravikumar and Lafferty [15] showed that this relaxation is tight and that solving the QP will result in solving the IP. However, this QP is still NP-hard, as the objective function is non-convex. To alleviate this difficulty, Ravikumar and Lafferty [15] relaxed the QP minimization to the following convex problem:

$$\begin{array}{ll} \min & S_{cvx}(\mathbf{y}) = (\boldsymbol{\phi} - \mathbf{d})^T \mathbf{y} + \mathbf{y}^T (\boldsymbol{\Psi} + \mathbf{D}) \mathbf{y}, \\ \text{s.t.} & \mathbf{y} \in \mathcal{M}, \end{array} \tag{2.1.5.4}$$

where the vector $\mathbf{d}$ is defined as follows

$$d_a(i) = \sum_{\substack{b=1 \\ b \neq a}}^{N} \sum_{j \in \mathcal{L}} |\psi_{a,b}(i, j)|, \tag{2.1.5.5}$$

and $\mathbf{D}$ is the square diagonal matrix with $\mathbf{d}$ as its diagonal.

2.1.5.0.3 Minimization. We now introduce a new method based on the Frank-Wolfe algorithm [22] to minimise problem (2.1.5.4). The Frank-Wolfe algorithm allows to minimise a convex function f over a convex feasible set $\mathcal{M}$. The key steps of the algorithm are shown in Algorithm 1. To be able to use the Frank-Wolfe algorithm, we need a way to compute the gradient of the objective function (Step 3), a method to compute the conditional gradient (Step 4) and a strategy to choose the step size (Step 5).

Algorithm 1: Frank-Wolfe algorithm

1 Get $\mathbf{y}^0 \in \mathcal{M}$
2 **while** *not converged* **do**
3 Compute the gradient at $\mathbf{y}^t$ as $\mathbf{g} = \nabla f(\mathbf{y}^t)$
4 Compute the conditional gradient as $\mathbf{s} = \text{argmin}_{\mathbf{s} \in \mathcal{M}} \langle \mathbf{s}, \mathbf{g} \rangle$
5 Compute a step-size $\alpha = \text{argmin}_{\alpha \in [0,1]} f(\alpha \mathbf{y}^t + (1 - \alpha)\mathbf{s})$
6 Move towards the negative conditional gradient $\mathbf{y}^{t+1} = \alpha \mathbf{y}^t + (1 - \alpha)\mathbf{s}$
7 **end**

Gradient Computation

Since the objective function is quadratic, its gradient can be computed as

$$\nabla S_{\text{cvx}}(\mathbf{y}) = (\boldsymbol{\phi} - \mathbf{d}) + 2(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y}. \tag{2.1.5.6}$$

What makes this equation expensive to compute in a naïve way is the matrix product with $\boldsymbol{\Psi}$. We observe that this operation can be performed using the filter-based method in linear time. Note that the other matrix-vector product, $\mathbf{Dy}$, is not expensive (linear in N) since $\mathbf{D}$ is a diagonal matrix.

Conditional Gradient

The conditional gradient is obtained by solving

$$\underset{\mathbf{s}\in\mathcal{M}}{\operatorname{argmin}} \langle \mathbf{s}, \nabla S_{\text{cvx}}(\mathbf{y}) \rangle. \tag{2.1.5.7}$$

Minimizing such an LP would usually be an expensive operation for problems of this dimension. However, we remark that, once the gradient has been computed, exploiting the properties of our problem allows us to solve problem (2.1.5.7) in a time linear in the number of random variables (N) and labels (M). Specifically, the following is an optimal solution to problem (2.1.5.7).

$$\mathbf{s}_a(i) = \begin{cases} 1 & \text{if } i = \operatorname{argmin}_{i\in\mathcal{L}} \frac{\partial S_{\text{cvx}}}{\partial y_a(i)} \\ 0 & \text{else.} \end{cases} . \tag{2.1.5.8}$$

Step Size Determination

In the original Frank-Wolfe algorithm, the step size α is simply chosen using line search. However we observe that, in our case, the optimal α can be computed by solving a second-order polynomial function of a single variable, which has a closed form solution that can be obtained efficiently. This observation has been previously exploited in the context of Structural SVM [35]. The derivations for this closed form solution can be found in supplementary material. With careful reutilisation of computations, this step can be performed without additional filter-based method calls. By choosing the optimal step size at each iteration, we reduce the number of iterations needed to reach convergence.

The above procedure converges to the global minimum of the convex relaxation and resorts to the filter-based method only once per iteration during the computation of the gradient and is therefore efficient. However, this solution has no guarantees to be even a local minimum of the original QP relaxation. To alleviate this, we will now introduce a difference-of-convex (DC) relaxation.

2.1.6 Difference of Convex Relaxation

2.1.6.1 DC Relaxation: General Case

The objective function of a general DC program can be specified as

$$S_{\text{CCCP}}(\mathbf{y}) = p(\mathbf{y}) - q(\mathbf{y}). \tag{2.1.6.1}$$

One can obtain one of its local minima using the Concave-Convex Procedure (CCCP) [36]. The key steps of this algorithm are described in Algorithm 2. Briefly, Step 3 computes the gradient of the concave part. Step 4 minimises a convex upper bound on the DC objective, which is tight at $\mathbf{y}^t$.

In order to exploit the CCCP algorithm for DC programs, we observe that the QP (2.1.5.2) can be rewritten as

$$\begin{aligned} \min_{\mathbf{y}} \quad & \boldsymbol{\phi}^T\mathbf{y} + \mathbf{y}^T(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y} - \mathbf{y}^T\mathbf{D}\mathbf{y}, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{M}. \end{aligned} \tag{2.1.6.2}$$

Formally, we can define $p(\mathbf{y}) = \boldsymbol{\phi}^T\mathbf{y} + \mathbf{y}^T(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y}$ and $q(\mathbf{y}) = \mathbf{y}^T\mathbf{D}\mathbf{y}$, which are both convex in $\mathbf{y}$.

Algorithm 2: CCCP Algorithm

1 Get $\mathbf{y}^0 \in \mathcal{M}$
2 **while** *not converged* **do**
3 | Linearise the concave part $\mathbf{g} = \nabla q(\mathbf{y}^t)$
4 | Minimise a convex upper-bound $\mathbf{y}^{t+1} = \text{argmin}_{\mathbf{y}\in\mathcal{M}}\, p(\mathbf{y}) - \mathbf{g}^T\mathbf{y}$
5 **end**

We observe that, since $\mathbf{D}$ is diagonal and the matrix product with $\boldsymbol{\Psi}$ can be computed using the filter based method, the gradient $\nabla q(\mathbf{y}^t) = 2\mathbf{D}\mathbf{y}$ (Step 3) is efficient to compute. The minimization of the convex problem (Step 4) is analogous to the convex QP formulation (2.1.5.4) presented above with different unary potentials. Since we do not place any restrictions on the form of the unary potentials, (Step 4) can be implemented using the method described in Section 2.1.5.

The CCCP algorithm provides a monotonous decrease in the objective function and will converge to a local minimum [37]. However, the above method will take several iterations to converge, each necessitating the solution of a convex QP, and thus requiring multiple calls to the filter-based method. While the filter-based method [21] allows us to compute operations on the pixel compatibility function in linear time, it still remains an expensive operation to perform. As we show next, if we introduce some additional restriction on our potentials, we can obtain a more efficient difference of convex decomposition.

2.1.6.2 DC Relaxation: Negative Semi-definite Compatibility

We now introduce a new DC relaxation of our objective function that takes advantage of the structure of the problem. Specifically, the convex problem to solve at each iteration does not depend on the filter-based method computations, which are the expensive steps in the previous method. Following the example of Krähenbühl and Koltun [23], we look at the specific case of negative semi-definite label compatibility function, such as the commonly used Potts model. Note that we consider a label compatibility function to be negative semi-definite if there exist a constant d such that the matrix $\boldsymbol{\mu} + d$ is negative semi-definite. Indeed, such a constant will not change the optimal value of the minimization problem and thus can be used to ensure negative semi-definitiveness. In particular, for the Potts model, we have $d = -1$ and the negative semi-definite matrix is negative identity. Taking advantage of the specific form of our pairwise terms (2.1.5.1), we can rewrite the problem as

$$S(\mathbf{y}) = \boldsymbol{\phi}^T\mathbf{y} - \mathbf{y}^T(\boldsymbol{\mu} \otimes \sum_m w^{(m)}\mathbf{I}_N)\mathbf{y}^T + \mathbf{y}^T(\boldsymbol{\mu} \otimes \sum_m w^{(m)}\mathbf{K}^{(m)})\mathbf{y}. \qquad (2.1.6.3)$$

The first two terms can be verified as being convex. The Gaussian kernel is positive semi-definite, so the Gram matrices $\mathbf{K}^{(m)}$ are positive semi-definite. By assumption, the label compatibility function is also negative semi-definite. The results from the Kronecker product between the Gram matrix and μ is therefore negative semi-definite.

2.1.6.2.1 Minimization. Once again we use the CCCP Algorithm. The main difference between the generic DC relaxation and this specific one is that Step 3 now requires a call to the filter-based method, while the iterations required to solve Step 4 do not. In other words, each iteration of CCCP only requires one call to the filter based method. This results in a significant improvement in speed. More details about this operation are available in the supplementary material.

2.1.7 LP relaxation

This section presents an accurate LP relaxation of the energy minimization problem and our method to optimise it efficiently using subgradient descent.

2.1.7.0.1 Relaxation. To simplify the description, we focus on the Potts model. However, our approach can easily be extended to more general pairwise potentials by approximating them using a hierarchical Potts model. Such an extension, inspired by [38], is presented in the supplementary material. We define the following notation: $K_{a,b} = \sum_m w^{(m)} k^{(m)}(\mathbf{f}_a^{(m)}, \mathbf{f}_b^{(m)})$, $\sum_a = \sum_{a=1}^{N}$ and $\sum_{b<a} = \sum_{b=1}^{a-1}$. With these notations, a LP relaxation of (2.1.4.5) is:

$$\begin{aligned} \min \quad & S_{LP}(\mathbf{y}) = \underbrace{\sum_a \sum_i \phi_a(i) y_a(i)}_{unary} + \underbrace{\sum_a \sum_{b \neq a} \sum_i K_{a,b} \frac{|y_a(i) - y_b(i)|}{2}}_{pairwise}, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{M}. \end{aligned} \tag{2.1.7.1}$$

The feasible set remains the same as the one we had for the QP and DC relaxations. In the case of integer solutions, $S_{LP}(\mathbf{y})$ has the same value as the objective function of the IP described in (2.1.4.5). The *unary* term is the same for both formulations. The *pairwise* term ensures that for every pair of random variables X_a, X_b, we add the cost $K_{a,b}$ associated with this edge only if they are not associated with the same labels.

2.1.7.0.2 Minimization. Kleinberg and Tardos [16] solve this problem by introducing extra variables for each pair of pixels to get a standard LP, with a linear objective function and linear constraints. In the case of a dense CRF, this is infeasible because it would introduce a number of variables quadratic in the number of pixels. We will instead use projected subgradient descent to solve this LP. To do so, we will reformulate the objective function, derive the subgradient, and present an algorithm to compute it efficiently.

Reformulation

The absolute value in the pairwise term of (2.1.4.5) prevents us from using the filtering approach. To address this issue, we consider that for any given label i, the variables $y_a(i)$ can be sorted in a descending order: $a \geq b \implies y_a(i) \leq y_b(i)$. This allows us to rewrite the pairwise term of the objective function (2.1.7.1) as:

$$\sum_i \sum_a \sum_{a \neq b} K_{a,b} \frac{|y_a(i) - y_b(i)|}{2} = \sum_i \sum_a \sum_{b>a} K_{a,b} y_a(i) - \sum_i \sum_a \sum_{b<a} K_{a,b} y_a(i). \tag{2.1.7.2}$$

A formal derivation of this equality can be found in supplementary material.

Subgradient

From (2.1.7.2), we rewrite the subgradient:

$$\frac{\partial S_{LP}}{\partial y_c(k)}(\mathbf{y}) = \phi_c(k) + \sum_{a>c} K_{a,c} - \sum_{a<c} K_{a,c}. \tag{2.1.7.3}$$

Note that in this expression, the dependency on the variable $\mathbf{y}$ is hidden in the bounds of the sum because we assumed that $y_a(k) \leq y_c(k)$ for all $a > c$. For a different value of $\mathbf{y}$, the elements of $\mathbf{y}$ would induce a different ordering and the terms involved in each summation would not be the same.

Subgradient Computation

What prevents us from evaluating (2.1.7.3) efficiently are the two sums, one over an upper triangular matrix ($\sum_{a>c} K_{a,c}$) and one over a lower triangular matrix ($\sum_{a<c} K_{a,c}$). As opposed to (2.1.4.6), which computes terms $\sum_{a,b} K_{a,b} v_b$ for all a using the filter-based method, the summation bounds here depend on the random variable we are computing the partial derivative for. While it would seems that the added sparsity provided by the upper and lower triangular matrices would simplify the operation, it is this sparsity itself that prevents us from interpreting the summations as convolution operations. Thus, we cannot use the filter-based method as described by Adams et al. [21].

We alleviate this difficulty by designing a novel divide-and-conquer algorithm. We describe our algorithm for the case of the upper triangular matrix. However, it can easily be adapted to compute the summation corresponding to the lower triangular matrix. We present the intuition behind the algorithm using an example. A rigorous development can be found in the supplementary material. If we consider $N = 6$ then $a, c \in \{1, 2, 3, 4, 5, 6\}$ and the terms we need to compute for a given label are:

$$\begin{pmatrix} \sum_{a>1} K_{a,1} \\ \sum_{a>2} K_{a,2} \\ \sum_{a>3} K_{a,3} \\ \sum_{a>4} K_{a,4} \\ \sum_{a>5} K_{a,5} \\ \sum_{a>6} K_{a,6} \end{pmatrix} = \underbrace{\left(\begin{array}{ccc|ccc} 0 & K_{2,1} & K_{3,1} & K_{4,1} & K_{5,1} & K_{6,1} \\ 0 & 0 & K_{3,2} & K_{4,2} & K_{5,2} & K_{6,2} \\ 0 & 0 & 0 & K_{4,3} & K_{5,3} & K_{6,3} \\ \hline 0 & 0 & 0 & 0 & K_{5,4} & K_{6,4} \\ 0 & 0 & 0 & 0 & 0 & K_{6,5} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)}_{\mathbf{U}} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{2.1.7.4}$$

We propose a divide and conquer approach that solves this problem by splitting the upper triangular matrix $\mathbf{U}$. The top-left and bottom-right parts are upper triangular matrices with half the size. We solve these subproblems recursively. The top-right part can be computed with the original filter based method. Using this approach, the total complexity to compute this sum is $\mathcal{O}(N \log(N))$.

With this algorithm, we have made feasible the computation of the subgradient. We can therefore perform projected subgradient descent on the LP objective efficiently. Since we need to compute the subgradient for each label separately due to the necessity of having sorted elements, the complexity associated with taking a gradient step is $\mathcal{O}(MN\log(N))$. To ensure the convergence, we choose as learning rate $(\beta^t)_{t=1}^{\infty}$ that is a square summable but not a summable sequence such as $(\frac{1}{1+t})_{t=1}^{\infty}$. We also make use of the work by Condat [39] to perform fast projection on the feasible set. The complete procedure can be found in Algorithm 3. Step 3 to 7 present the subgradient computation for each label. Using this subgradient, Step 8 shows the update rule for y^t. Finally, Step 9 project this new estimate onto the feasible space.

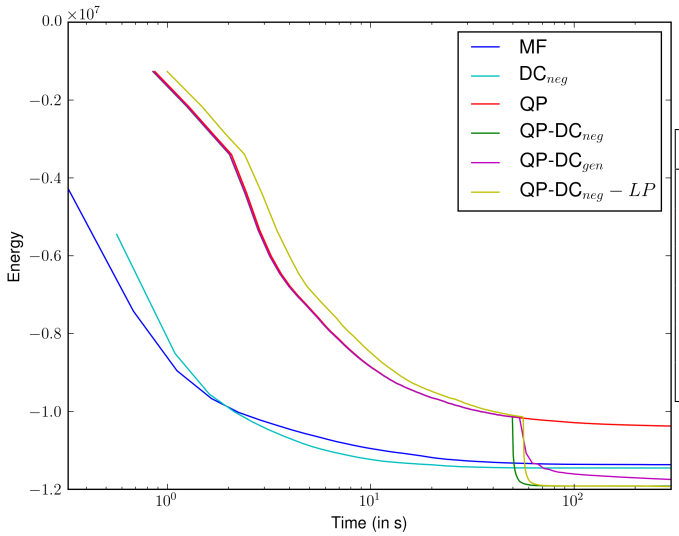
Algorithm 3: LP subgradient descent

```
1  Get y^0 ∈ M
2  while not converged do
3  |  for i ∈ L do
4  |  |  Sort y_a(i)   ∀a ∈ [1, N]
5  |  |  Reorder K
6  |  |  g(i) = ∇S_LP(y^t(i))
7  |  end
8  |  y^{t+1} = y^t − β^t · g
9  |  Project y^{t+1} on the feasible space
10 end
```

The algorithm that we introduced converges to a global minimum of the LP relaxation. By using the rounding procedure introduced by Kleinberg and Tardos [16], it has a multiplicative bound of 2 for the dense CRF labelling problem on Potts models and $\mathcal{O}(\log(M))$ for metric pairwise potentials.

2.1.8 Experiments

We now demonstrate the benefits of using continuous relaxations of the energy minimization problem on two applications: stereo matching and semantic segmentation. We provide results for the following methods: the Convex QP relaxation ($\mathbf{QP_{cvx}}$), the generic and negative semi-definite specific DC relaxations ($\mathbf{DC_{gen}}$ and $\mathbf{DC_{neg}}$) and the LP relaxation (**LP**). We compare solutions obtained by our methods with the mean-field baseline (**MF**).



(a) Runtime comparisons

Method	Final energy
MF	-1.137e+07
DC_{neg}	-1.145e+07
QP	-1.037e+07
QP-DC_{neg}	-1.191e+07
QP-DC_{gen}	-1.175e+07
QP-DC_{neg}-LP	**-1.193e+07**

(b) Final Energy achieved

Figure 2.1: *Evolution of achieved energies as a function of time on a stereo matching problem (Teddy Image). While the **QP** method leads to the worst result, using it as an initialisation greatly improves results. In the case of negative semi-definite potentials, the specific $\boldsymbol{DC_{neg}}$ method is as fast as mean-field, while additionally providing guarantees of monotonous decrease.(Best viewed in colour*

2.1.8.1 Stereo Matching

2.1.8.1.1 Data. We compare these methods on images extracted from the Middlebury stereo matching dataset [40]. The unary terms are obtained using the absolute difference matching function of [40]. The pixel compatibility function is similar to the one used by Krähenbühl and Koltun [5] and is described in the supplementary material. The label compatibility function is a Potts model.

2.1.8.1.2 Results. We present a comparison of runtime in Figure (2.1a), as well as the associated final energies for each method in Table (2.1b). Similar results for other problem instances can be found in the supplementary materials.

We observe that continuous relaxations obtain better energies than their mean-field counterparts. For a very limited time-budget, **MF** is the fastest method, although $\mathbf{DC}_{neg}$ is competitive and reach lower energies. When using **LP**, optimising a better objective function allows us to escape the local minima to which $\mathbf{DC_{neg}}$ converges. However, due to the higher complexity and the fact that we need to perform divide-and-conquer separately for all labels, the method is slower. This is particularly visible for problems with a high number of labels. This indicates that the LP relaxation might be better suited to fine-tune accurate solutions obtained by faster alternatives. For example, this can be achieved by restricting the LP

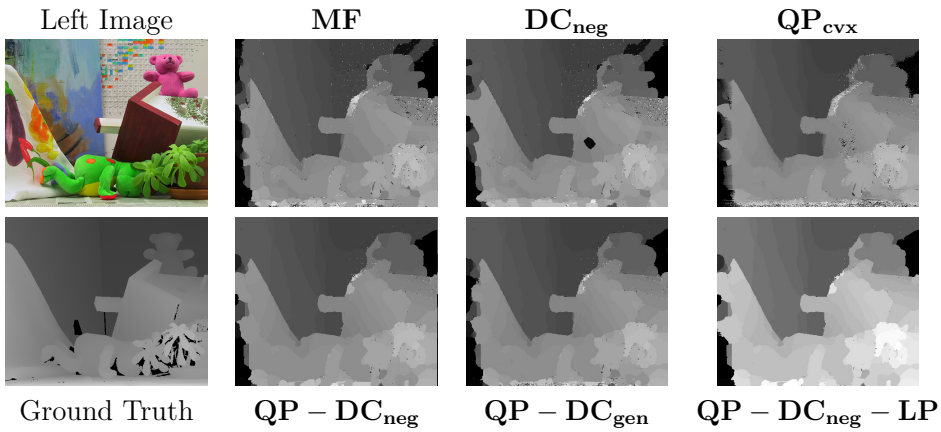


Figure 2.2: Stereo matching results on the Teddy image. Continuous relaxation achieve smoother labeling, as expected by their lower energies.

to optimise over a subset of relevant labels, that is, labels that are present in the solutions provided by other methods. Qualitative results for the Teddy image can be found in Figure 2.2 and additional outputs are present in supplementary material. We can see that lower energy translates to better visual results: note the removal of the artifacts in otherwise smooth regions (for example, in the middle of the sloped surface on the left of the image).

2.1.8.2 Image Segmentation

2.1.8.2.1 Data. We now consider an image segmentation task evaluated on the PASCAL VOC 2010 [41] dataset. For the sake of comparison, we use the same data splits and unary potentials as the one used by Krähenbühl and Koltun [5]. We perform cross-validation to select the best parameters of the pixel compatibility function for each method using Spearmint [42].

2.1.8.2.2 Results. The energy results obtained using the parameters cross validated for **DC$_{neg}$** are given in Table 2.1. **MF5** corresponds to mean-field ran for 5 iterations as it is often the case in practice [5, 25].

Once again, we observe that continuous relaxations provide lower energies than mean-field based approaches. To add significance to this result, we also compare energies image-wise. In all but a few cases, the energies obtained by the continuous relaxations are better or equal to the mean-field ones. This provides conclusive evidence for our central hypothesis that continuous relaxations are better suited to the problem of energy minimization in dense CRFs.

For completeness, we also provide energy and segmentation results for the parameters tuned for **MF** in the supplementary material. Even in that unfavourable

	Unary	**MF5**	**MF**	**QP_{cvx}**	**DC_{gen}**	**DC_{neg}**	**LP**	**Avg. E**	**Acc**	**IoU**
Unary	-	0	0	0	0	0	0	0	79.04	27.43
MF5	99	-	13	0	0	0	0	-600	79.13	27.53
MF	99	0	-	0	0	0	0	-600	79.13	27.53
QP_{cvx}	99	99	99	-	0	0	0	-6014	80.38	28.56
DC_{gen}	99	99	99	85	-	0	1	-6429	80.41	28.59
DC_{neg}	99	99	99	98	97	-	4	-6613	80.43	28.60
LP	99	99	99	98	97	87	-	**-6697**	**80.49**	**28.68**

Table 2.1: Percentage of images the row method outperforms the column method on final energy, average energy over the test set and Segmentation performance. Continuous relaxations dominate mean-field approaches on almost all images and improve significantly more compared to the Unary baseline. Parameters tuned for **DC_{neg}**.

setting, continuous relaxations still provide better energies. Note that, due to time constraints, we run the LP subgradient descent for only 5 iterations of subgradient descent. Moreover, to be able to run more experiments, we also restricted the number of labels by discarding labels that have a very small probability to appear given the initialisation.

Some qualitative results can be found in Figure 2.3. When comparing the segmentations for **MF** and **DC_{neg}**, we can see that the best one is always the one we tune parameters for. A further interesting caveat is that although we always find a solution with better energy, it does not appear to be reflected in the quality of the segmentation. While in the previous case with stereo vision, better energy implied qualitatively better reconstruction it is not so here. Similar observation was made by Wang et al [33].

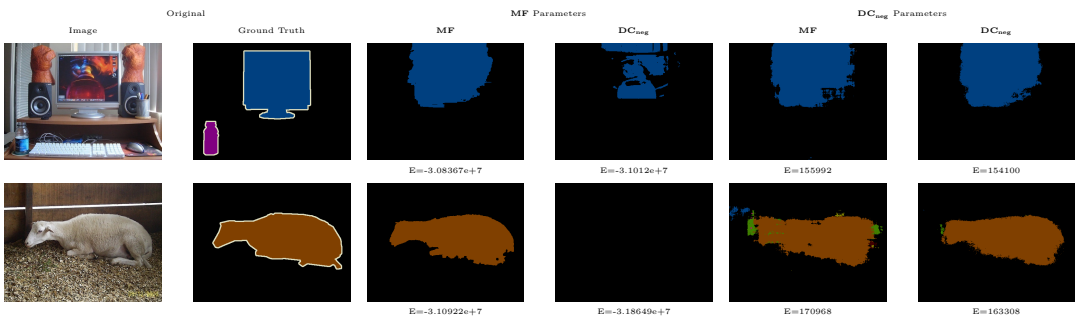


Figure 2.3: Segmentation results on sample images. We see that **DC_{neg}** leads to better energy in all cases compared to **MF**. Segmentation results are better for **MF** for the **MF**-tuned parameters and better for **DC_{neg}** for the **DC_{neg}**-tuned parameters.

2.1.9 Discussion

Our main contribution are four efficient algorithms for the dense CRF energy minimization problem based on QP, DC and LP relaxations. We showed that continuous relaxations give better energies than the mean-field based approaches. Our best performing method, the LP relaxation, suffers from its high runtime. To go beyond this limit, move making algorithms such as α-expansion [43] could be used and take advantage of the fact that this relaxation solves exactly the original IP for the two label problem. In future work, we also want to investigate the effect of learning specific parameters for these new inference methods using the framework of [31].

2.2 Efficient Linear Programming for Dense CRFs

2.2.1 Preamble

The remaining of this section contains a paper published at CVPR 2017, namely [44]. This work was done in collaboration with Thalaiyasingam Ajanthan which worked on the proximal minimization algorithm for the LP relaxation of Dense CRFs of Section 2.2.4.

The fully connected conditional random field (CRF) with Gaussian pairwise potentials has proven popular and effective for multi-class semantic segmentation. While the energy of a dense CRF can be minimized accurately using a linear programming (LP) relaxation, the state-of-the-art algorithm is too slow to be useful in practice. To alleviate this deficiency, we introduce an efficient LP minimization algorithm for dense CRFs. To this end, we develop a proximal minimization framework, where the dual of each proximal problem is optimized via block coordinate descent. We show that each block of variables can be efficiently optimized. Specifically, for one block, the problem decomposes into significantly smaller subproblems, each of which is defined over a single pixel. For the other block, the problem is optimized via conditional gradient descent. This has two advantages: 1) the conditional gradient can be computed in a time linear in the number of pixels and labels; and 2) the optimal step size can be computed analytically. Our experiments on standard datasets provide compelling evidence that our approach outperforms all existing baselines including the previous LP based approach for dense CRFs.

2.2.2 Introduction

In the past few years, the dense conditional random field (CRF) with Gaussian pairwise potentials has become popular for multi-class image-based semantic segmentation. At the origin of this popularity lies the use of an efficient filtering method [21], which was shown to lead to a linear time mean-field inference strategy [5]. Recently, this filtering method was exploited to minimize the dense CRF energy using other, typically more effective, continuous relaxation methods [14]. Among the relaxations considered in [14], the linear programming (LP) relaxation provides strong theoretical guarantees on the quality of the solution [16, 45].

In [14], the LP was minimized via projected subgradient descent. While relying on the filtering method, computing the subgradient was shown to be *linearithmic* in the number of pixels, but not *linear*. Moreover, even with the use of a line search strategy, the algorithm required a large number of iterations to converge, making it inefficient.

We introduce an iterative LP minimization algorithm for a dense CRF with Gaussian pairwise potentials which has *linear* time complexity per iteration. To this end, instead of relying on a standard subgradient technique, we propose to make use of the proximal method [46]. The resulting proximal problem has a smooth dual, which can be efficiently optimized using block coordinate descent. We show that each block of variables can be optimized efficiently. Specifically, for one block, the problem decomposes into significantly smaller subproblems, each of which is defined over a single pixel. For the other block, the problem can be optimized via the Frank-Wolfe algorithm [22, 35]. We show that the conditional gradient required by this algorithm can be computed efficiently. In particular, we modify the filtering method of [21] such that the conditional gradient can be computed in a time *linear* in the number of pixels and labels. Besides this linear complexity, our approach has two additional benefits. First, it can be initialized with the solution of a faster, less accurate algorithm, such as mean-field [5] or the difference of convex (DC) relaxation of [14], thus speeding up convergence. Second, the optimal step size of our iterative procedure can be obtained analytically, thus preventing the need to rely on an expensive line search procedure.

We demonstrate the effectiveness of our algorithm on the MSRC and Pascal VOC 2010 [41] segmentation datasets. The experiments evidence that our algorithm is significantly faster than the state-of-the-art LP minimization technique of [14]. Furthermore, it yields assignments whose energies are much lower than those obtained by other competing methods [5, 14]. Altogether, our framework constitutes the first efficient and effective minimization algorithm for dense CRFs with Gaussian pairwise potentials.

2.2.3 Preliminaries

Before introducing our method, let us first provide some background on the dense CRF model and its LP relaxation.

Dense CRF Energy Function. A dense CRF is defined on a set of n random variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, where each random variable X_a takes a label $x_a \in \mathcal{L}$, with $|\mathcal{L}| = m$. For a given labelling $\mathbf{x}$, the energy associated with a pairwise dense CRF can be expressed as

$$E(\mathbf{x}) = \sum_{a=1}^{n} \phi_a(x_a) + \sum_{a=1}^{n} \sum_{\substack{b=1 \\ b \neq a}}^{n} \psi_{ab}(x_a, x_b) \ , \tag{2.2.3.1}$$

where ϕ_a and ψ_{ab} denote the *unary potentials* and *pairwise potentials*, respectively. The unary potentials define the data cost and the pairwise potentials the smoothness cost.

Gaussian Pairwise Potentials. Similarly to [5, 14], we consider Gaussian pairwise potentials, which have the following form:

$$\begin{aligned} \psi_{ab}(x_a, x_b) &= \mu(x_a, x_b) \sum_c w^{(c)} \, k\left(\mathbf{f}_a^{(c)}, \mathbf{f}_b^{(c)}\right) \ , \\ k(\mathbf{f}_a, \mathbf{f}_b) &= \exp\left(\frac{-\|\mathbf{f}_a - \mathbf{f}_b\|^2}{2}\right) \ . \end{aligned} \tag{2.2.3.2}$$

Here, $\mu(x_a, x_b)$ is referred to as the *label compatibility* function and the mixture of Gaussian kernels as the *pixel compatibility* function. The weights $w^{(c)}$ define the mixture coefficients, and $\mathbf{f}_a^{(c)} \in \mathbb{R}^{\mathrm{d}^{(c)}}$ encodes features associated to the random variable X_a, where $d^{(c)}$ is the feature dimension. For semantic segmentation, each pixel in an image corresponds to a random variable. In practice, as in [5, 14], we then use the position and RGB values of a pixel as features, and assume the label compatibility function to be the Potts model, that is, $\mu(x_a, x_b) = \mathbf{1}[x_a \neq x_b]$. These potentials proved to be useful in obtaining fine grained labellings in segmentation tasks [5].

Integer Programming Formulation. An alternative way of representing a labelling is by defining indicator variables $y_{a:i} \in \{0, 1\}$, where $y_{a:i} = 1$ if and only if $x_a = i$. Using this notation, the energy minimization problem can be written as the following Integer Program (IP):

$$\begin{aligned} \min_{\mathbf{y}} \quad & E(\mathbf{y}) \ = \sum_a \sum_i \phi_{a:i} \, y_{a:i} + \sum_{a, b \neq a} \sum_{i,j} \psi_{ab:ij} \, y_{a:i} \, y_{b:j} \ , \\ \text{s.t.} \quad & \sum_i y_{a:i} = 1 \qquad \forall \, a \in \{1 \ldots n\} \ , \\ & y_{a:i} \quad \in \{0, 1\} \quad \forall \, a \in \{1 \ldots n\}, \quad \forall \, i \in \mathcal{L} \ . \end{aligned} \tag{2.2.3.3}$$

Here, we use the shorthand $\phi_{a:i} = \phi_a(i)$ and $\psi_{ab:ij} = \psi_{ab}(i,j)$. The first set of constraints ensure that each random variable is assigned exactly one label. Note that the value of objective function is equal to the energy of the labelling encoded by $\mathbf{y}$.

Linear Programming Relaxation. By relaxing the binary constraints of the indicator variables in (2.2.3.3) and using the fact that the label compatibility function is the Potts model, the linear programming relaxation [16] of (2.2.3.3) is defined as

$$\begin{aligned} \min_{\mathbf{y}} \quad & \tilde{E}(\mathbf{y}) \;= \sum_a \sum_i \phi_{a:i}\, y_{a:i} + \sum_{a,b\neq a} \sum_i K_{ab} \frac{|y_{a:i} - y_{b:i}|}{2} \;, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{M} = \left\{ \mathbf{y} \;\middle|\; \begin{array}{l} \sum_i y_{a:i} = 1,\; a \in \{1 \dots n\} \\ y_{a:i} \geq 0,\; a \in \{1 \dots n\},\; i \in \mathcal{L} \end{array} \right\} \;, \end{aligned} \tag{2.2.3.4}$$

where $K_{ab} = \sum_c w^{(c)}\, k\left(\mathbf{f}_a^{(c)}, \mathbf{f}_b^{(c)}\right)$. For integer labellings, the LP objective $\tilde{E}(\mathbf{y})$ has the same value as the IP objective $E(\mathbf{y})$. It is also worth noting that this LP relaxation is known to provide the best theoretical bounds [16]. Using standard solvers to minimize this LP would require the introduction of $\mathcal{O}(n^2)$ variables, making it intractable. Therefore the non-smooth objective of Eq. (2.2.3.4) has to be optimized directly. This was handled using projected subgradient descent in [14], which also turns out to be inefficient in practice. In this paper, we introduce an efficient algorithm to tackle this problem while maintaining *linear* scaling in both space and time complexity.

2.2.4 Proximal Minimization for LP Relaxation

Our goal is to design an efficient minimization strategy for the LP relaxation in (2.2.3.4). To this end, we propose to use the proximal minimization algorithm [46]. This guarantees monotonic decrease in the objective value, enabling us to leverage faster, less accurate methods for initialization. Furthermore, the additional quadratic regularization term makes the dual problem smooth, enabling the use of more sophisticated optimization methods. In the remainder of this paper, we detail this approach and show that each iteration has linear time complexity. In practice, our algorithm converges in a small number of iterations, thereby making the overall approach computationally efficient.

The proximal minimization algorithm [46] is an iterative method that, given the current estimate of the solution $\mathbf{y}^k$, solves the problem

$$\begin{aligned} \min_{\mathbf{y}} \quad & \tilde{E}(\mathbf{y}) + \frac{1}{2\lambda} \left\| \mathbf{y} - \mathbf{y}^k \right\|^2 \;, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{M} \;, \end{aligned} \tag{2.2.4.1}$$

where λ sets the strength of the proximal term.

Note that (2.2.4.1) consists of piecewise linear terms and a quadratic regularization term. Specifically, the piecewise linear term comes from the pairwise term $|y_{a:i} - y_{b:i}|$ in (2.2.3.4) that can be reformulated as $\max\{y_{a:i} - y_{b:i}, y_{b:i} - y_{a:i}\}$. The proximal term $\|\mathbf{y} - \mathbf{y}^k\|^2$ provides the quadratic regularization. In this section, we introduce a new algorithm that is tailored to this problem. In particular, we optimally solve the Lagrange dual of (2.2.4.1) in a block-wise fashion.

Algorithm 4: Proximal minimization of LP

Data: Initial solution $\mathbf{y}^0 \in \mathcal{M}$ and the dual objective g

for $k \leftarrow 0 \dots K$ **do**
  $A\boldsymbol{\alpha}^0 \leftarrow \mathbf{0}, \quad \boldsymbol{\beta}^0 \leftarrow \mathbf{0}, \quad \boldsymbol{\gamma}^0 \leftarrow \mathbf{0}$; `// Feasible initialization`
  for $t \leftarrow 0 \dots T$ **do**
    $(\boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) \leftarrow \operatorname{argmin}_{\beta,\gamma} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}, \boldsymbol{\gamma})$; `// Sec. 2.2.4.2.1`
    $\tilde{\mathbf{y}}^t \leftarrow \lambda(A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}) + \mathbf{y}^k$
    `/* Current primal solution may be infeasible */`
    $A\mathbf{s}^t \leftarrow$ conditional gradient of g, computed using $\tilde{\mathbf{y}}^t$; `// Sec. 2.2.4.2.2`
    $\delta \leftarrow$ optimal step size given $(\mathbf{s}^t, \boldsymbol{\alpha}^t, \tilde{\mathbf{y}}^t)$; `// Sec. 2.2.4.2.2`
    $A\boldsymbol{\alpha}^{t+1} \leftarrow (1-\delta)A\boldsymbol{\alpha}^t + \delta A\mathbf{s}^t$; `// Frank-Wolfe update on` $\boldsymbol{\alpha}$
  end
  `/* Project the primal solution to the feasible set` $\mathcal{M}$ `*/`
  $\mathbf{y}^{k+1} \leftarrow P_{\mathcal{M}}(\tilde{\mathbf{y}}^t)$
end

2.2.4.1 Dual Formulation

Let us first write the proximal problem (2.2.4.1) in the standard form by introducing auxiliary variables $z_{ab:i}$.

$$\min_{\mathbf{y},\mathbf{z}} \quad \sum_a \sum_i \phi_{a:i}\, y_{a:i} + \sum_{a,b\neq a} \sum_i \frac{K_{ab}}{2} z_{ab:i} + \frac{1}{2\lambda}\|\mathbf{y} - \mathbf{y}^k\|^2 \,, \tag{2.2.4.2a}$$

$$\text{s.t.} \quad z_{ab:i} \geq y_{a:i} - y_{b:i} \quad \forall\, a \neq b \quad \forall\, i \in \mathcal{L} \,, \tag{2.2.4.2b}$$

$$z_{ab:i} \geq y_{b:i} - y_{a:i} \quad \forall\, a \neq b \quad \forall\, i \in \mathcal{L} \,, \tag{2.2.4.2c}$$

$$\sum_i y_{a:i} = 1 \quad \forall\, a \in \{1 \dots n\} \,, \tag{2.2.4.2d}$$

$$y_{a:i} \geq 0 \quad \forall\, a \in \{1 \dots n\} \quad \forall\, i \in \mathcal{L} \,. \tag{2.2.4.2e}$$

We introduce three blocks of dual variables. Namely, $\boldsymbol{\alpha} = \{\alpha^1_{ab:i}, \alpha^2_{ab:i} \mid a \neq b, i \in \mathcal{L}\}$ for the constraints in Eqs. (2.2.4.2b) and (2.2.4.2c), $\boldsymbol{\beta} = \{\beta_a \mid a \in \{1 \dots n\}\}$ for Eq. (2.2.4.2d) and $\boldsymbol{\gamma} = \{\gamma_{a:i} \mid a \in \{1 \dots n\}, i \in \mathcal{L}\}$ for Eq. (2.2.4.2e), respectively.

The vector $\boldsymbol{\alpha}$ has $p = 2n(n-1)m$ elements. Here, we introduce two matrices that will be useful to write the dual problem compactly.

Definition 2.2.1. *Let $A \in \mathbb{R}^{\mathrm{nm}\times\mathrm{p}}$ and $B \in \mathbb{R}^{\mathrm{nm}\times\mathrm{n}}$ be two matrices such that*

$$\begin{aligned}(A\boldsymbol{\alpha})_{a:i} &= -\sum_{b\neq a}\left(\alpha^1_{ab:i} - \alpha^2_{ab:i} + \alpha^2_{ba:i} - \alpha^1_{ba:i}\right) \ , \\ (B\boldsymbol{\beta})_{a:i} &= \beta_a \ .\end{aligned} \tag{2.2.4.3}$$

We can now state our first proposition.

Proposition 2.2.1. *Given matrices $A \in \mathbb{R}^{\mathrm{nm}\times\mathrm{p}}$ and $B \in \mathbb{R}^{\mathrm{nm}\times\mathrm{n}}$ and dual variables $(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma})$.*

1. *The Lagrange dual of* (2.2.4.2a) *takes the following form:*

$$\min_{\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}} g(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}) = \frac{\lambda}{2}\|A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}\|^2 + \left\langle A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}, \mathbf{y}^k\right\rangle - \langle \mathbf{1}, \boldsymbol{\beta}\rangle \ , \tag{2.2.4.4}$$

$$\begin{aligned} s.t. \qquad \gamma_{a:i} \quad &\geq 0 \quad \forall\, a \in \{1\dots n\} \quad \forall\, i \in \mathcal{L} \ , \\ \boldsymbol{\alpha} \in \mathcal{C} \quad &= \left\{ \boldsymbol{\alpha} \;\middle|\; \begin{array}{l} \alpha^1_{ab:i} + \alpha^2_{ab:i} = \frac{K_{ab}}{2},\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \\ \alpha^1_{ab:i}, \alpha^2_{ab:i} \geq 0,\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \end{array} \right\} \ . \end{aligned}$$

2. *The primal variables $\mathbf{y}$ satisfy*

$$\mathbf{y} = \lambda\left(A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}\right) + \mathbf{y}^k \ . \tag{2.2.4.5}$$

Proof. In Appendix A.2.1.1. □

2.2.4.2 Algorithm

The dual problem (2.2.4.4), in its standard form, can only be tackled using projected gradient descent. However, by separating the variables based on the type of the feasible domains, we propose an efficient block coordinate descent approach. Each of these blocks are amenable to more sophisticated optimization, resulting in a computationally efficient algorithm. As the dual problem is strictly convex and smooth, the optimal solution is still guaranteed. For $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, the problem decomposes over the pixels, as shown in 2.2.4.2.1, therefore making it efficient. The minimization with respect to $\boldsymbol{\alpha}$ is over a compact domain, which can be efficiently tackled using the Frank-Wolfe algorithm [22, 35]. Our complete algorithm is summarized in Algorithm 4. In the following sections, we discuss each step in more detail.

2.2.4.2.1 Optimizing over $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ We first turn to the problem of optimizing over $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ while $\boldsymbol{\alpha}^t$ is fixed. Since the dual variable $\boldsymbol{\beta}$ is unconstrained, the minimum value of the dual objective g is attained when $\nabla_\beta g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}, \boldsymbol{\gamma}) = 0$.

Proposition 2.2.2. *If $\nabla_\beta g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}, \boldsymbol{\gamma}) = 0$, then $\boldsymbol{\beta}$ satisfy*

$$\boldsymbol{\beta} = B^T \left(A\boldsymbol{\alpha}^t + \boldsymbol{\gamma} - \boldsymbol{\phi} \right) / m \ . \tag{2.2.4.6}$$

Proof. More details on the simplification is given in Appendix A.2.1.2. □

Note that, now, $\boldsymbol{\beta}$ is a function of $\boldsymbol{\gamma}$. We therefore substitute $\boldsymbol{\beta}$ in (2.2.4.4) and minimize over $\boldsymbol{\gamma}$. Interestingly, the resulting problem can be optimized independently for each pixel, with each subproblem being an m dimensional quadratic program (QP) with nonnegativity constraints, where m is the number of labels.

Proposition 2.2.3. *The optimization over $\boldsymbol{\gamma}$ decomposes over pixels where for a pixel a, this QP has the form*

$$\begin{aligned} \min_{\boldsymbol{\gamma}_a} \quad & \frac{1}{2} \boldsymbol{\gamma}_a^T Q \boldsymbol{\gamma}_a + \left\langle \boldsymbol{\gamma}_a, Q \left((A\boldsymbol{\alpha}^t)_a - \boldsymbol{\phi}_a \right) + \mathbf{y}_a^k \right\rangle \ , \\ s.t. \quad & \boldsymbol{\gamma}_a \geq \mathbf{0} \ . \end{aligned} \tag{2.2.4.7}$$

Here, $\boldsymbol{\gamma}_a$ denotes the vector $\{\gamma_{a:i} \mid i \in \mathcal{L}\}$ and $Q = \lambda \left(I - \mathbf{1}/m \right) \in \mathbb{R}^{\mathrm{m} \times \mathrm{m}}$, with I the identity matrix and $\mathbf{1}$ the matrix of all ones.

Proof. In Appendix A.2.1.2. □

We use the algorithm of [47] to efficiently optimize every such QP. In our case, due to the structure of the matrix Q, the time complexity of an iteration is linear in the number of labels. Hence, the overall time complexity of optimizing over $\boldsymbol{\gamma}$ is $\mathcal{O}(nm)$. Once the optimal $\boldsymbol{\gamma}$ is computed for a given $\boldsymbol{\alpha}^t$, the corresponding optimal $\boldsymbol{\beta}$ is given by Eq. (2.2.4.6).

2.2.4.2.2 Optimizing over $\boldsymbol{\alpha}$ We now turn to the problem of optimizing over $\boldsymbol{\alpha}$ given $\boldsymbol{\beta}^t$ and $\boldsymbol{\gamma}^t$. To this end, we use the Frank-Wolfe algorithm [22], which has the advantage of being projection free. Furthermore, for our specific problem, we show that the required conditional gradient can be computed efficiently and the optimal step size can be obtained analytically.

Conditional Gradient Computation. The conditional gradient with respect to $\boldsymbol{\alpha}$ is obtained by solving the following linearization problem

$$\mathbf{s} = \underset{\hat{\mathbf{s}} \in \mathcal{C}}{\operatorname{argmin}} \left\langle \hat{\mathbf{s}}, \nabla_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) \right\rangle \ . \tag{2.2.4.8}$$

Here, $\nabla_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t)$ denotes the gradient of the dual objective function with respect to $\boldsymbol{\alpha}$ evaluated at $(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t)$.

Proposition 2.2.4. *The conditional gradient* $\mathbf{s}$ *satisfy*

$$(A\mathbf{s})_{a:i} = -\sum_b \left(K_{ab} \mathbf{1}[\tilde{y}^t_{a:i} \geq \tilde{y}^t_{b:i}] - K_{ab} \mathbf{1}[\tilde{y}^t_{a:i} \leq \tilde{y}^t_{b:i}] \right) \ , \tag{2.2.4.9}$$

where $\tilde{\mathbf{y}}^t = \lambda \left(A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi} \right) + \mathbf{y}^k$ *using Eq.* (2.2.4.5).

Proof. In Appendix A.2.1.3. □

Note that Eq. (2.2.4.9) has the same form as the LP subgradient (Eq. (20) in [14]). This is not a surprising result. In fact, it has been shown that, for certain problems, there exists a duality relationship between subgradients and conditional gradients [48]. To compute this subgradient, the state-of-the-art algorithm proposed in [14] has a time complexity linearithmic in the number of pixels. Unfortunately, since this constitutes a critical step of both our algorithm and that of [14], such a linearithmic cost greatly affects their efficiency. In Section 2.2.5, however, we show that this complexity can be reduced to linear, thus effectively leading to a speedup of an order of magnitude in practice.

Optimal Step Size. One of the main difficulties of using an iterative algorithm, whether subgradient or conditional gradient descent, is that its performance depends critically on the choice of the step size. Here, we can analytically compute the optimal step size that results in the maximum decrease in the objective for the given descent direction.

Proposition 2.2.5. *The optimal step size* δ *satisfy*

$$\delta = P_{[0,1]} \left(\frac{\langle A\boldsymbol{\alpha}^t - A\mathbf{s}^t, \tilde{\mathbf{y}}^t \rangle}{\lambda \| A\boldsymbol{\alpha}^t - A\mathbf{s}^t \|^2} \right) \ . \tag{2.2.4.10}$$

Here, $P_{[0,1]}$ *denotes the projection to the interval* $[0, 1]$*, that is, clipping the value to lie in* $[0, 1]$.

Proof. In Appendix A.2.1.4. □

Memory Efficiency. For a dense CRF, the dual variable $\boldsymbol{\alpha}$ requires $\mathcal{O}(n^2m)$ storage, which becomes infeasible since n is the number of pixels in an image. Note, however, that $\boldsymbol{\alpha}$ always appears in the product $\tilde{\boldsymbol{\alpha}} = A\boldsymbol{\alpha}$ in Algorithm 4. Therefore, we only store the variable $\tilde{\boldsymbol{\alpha}}$, which reduces the storage complexity to $\mathcal{O}(nm)$.

2.2.4.2.3 Summary To summarize, our method has four desirable qualities of an efficient iterative algorithm. First, it can benefit from an initial solution obtained by a faster but less accurate algorithm, such as mean-field or DC relaxation. Second, with our choice of a quadratic proximal term, the dual of the proximal problem can be efficiently optimized in a block-wise fashion. Specifically, the dual variables $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are computed efficiently by minimizing one small QP (of dimension the number of labels) for each pixel independently. The remaining dual variable $\boldsymbol{\alpha}$ is optimized using the Frank-Wolfe algorithm, where the conditional gradient is computed in linear time, and the optimal step size is obtained analytically. Overall, the time complexity of one iteration of our algorithm is $\mathcal{O}(nm)$. To the best of our knowledge, this constitutes the first LP minimization algorithm for dense CRFs that has linear time iterations. We denote this standard algorithm as PROX-LP.

2.2.5 Fast Conditional Gradient Computation

The algorithm described in the previous section assumes that the conditional gradient (Eq. (2.2.4.9)) can be computed efficiently. Note that Eq. (2.2.4.9) contains two terms that are similar up to sign and order of the label constraint in the indicator function. To simplify the discussion, let us focus on the first term and on a particular label i, which we will not explicitly write in the remainder of this section. The second term in Eq. (2.2.4.9) and the other labels can be handled in the same manner. With these simplifications, we need to efficiently compute an expression of the form

$$\forall\, a \in \{1 \dots n\}, \quad v'_a = \sum_b k(\mathbf{f}_a, \mathbf{f}_b)\, \mathbf{1}[y_a \geq y_b] \;, \tag{2.2.5.1}$$

with $y_a, y_b \in [0, 1]$ and $\mathbf{f}_a, \mathbf{f}_b \in \mathbb{R}^{\mathrm{d}}$ for all $a, b \in \{1 \dots n\}$.

The usual way of speeding up computations involving such Gaussian kernels is by using the efficient filtering method [21]. This approximate method has proven accurate enough for similar applications [5, 14]. In our case, due to the ordering constraint $\mathbf{1}[y_a \geq y_b]$, the symmetry is broken and the direct application of the filtering method is impossible. In [14], the authors tackled this problem using a divide-and-conquer strategy, which lead to a time complexity of $\mathcal{O}(d^2 n \log(n))$. In practice, this remains a prohibitively high run time, particularly since gradient

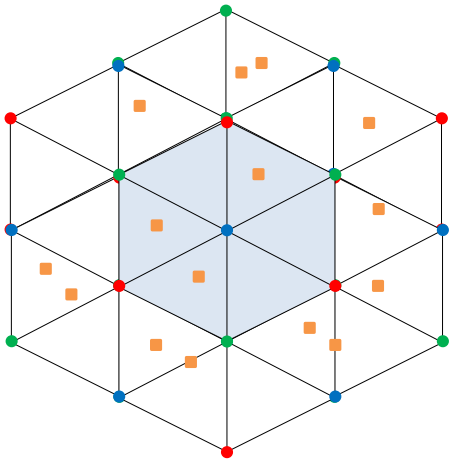


Figure 2.4: *A 2-dimensional hyperplane tessellated by the permutohedral lattice. The feature points are denoted with squares and the lattice points with circles. The neighborhood of the center lattice point is shaded and, for a feature point, the neighbouring lattice points are the vertices of the enclosing triangle.*

computations are performed many times over the course of the algorithm. Here, we introduce a more efficient method.

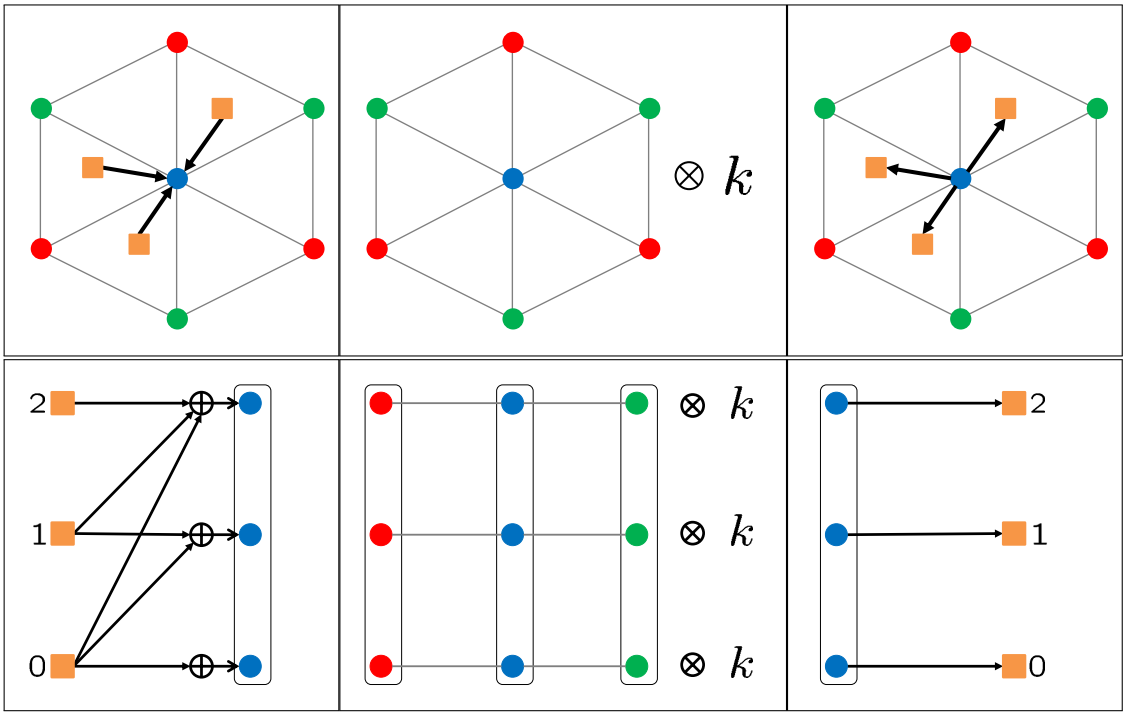
Specifically, we show that the term in Eq. (2.2.5.1) can be computed in $\mathcal{O}(Hdn)$ time (where H is a small constant defined in Section 2.2.5.2), at the cost of additional storage. In practice, this leads to a speedup of one order of magnitude. Below, we first briefly review the original filtering algorithm and then explain our modified algorithm that efficiently handles the ordering constraints.

2.2.5.1 Original Filtering Method

In this section, we assume that the reader is familiar with the permutohedral lattice based filtering method [21] andonly a brief overview is provided. We refer the interested reader to the original paper [21].

In [21], each pixel $a \in \{1 \dots n\}$ is associated with a tuple $(\mathbf{f}_a, v_a)$, which we call a *feature point*. The elements of this tuple are the feature $\mathbf{f}_a \in \mathbb{R}^{\mathrm{d}}$ and the value $v_a \in \mathbb{R}$. Note that, in our case, $v_a = 1$ for all pixels. At the beginning of the algorithm, the feature points are embedded in a d-dimensional hyperplane tessellated by the *permutohedral lattice* (see Fig. 2.4). The vertices of this permutohedral lattice are called *lattice points*, and each lattice point l is associated with a scalar value $\bar{v}_l$.

Once the permutohedral lattice is constructed, the algorithm performs three main steps: *splatting*, *blurring* and *slicing*. During splatting, for each lattice point, the values of the neighbouring feature points are accumulated using barycentric interpolation. Next, during blurring, the values of the lattice points are convolved with a one dimensional truncated Gaussian kernel along each feature dimension



(a) Splat **(b)** Blur **(c)** Slice

Figure 2.5: Top row: *Original filtering method. The barycentric interpolation is denoted by an arrow and k here is the truncated Gaussian kernel. During splatting, for each lattice point, the values of the neighbouring feature points are accumulated using barycentric interpolation. Next, the lattice points are blurred with k along each dimension. Finally, the values of the lattice points are given back to the feature points using the same barycentric weights.* **Bottom row:** *Our modified filtering method. Here, $H = 3$, and the figure therefore illustrates 3 lattices. We write the bin number of each feature point next to the point. Note that, at the splatting step, the value of a feature point is accumulated to its neighbouring lattice points only if it is above or equal to the feature point level. Then, blurring is performed at each level independently. Finally, the resulting values are recovered from the lattice points at the feature point level.*

separately. Finally, during slicing, the resulting values of the lattice points are propagated back to the feature points using the same barycentric weights. These steps are explained graphically in the top row of Fig. 2.5. The pseudocode of the algorithm is given in Appendix A.2.2.1. The time complexity of this algorithm is $\mathcal{O}(dn)$ [5, 21], and the complexity of the permutohedral lattice creation $\mathcal{O}(d^2n)$. Since the approach in [14] creates multiple lattices at every iteration, the overall complexity of this approach is $\mathcal{O}(d^2n\log(n))$.

Note that, in this original algorithm, there is no notion of score y_a associated with each pixel. In particular, during splatting, the values v_a are accumulated to the neighbouring lattice points without considering their scores. Therefore, this algorithm cannot be directly applied to handle our ordering constraint $\mathbf{1}[y_a \geq y_b]$.

2.2.5.2 Modified Filtering Method

We now introduce a filtering-based algorithm that can handle ordering constraints. To this end, we uniformly discretize the continuous interval $[0, 1]$ into H different discrete bins, or levels. Note that each pixel, or feature point, belongs to exactly one of these bins, according to its corresponding score. We then propose to instantiate H permutohedral lattices, one for each level $h \in \{0 \dots H-1\}$. In other words, at each level h, there is a lattice point l, whose value we denote by $\bar{v}_{l:h}$. To handle the ordering constraints, we then modify the splatting step in the following manner. A feature point belonging to bin q is splat to the permutohedral lattices corresponding to levels $q \leq h < H$. Blurring is then performed independently in each individual permutohedral lattice. This guarantees that a feature point will only influence the values of the feature points that belong to the same level or higher ones. In other words, a feature point b influences the value of a feature point a only if $y_a \geq y_b$. Finally, during the slicing step, the value of a feature point belonging to level q is recovered from the q^{th} permutohedral lattice. Our algorithm is depicted graphically in the bottom row of Fig. 2.5. Its pseudocode is provided in Appendix A.2.2.2. Note that, while discussed for constraints of the form $\mathbf{1}[y_a \geq y_b]$, this algorithm can easily be adapted to handle $\mathbf{1}[y_a \leq y_b]$ constraints, which are required for the second term in Eq. (2.2.4.9).

Overall, our modified filtering method has a time complexity of $\mathcal{O}(Hdn)$ and a space complexity of $\mathcal{O}(Hdn)$. Note that the complexity of the lattice creation is still $\mathcal{O}(d^2n)$ and can be reused for each of the H instances. Moreover, as opposed to the method in [14], this operation is performed only once, during the initialization step. In practice, we were able to choose H as small as 10, thus achieving a substantial speedup compared to the divide-and-conquer strategy of [14]. By discretizing the interval $[0, 1]$, we add another level of approximation to the overall algorithm. However, this approximation can be eliminated by using a dynamic data structure, which we briefly explain in Appendix A.2.2.2.1.

2.2.6 Related Work

We review the past work on three different aspects of our work in order to highlight our contributions.

Dense CRF. The fully-connected CRF has become increasingly popular for semantic segmentation. It is particularly effective at preventing oversmoothing, thus providing better accuracy at the boundaries of objects. As a matter of fact, in a complementary direction, many methods have now proposed to combine dense CRFs with convolutional neural networks [29, 30, 49] to achieve state-of-the-art performance on segmentation benchmarks.

The main challenge that had previously prevented the use of dense CRFs is their computational cost at inference, which, naively, is $\mathcal{O}(n^2)$ per iteration. In the case of Gaussian pairwise potential, the efficient filtering method of [21] proved to be key to the tractability of inference in the dense CRF. While an approximate method, the accuracy of the computation proved sufficient for practical purposes. This was first observed in [5] for the specific case of mean-field inference. More recently, several continuous relaxations, such as QP, DC and LP, were also shown to be applicable to minimizing the dense CRF energy by exploiting this filtering procedure in various ways [14]. Unfortunately, while tractable, minimizing the LP relaxation, which is known to provide the best approximation to the original labelling problem, remained too slow in practice [14]. Our algorithm is faster both theoretically and empirically. Furthermore, and as evidenced by our experiments, it yields lower energy values than any existing dense CRF inference strategy.

LP Relaxation. There are two ways to relax the integer program (2.2.3.3) to a linear program, depending on the label compatibility function: 1) the standard LP relaxation [50]; and 2) the LP relaxation specialized to the Potts model [16]. There are many notable works on minimizing the standard LP relaxation on sparse CRFs. This includes the algorithms that directly make use the dual of this LP [51–53] and those based on a proximal minimization framework [54, 55]. Unfortunately, all of the above algorithms exploit the sparsity of the problem, and they would yield an $\mathcal{O}(n^2)$ cost per iteration in the fully-connected case. In this work, we focus on the Potts model based LP relaxation for dense CRFs and provide an algorithm whose iterations have time complexity $\mathcal{O}(n)$. Even though we focus on the Potts model, as pointed out in [14], this LP relaxation can be extended to general label compatibility functions using a hierarchical Potts model [38].

Frank-Wolfe. The optimization problem of structural support vector machines (SVM) has a form similar to our proximal problem. The Frank-Wolfe algorithm [22] was shown to provide an effective and efficient solution to such a problem via block-coordinate optimization [35]. Several works have recently focused on improving the performance of this algorithm [56, 57] and extended its application domain [58]. Our work draws inspiration from this structural SVM literature, and makes use of the Frank-Wolfe algorithm to solve a subtask of our overall LP minimization method. Efficiency, however, could only be achieved thanks to our modification of the efficient filtering procedure to handle ordering constraints.

To the best of our knowledge, our approach constitutes the first LP minimization algorithm for dense CRFs to have linear time iterations. Our experiments demonstrate the importance of this result on both speed and labelling quality. Being fast, our algorithm can be incorporated in any end-to-end learning framework, such as [49]. We therefore believe that it will have a significant impact on future semantic segmentation results, and potentially in other application domains.

2.2.7 Experiments

In this section, we will first discuss two variants that further speedup our algorithm and some implementation details. We then turn to the empirical results.

2.2.7.1 Accelerated Variants

Empirically we observed that, our algorithm can be accelerated by restricting the optimization procedure to affect only relevant subsets of labels and pixels. These subsets can be identified from an intermediate solution of PROX-LP. In particular, we remove the label i from the optimization if $y_{a:i} < 0.01$ for all pixels a. In other words, the score of a label i is insignificant for all the pixels. We denote this version as PROX-LP$_\ell$. Similarly, we optimize over a pixel only if it is *uncertain* in choosing a label. Here, a pixel a is called uncertain if $\max_i y_{a:i} < 0.95$. In other words, no label has a score higher than 0.95. The intuition behind this strategy is that, after a few iterations of PROX-LP$_\ell$, most of the pixels are labelled correctly, and we only need to fine tune the few remaining ones. In practice, we limit this restricted set to 10% of the total number of pixels. We denote this accelerated algorithm as PROX-LP$_{\text{acc}}$. As shown in our experiments, PROX-LP$_{\text{acc}}$ yields a significant speedup at virtually no loss in the quality of the results.

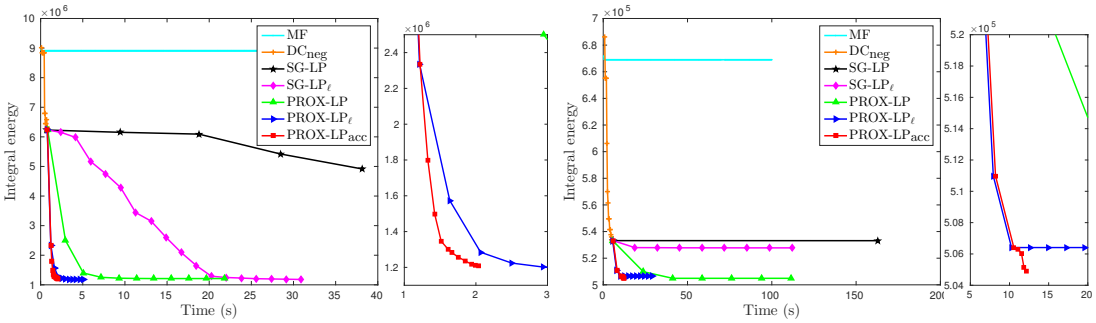


Figure 2.6: *Assignment energy as a function of time for DC_{neg} parameters for an image in (**left**) MSRC and (**right**) Pascal. A zoomed-in version is shown next to each plot. Except MF, all other algorithms are initialized with DC_{neg}. Note that PROX-LP clearly outperforms SG-LP$_\ell$ by obtaining much lower energies in fewer iterations. Furthermore, the accelerated versions of our algorithm obtain roughly the same energy as PROX-LP but significantly faster.*

2.2.7.2 Implementation Details

In practice, we initialize our algorithm with the solution of the best continuous relaxation algorithm, which is called DC_{neg} in [14]. The parameters of our algorithm, such as the proximal regularization constant λ and the stopping criteria, are chosen manually. A small value of λ leads to easier minimization of the proximal problem, but also yields smaller steps at each proximal iteration. We found $\lambda = 0.1$ to work well in all our experiments. We fixed the maximum number of proximal steps (K in Algorithm 4) to 10, and each proximal step is optimized for a maximum of 5 Frank-Wolfe iterations (T in Algorithm 4). In all our experiments the number of levels H is fixed to 10.

2.2.7.3 Segmentation Results

We evaluated our algorithm on the MSRC and Pascal VOC 2010 [41] segmentation datasets, and compare it against mean-field inference (MF) [5], the best performing continuous relaxation method of [14] (DC_{neg}) and the subgradient based LP minimization method of [14] (SG-LP). Note that, in [14], the LP was initialized with the DC_{neg} solution and optimized for 5 iterations. Furthermore, the LP optimization was performed on a subset of labels identified by the DC_{neg} solution in a similar manner to the one discussed in Section 2.2.7.1. We refer to this algorithm as SG-LP$_\ell$. For all the baselines, we employed the respective authors' implementations that were obtained from the web or through personal communication. Furthermore, for all the algorithms, the integral labelling is computed from the fractional solution using the *argmax* rounding scheme.

		MF5	MF	DC_{neg}	SG-LP_ℓ	PROX-LP	PROX-LP_ℓ	PROX-LP_{acc}	Ave. E ($\times 10^3$)	Ave. T (s)	Acc.	IoU
MSRC	MF5	-	0	0	0	0	0	0	8078.0	**0.2**	79.33	52.30
	MF	96	-	0	0	0	0	0	8062.4	0.5	79.35	52.32
	DC_{neg}	96	96	-	0	0	0	0	3539.6	1.3	83.01	57.92
	SG-LP_ℓ	96	96	90	-	3	1	1	3335.6	13.6	83.15	58.09
	PROX-LP	96	96	94	92	-	13	45	1274.4	23.5	83.99	**59.66**
	PROX-LP_ℓ	96	96	95	94	81	-	61	**1189.8**	6.3	83.94	59.50
	PROX-LP_{acc}	96	96	95	94	49	31	-	1340.0	3.7	**84.16**	59.65
Pascal	MF5	-	13	0	0	0	0	0	1220.8	0.8	79.13	27.53
	MF	2	-	0	0	0	0	0	1220.8	**0.7**	79.13	27.53
	DC_{neg}	99	99	-	-	0	0	0	629.5	3.7	80.43	28.60
	SG-LP_ℓ	99	99	95	-	5	12	12	617.1	84.4	80.49	**28.68**
	PROX-LP	99	99	95	84	-	32	50	507.7	106.7	80.63	28.53
	PROX-LP_ℓ	99	99	86	86	64	-	43	**502.1**	22.1	**80.65**	28.29
	PROX-LP_{acc}	99	99	86	86	46	39	-	507.7	14.7	80.58	28.45

Table 2.2: *Results on the MSRC and Pascal datasets with the parameters tuned for DC_{neg}. We show: the percentage of images where the row method strictly outperforms the column one on the final integral energy, the average integral energy over the test set, the average run time, the segmentation accuracy and the intersection over union score. Note that all versions of our algorithm obtain much lower energies than the baselines. Interestingly, while our fully accelerated version does slightly worse in terms of energy, it is the best in terms of the segmentation accuracy in MSRC.*

For both datasets, we used the same splits and unary potentials as in [5]. The pairwise potentials were defined using two kernels: a spatial kernel and a bilateral one [5]. For each method, the kernel parameters were cross validated on validation data using Spearmint [42]. To be able to compare energy values, we then evaluated all methods with the same parameters. In other words, for each dataset, each method was run several times with different parameter values. The final parameter values for MF and DC_{neg} are given in Appendix A.2.3.1. Note that, on MSRC, cross-validation was performed on the less accurate ground truth provided with the original dataset. Nevertheless, we evaluated all methods on the accurate ground truth annotations provided by [5].

The results for the parameters tuned for DC_{neg} on the MSRC and Pascal datasets are given in Table 2.2. Here MF5 denotes the mean-field algorithm run for 5 iterations. In Fig. 2.6, we show the assignment energy as a function of time for an image in MSRC (the tree image in Fig. 2.7) and for an image in Pascal (the sheep image in Fig. 2.7). Furthermore, we provide some of the segmentation results in Fig. 2.7.

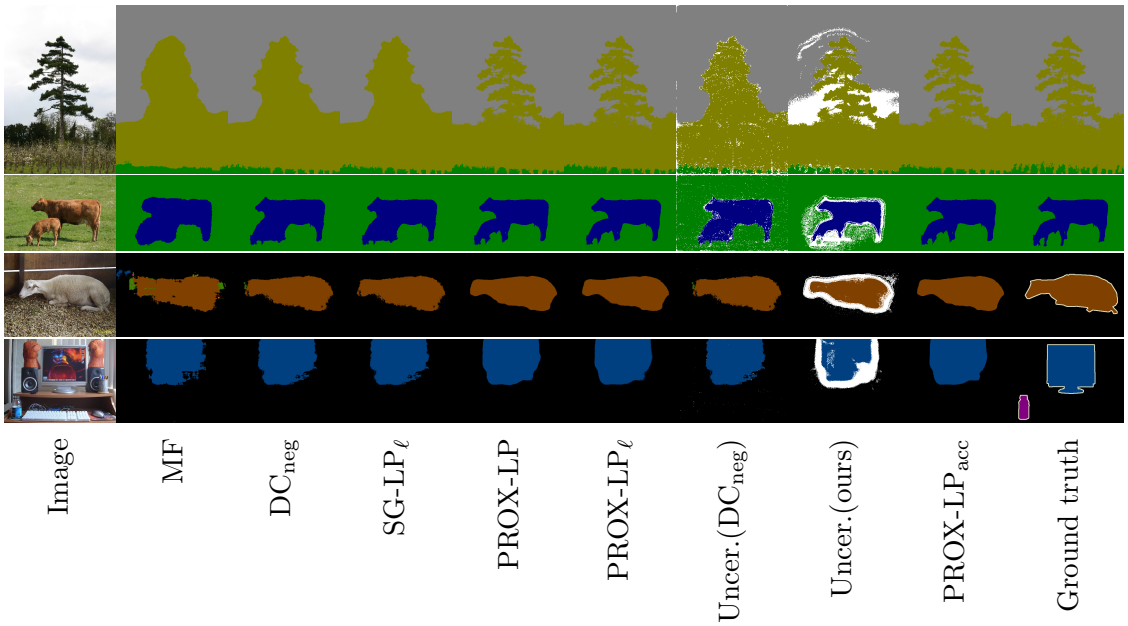


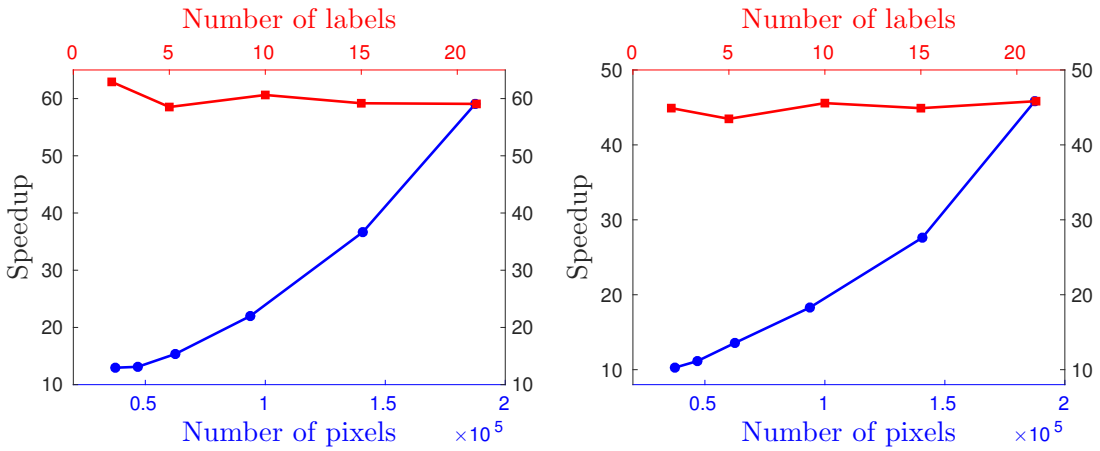
Figure 2.7: *Results with DC_{neg} parameters, for an image in (**top**) MSRC and (**bottom**) Pascal. The uncertain pixels identified by DC_{neg} and PROX-LP_{acc} are marked in white. Note that, all versions of our algorithm obtain visually good segmentations. In addition, even though DC_{neg} is less accurate (the percentatge of uncertain pixels for DC_{neg} is usually less than 1%) in predicting uncertain pixels, our algorithm marks most of the crucial pixels (object boundaries and shadows) as uncertain. Furthermore, in the MSRC images, the improvement of PROX-LP_{acc} over the baselines is clearly visible and the final segmentation is virtually the same as the accurate ground truth. (Best viewed in color)*

In summary, PROX-LP$_\ell$ obtains the lowest integral energy in both datasets. Furthermore, our fully accelerated version is the fastest LP minimization algorithm and always outperforms the baselines by a great margin in terms of energy. From Fig. 2.7, we can see that PROX-LP$_{\text{acc}}$ marks most of the crucial pixels (e.g., object boundaries) as *uncertain*, and optimizes over them efficiently and effectively. Note that, on top of being fast, PROX-LP$_{\text{acc}}$ obtains the highest accuracy in MSRC for the parameters tuned for DC$_{\text{neg}}$.

To ensure consistent behaviour across different energy parameters, we ran the same experiments for the parameters tuned for MF. In this setting, all versions of our algorithm again yield significantly lower energies than the baselines. The quantitative and qualitative results for this parameter setting are given in Appendix A.2.3.2.1.

2.2.7.4 Modified Filtering Method

We then compare our modified filtering method, described in Section 2.2.5, with the divide-and-conquer strategy of [14]. To this end, we evaluated both algorithms on one of the Pascal VOC test images (the sheep image in Fig. 2.7), but varying



(a) Spatial kernel ($d = 2$) **(b)** Bilateral kernel ($d = 5$)

Figure 2.8: *Speedup of our modified filtering algorithm over the divide-and-conquer strategy of [14] on a Pascal image. Note that our speedup grows with the number of pixels and is approximately constant with respect to the number of labels. (Best viewed in color)*

the image size, the number of labels and the Gaussian kernel standard deviation. Note that, to generate a plot for one variable, the other variables are fixed to their respective standard values. The standard value for the number of pixels is 187500, for the number of labels 21, and for the standard deviation 1. For this experiment, the conditional gradients were computed from a random primal solution $\tilde{\mathbf{y}}^t$. In Fig. 2.8, we show the speedup of our modified filtering approach over the one of [14] as a function of the number of pixels and labels. As shown in Appendix A.2.3.4, the speedup with respect to the kernel standard deviation is roughly constant. The timings were averaged over 10 runs, and we observed only negligible timing variations between the different runs.

In summary, our modified filtering method is $10 - 65$ times faster than the state-of-the-art algorithm of [14]. Furthermore, note that all versions of our algorithm operate in the region where the speedup is around $45 - 65$.

2.2.8 Discussion

We have introduced the first LP minimization algorithm for dense CRFs with Gaussian pairwise potentials whose iterations are linear in the number of pixels and labels. Thanks to the efficiency of our algorithm and to the tightness of the LP relaxation, our approach yields much lower energy values than state-of-the-art dense CRF inference methods. Furthermore, our experiments demonstrated that, with the right set of energy parameters, highly accurate segmentation results can be obtained with our algorithm. The speed and effective energy minimization of our algorithm make it a perfect candidate to be incorporated in an end-to-end learning framework, such as [49]. This, we believe, will be key to further improving the accuracy of deep semantic segmentation architectures.

3

Optimization by Machine Learning

Contents

3.1 Adaptive Neural Compilation

3.1.1 Preamble

The remaining of this section contains a paper published at NeurIPS 2016, namely [59]. This paper is a joint work with Rudy Bunel.

This paper proposes an adaptive neural-compilation framework to address the problem of learning efficient programs. Traditional code optimization strategies

used in compilers are based on applying pre-specified set of transformations that make the code faster to execute without changing its semantics. In contrast, our work involves adapting programs to make them more efficient while considering correctness only on a target input distribution. Our approach is inspired by the recent works on differentiable representations of programs. We show that it is possible to compile programs written in a low-level language to a differentiable representation. We also show how programs in this representation can be optimised to make them efficient on a target input distribution. Experimental results demonstrate that our approach enables learning *specifically-tuned* algorithms for given data distributions with a high success rate.

3.1.2 Introduction

Algorithm design often requires making simplifying assumptions about the input data. Consider, for instance, the computational problem of accessing an element in a linked list. Without the knowledge of the input data distribution, one can only specify an algorithm that runs in a time linear in the number of elements of the list. However, suppose all the linked lists that we encountered in practice were ordered in memory. Then it would be advantageous to design an algorithm specifically for this task as it can lead to a constant running time. Unfortunately, the input data distribution of a real world problem cannot be easily specified as in the above simple example. The best that one can hope for is to obtain samples drawn from the distribution. A natural question that arises from these observations: "How can we adapt a generic algorithm for a computational task using samples from an unknown input data distribution?"

The process of finding the most efficient implementation of an algorithm has received considerable attention in the theoretical computer science and code optimization community. Recently, Conditionally Correct Superoptimization [60] was proposed as a method for leveraging samples of the input data distribution to go beyond semantically equivalent optimization and towards data-specific performance improvements. The underlying procedure is based on a stochastic search over the space of all possible programs. Additionally, they restrict their applications to reasonably small, loop-free programs, thereby limiting their impact in practice.

In this work, we take inspiration from the recent wave of machine-learning frameworks for estimating programs. Using recurrent models, [61] introduced a fully differentiable representation of a program, enabling the use of gradient-based methods to learn a program from examples. Many other models that have been published recently [62–65] build and improve on the early work by [61].

Unfortunately, these models are usually complex to train and need to rely on methods such as curriculum learning or gradient noise to reach good solutions as shown by [66]. Moreover, their interpretability is limited. The learnt model is too complex for the underlying algorithm to be recovered and transformed into a regular computer program.

The main focus of the machine-learning community has thus far been on learning programs from scratch, with little emphasis on running time. However, for nearly all computational problems, it is feasible to design generic algorithms for the worst-case. We argue that a more pragmatic goal for the machine learning community is to design methods for adapting existing programs for specific input data distributions. To this end, we propose the Adaptive Neural Compiler (ANC). We design a compiler capable of mechanically converting algorithms to a differentiable representation, thereby providing adequate initialisation to the difficult problem of optimal program learning. We then present a method to improve this compiled program using data-driven optimization, alleviating the need to perform a wide search over the set of all possible programs. We show experimentally that this framework is capable of adapting simple generic algorithms to perform better on given datasets.

3.1.3 Related Works

The idea of compiling programs to neural networks has previously been explored in the literature. [67] described how to build a Neural Network that would perform the same operations as a given program. A compiler has been designed by [68] targeting an extended version of Pascal. A complete implementation was achieved when [69] wrote a compiler for NETDEF, a language based on the Occam programming language. While these methods allow us to obtain an exact representation of a program as a neural network, they do not lend themselves to optimization to improve the original program. Indeed, in their formulation, each elementary step of a program is expressed as a group of neurons with a precise topology, set of weights and biases, thereby rendering learning via gradient descent infeasible. Performing gradient descent in this parameter space would result in invalid operations and thus is unlikely to lead to any improvement. The recent work by [70] on Neural Programmer-Interpreters (NPI) can also be seen as a way to compile any program into a neural network. It does so by learning a model that mimics the program. While more flexible than previous approaches, the NPI only learns to reproduce an existing program. Therefore it cannot be used to find a new and possibly better program.

Another approach to this learning problem is the one taken by the code optimization community. By exploring the space of all possible programs, either

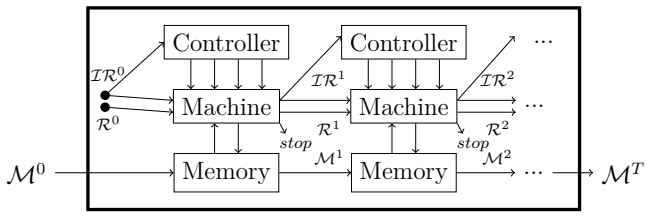
exhaustively [71] or in a stochastic manner [9], they search for programs having the same results but being more efficient. The work of [60] broadens the space of acceptable improvements to data-specific optimizations as opposed to the provably equivalent transformations that were previously the only ones considered. However, this method is still reliant on non-gradient-based methods for efficient exploration of the space. By representing everything in a differentiable manner, we aim to obtain gradients to guide the exploration.

Recently, [61] introduced a learnable representation of programs, called the Neural Turing Machine (NTM). The NTM uses an LSTM as a Controller, which outputs commands to be executed by a deterministic differentiable Machine. From examples of input/output sequences, they manage to learn a Controller such that the model becomes capable of performing simple algorithmic tasks. Extensions of this model have been proposed in [64, 65] where the memory tape was replaced by differentiable versions of stacks or lists. [62] modified the NTM to introduce a notion of pointers making it more amenable to represent traditional programs. Parallel works have been using Reinforcement Learning techniques such as the REINFORCE algorithm [72–74] or Q-learning [75] to be able to work with non differentiable versions of the above mentioned models. All these models are trained only with a loss based on the difference between the output of the model and the expected output. This weak supervision results in learning becoming more difficult. For instance the Neural RAM [62] requires a high number of random restarts before converging to a correct solution [66], even when using the best hyperparameters obtained through a large grid search.

In our work, we will first show that we can design a new neural compiler whose target will be a Controller-Machine model. This makes the compiled model amenable to learning from examples. Moreover, we can use it as initialisation for the learning procedure, allowing us to aim for the more complex task of finding an efficient algorithm.

3.1.4 Model

Our model is composed of two parts: *(i)* a Controller, in charge of specifying what should be executed; and *(ii)* a Machine, following the commands of the Controller. We start by describing the global architecture of the model. For the sake of simplicity, the general description will present a non-differentiable version of the model. Section 3.1.4.2 will then explain the modifications required to make this model completely differentiable. A more detailed description of the model is provided in the supplementary material.



(a) General view of the whole Model.

Inst	arg1	arg2	output	side effect
STOP	-	-	0	$stop = 1$
ZERO	-	-	0	-
INC	a	-	a+1	-
DEC	a	-	a-1	-
ADD	a	b	a+b	-
SUB	a	b	a-b	-
MIN	a	b	min(a,b)	-
MAX	a	b	max(a,b)	-
READ	a	-	m_a^t	Memory access
WRITE	a	b	0	$m_a^t = b$
JEZ	a	b	0	$\mathcal{IR}^t = \text{b}$ if a = 0

(b) Machine instructions.

Figure 3.1: Model components.

3.1.4.1 General Model

We first define for each timestep t the memory tape that contains M integer values $\mathcal{M}^t = \{m_1^t, m_2^t, \ldots, m_M^t\}$, registers that contain R values $\mathcal{R}^t = \{r_1^t, r_2^t, \ldots, r_R^t\}$ and the instruction register that contains a single value $\mathcal{IR}^t$. We also define a set of instructions that can be executed, whose main role is to perform computations using the registers. For example, add the values contained in two registers. We also define as a side effect any action that involves elements other than the input and output values of the instruction. Interaction with the memory is an example of such side effect. All the instructions, their computations and side effects are detailed in Figure 3.1b.

As can be seen in Figure 3.1a the execution model takes as input an initial memory tape $\mathcal{M}^0$ and outputs a final memory tape $\mathcal{M}^T$ after T steps. At each step t, the Controller uses the instruction register $\mathcal{IR}^t$ to compute the command for the Machine. The command is a 4-tuple e, a, b, o. The first element e is the instruction that should be executed by the Machine, enumerated as an integer. The elements a and b specify which registers should be used as arguments for the given instruction. The last element o specifies in which register the output of the instruction should be written. For example, the command $\{ADD, 2, 3, 1\}$ means that only the value of the first register should change, following $r_1^{t+1} = ADD(r_2^t, r_3^t)$. Then the Machine will execute this command, updating the values of the memory, the registers and the instruction register. The Machine always performs two other operations apart from the required instruction. It outputs a *stop* flag that allows the model to decide when to stop the execution. It also increments the instruction register $\mathcal{IR}^t$ by one at each iteration.

3.1.4.2 Differentiability

The model presented above is a simple execution machine but it is not differentiable. In order to be able to train this model end-to-end from a loss defined over the final memory tape, we need to make every intermediate operation differentiable.

To achieve this, we replace every discrete value in our model by a multinomial distribution over all the possible values that could have been taken. Moreover, each hard choice that would have been non-differentiable is replaced by a continuous soft choice. We will henceforth use bold letters to indicate the probabilistic version of a value.

First, the memory tape $\mathcal{M}^t$ is replaced by an $M \times M$ matrix $\mathbf{M}^t$, where $\mathbf{M}^t_{i,j}$ corresponds to the probability of m^t_i taking the value j. The same change is applied to the registers $\mathcal{R}^t$, replacing them with an $R \times M$ matrix $\mathbf{R}^t$, where $\mathbf{R}^t_{i,j}$ represents the probability of r^t_i taking the value j. Finally, the instruction register is also transformed, the same way as the other registers, from a single value $\mathcal{IR}^t$ to a vector of size M noted $\boldsymbol{\mathcal{IR}}^t$, where the i-th element represents its probability to take the value i.

The Machine does not contain any learnable parameter and will just execute a given command. To make it differentiable, the Machine now takes as input four probability distributions $\mathbf{e}^t$, $\mathbf{a}^t$, $\mathbf{b}^t$ and $\mathbf{o}^t$, where $\mathbf{e}^t$ is a distribution over instructions, and $\mathbf{a}^t, \mathbf{b}^t$ and $\mathbf{o}^t$ are distributions over registers. We compute the argument values $\mathbf{arg_1}^t$ and $\mathbf{arg_2}^t$ as convex combinations of delta-function probability distributions of the different registers values:

$$\mathbf{arg_1}^t = \sum_{i=1}^{R} \mathbf{a}^t_i \mathbf{r}^t_i \qquad \mathbf{arg_2}^t = \sum_{i=1}^{R} \mathbf{b}^t_i \mathbf{r}^t_i, \tag{3.1.4.1}$$

where $\mathbf{a}^t_i$ and $\mathbf{b}^t_i$ are the i-th values of the vectors $\mathbf{a}^t$ and $\mathbf{b}^t$. Using these values, we can compute the output value of each instruction k using the following formula:

$$\forall 0 \leq c \leq M \quad \mathbf{out}^t_{k,c} = \sum_{0 \leq i,j \leq M} \mathbf{arg_1}^t_i \cdot \mathbf{arg_2}^t_j \cdot \mathbb{1}[g_k(i,j) = c \mod M], \tag{3.1.4.2}$$

where g_k is the function associated to the k-th instruction as presented in Table 3.1b, $\mathbf{out}^t_{k,c}$ is the probability for an instruction k to output the value c at the time-step t and $\mathbf{arg_1}^t_i$ is the probability of the argument 1 having the value i at the time-step t. Since the executed instruction is controlled by the probability $\mathbf{e}$, the output for all instructions will also be a convex combination: $\mathbf{out}^t = \sum_{k=1}^{N} \mathbf{e}^t_k \mathbf{out}^t_k$, where N is the number of instructions. This value is then stored into the registers by performing a soft-write parametrised by $\mathbf{o}^t$: the value stored in the i-th register

at time $t+1$ is $\mathbf{r}_i^{t+1} = \mathbf{r}_i^t(1 - \mathbf{o}_i^t) + \mathbf{out}^t \mathbf{o}_i^t$, allowing the choice of the output register to be differentiable.

A special case is associated with the *stop* signal. When executing the model, we keep track of the probability that the program should have terminated before this iteration based on the probability associated at each iteration with the specific instruction that controls this flag. Once this probability goes over a threshold $\eta_{\text{stop}} \in (0, 1]$, the execution is halted. We applied the same techniques to make the side-effects differentiable, this is presented in the supplementary materials.

The Controller is the only learnable part of our model. The first learnable part is the initial values for the registers $\mathbf{R}^0$ and for the instruction register $\boldsymbol{\mathcal{IR}}^0$. The second learnable part is the parameters of the Controller which computes the required distributions using:

$$\mathbf{e}^t = \mathbf{W}_e * \boldsymbol{\mathcal{IR}}^t, \qquad \mathbf{a}^t = \mathbf{W}_a * \boldsymbol{\mathcal{IR}}^t, \qquad \mathbf{b}^t = \mathbf{W}_b * \boldsymbol{\mathcal{IR}}^t, \qquad \mathbf{o}^t = \mathbf{W}_o * \boldsymbol{\mathcal{IR}}^t \tag{3.1.4.3}$$

where $\mathbf{W}_e$ is an $N \times M$ matrix and $\mathbf{W}_a$, $\mathbf{W}_b$ and $\mathbf{W}_o$ are $R \times M$ matrices. A representation of these matrices can be found in Figure 3.2c. The Controller as defined above is composed of four independent, fully-connected layers. In Section 3.1.5.3 we will see that this complexity is sufficient for our model to be able to represent *any* program.

Henceforth, we will denote by $\boldsymbol{\theta} = \{\mathbf{R}^0, \boldsymbol{\mathcal{IR}}^0, \mathbf{W}_e, \mathbf{W}_a, \mathbf{W}_b, \mathbf{W}_o\}$ the set of learnable parameters.

3.1.5 Adaptative Neural Compiler

We will now present the Adaptive Neural Compiler. Its goal is to find the best set of weights $\boldsymbol{\theta}^*$ for a given dataset such that our model will perform the correct input/output mapping as efficiently as it can. We begin by describing our learning objective in details. The two subsequent sections will focus on making the optimization of our learning objective computationally feasible.

3.1.5.1 Objective Function

Our goal is to solve a given algorithmic problem efficiently. The algorithmic problem is defined as a set of input/output pairs. We also have access to a generic program that is able to perform the required mapping. In our example of accessing elements in a linked list, the transformation would consist in writing down the desired value at the specified position in the tape. The program given to us would iteratively go through the elements of the linked list, find the desired value and write it down

at the desired position. If there exists some bias that would allow this traversal to be faster, we expect the program to exploit it.

Our approach to this problem is to construct a differentiable objective function that maps controller parameters to a loss. We define this loss based on the states of the memory tape and outputs of the Controller at each step of the execution. The precise mathematical formulation for each term of the loss is given in the supplementary materials. Here we present the motivation behind each of them.

Correctness We first want the final memory tape to match the expected output for a given input.

Halting To prevent programs from taking an infinite amount of time without stopping, we define a maximum number of iterations T_{max} after which the execution is halted. Moreover, we add a penalty if the Controller didn't halt before this limit.

Efficiency We penalise each iteration taken by the program where it does not stop.

Confidence We finally make sure that if the Controller wants to stop, the current output is correct.

If only the correctness term was considered, nothing would encourage the learnt algorithm to halt as soon as it finished. If only correctness and halting were considered, then the program may not halt as early as possible. Confidence enables the algorithm to better evaluate when to stop.

The loss is a weighted sum of the four above-mentioned terms. We denote the loss of the i-th training sample, given parameters $\boldsymbol{\theta}$, as $L_i(\boldsymbol{\theta})$. Our learning objective is then specified as:

$$\min_{\boldsymbol{\theta}} \quad \sum_i L_i(\boldsymbol{\theta}) \quad \text{s.t. } \boldsymbol{\theta} \in \boldsymbol{\Theta}, \tag{3.1.5.1}$$

where Θ is a set over the parameters such that the outputs of the Controller, the initial values of each register and of the instruction register are all probability distributions.

The above optimization is a highly non-convex problem. In the rest of this section, we will first present a small modification to the model that will remove the constraints to be able to use standard gradient descent-based methods. Moreover, a good initialisation is helpful to solve these non-convex problems. To alleviate this deficiency, we now introduce our Neural Compiler that will provide a good initialisation.

3.1.5.2 Reformulation

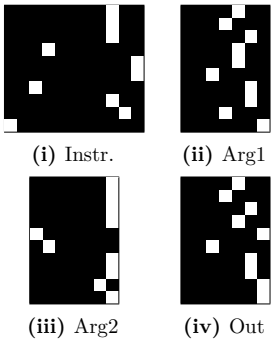
In order to use gradient descent methods without having to project the parameters on $\mathbf{\Theta}$, we alter the formulation of the Controller. We use softmax layers to be able to learn learn unormalized scores that are then mapped to probability distributions. We add one after each linear layer of the Controller and for the initial values of the registers. This way, we transform the constrained-optimization problem into an unconstrained one, allowing us to use standard gradient descent methods. As discussed in other works [66], this kind of model is hard to train and requires a high number of random restarts before converging to a good solution. We will now present a Neural Compiler that will provide good initialisations to help with this problem.

3.1.5.3 Neural Compiler

```
                                    Initial Registers:

                                    R1 = 6;  R2 = 2;  R3 = 0;
    var head = 0;                   R4 = 2;  R5 = 1;  R6 = 0;
    var nb_jump = 1;                R7 = 0;
    var out_write = 2;
                                    Program:

    nb_jump = READ(nb_jump);        0 : R5 = READ (R5, R7)
    out_write = READ(out_write);    1 : R4 = READ (R4, R7)
loop: head = READ(head);            2 : R6 = READ (R6, R7)
    nb_jump = DEC(nb_jump);         3 : R5 = DEC  (R5, R7)
    JEZ(nb_jump, end);              4 : R7 = JEZ  (R5, R1)
    JEZ(0, loop);                   5 : R3 = JEZ  (R3, R2)
end: head = INC(head);              6 : R6 = INC  (R6, R7)
    head = READ(head);              7 : R6 = READ (R6, R7)
    WRITE(out_write, head);         8 : R7 = WRITE(R4, R6)
    STOP();                         9 : R7 = STOP (R7, R7)
```



(a) Input program **(b)** Intermediary representation **(c)** Weights

Figure 3.2: Example of the compilation process. (3.2a) Program written to perform the **ListK** task. Given a pointer to the head of a linked list, an integer k, a target cell and a linked list, write in the target cell the k-th element of the list. (3.2b) Intermediary representation of the program. This corresponds to the instruction that a Random Access Machine would need to perform to execute the program. (3.2c) Representation of the weights that encodes the intermediary representation. Each row of the matrix correspond to one state/line. Initial value of the registers are also parameters of the model, omitted here.

The goal for the Neural Compiler is to convert an algorithm, written as an unambiguous program, to a set of parameters. These parameters, when put into the controller, will reproduce the exact steps of the algorithm. This is very similar to the problem framed by [70], but we show here a way to accomplish it *without any learning.*

The different steps of the compilation are illustrated in Figure 3.2. The first step is to go from the written version of the program to the equivalent list of low level instruction. This step can be seen as going from Figure 3.2a to Figure 3.2b. The illustrative example uses a fairly low-level language but traditional features of programming languages such as `loop`s or `if`-statements can be supported using the `JEZ` instruction. The use of constants as arguments or as values is handled by introducing new registers that hold these values. The value required to be passed as target position to the `JEZ` instruction can be resolved at compile time.

Having obtained this intermediate representation, generating the parameters is straightforward. As can be seen in Figure 3.2b, each line contains one instruction, the two input registers and the output register, and corresponds to a command that the Controller will have to output. If we ensure that $\boldsymbol{\mathcal{IR}}$ is a Dirac-delta distribution on a given value, then the matrix-vector product is equivalent to selecting a row of the weight matrix. As $\boldsymbol{\mathcal{IR}}$ is incremented at each iteration, the Controller outputs the rows of the matrix in order. We thus have a one-to-one mapping between the lines of the intermediate representation and the rows of the weight matrix. An example of these matrices can be found in Figure 3.2c. The weight matrix has 10 rows, corresponding to the number of lines of code of our intermediate representation. For example, on the first line of the matrix corresponding to the output (3.2civ), we see that the fifth element has value 1. This is linked to the first line of code where the output of the READ operation is stored into the fifth register. With this representation, we can note that the number of parameters is linear in the number of lines of code in the original program and that the largest representable number in our Machine needs to be greater than the number of lines in our program.

Moreover, any program written in a regular assembly language can be rewritten to use only our restricted set of instructions. This can be done firstly because all the conditionals of the assembly language can be expressed as a combination of arithmetic and `JEZ` instructions. Secondly because all the arithmetic operations can be represented as a combination of our simple arithmetic operations, `loop`s and `if`s statements. This means that *any* program that can run on a regular computer, can be first rewritten to use our restricted set of instructions and then compiled down to a set of weights for our model. Even though other models use LSTM as controller, we showed here that a Controller composed of simple linear functions is expressive enough. The advantage of this simpler model is that we can now easily interpret the weights of our model in a way that is not possible if we use a recurrent network as a controller.

The most straightforward way to leverage the results of the compilation is to initialise the Controller with the weights obtained through compilation of the generic algorithm. To account for the extra softmax layer, we need to multiply the weights produced by the compiler by a large constant to output Dirac-delta distributions. Some results associated with this technique can be found in Section 3.1.6.1. However, if we initialise with exactly this sharp set of parameters, the training procedure is not able to move away from the initialisation as the gradients associated with the softmax in this region are very small. Instead, we initialise the controller with a non-ideal version of the generic algorithm. This means that the choice with the highest probability in the output of the Controller is correct, but the probability of other choices is not zero. As can be seen in Section 3.1.6.2, this allows the Controller to learn by gradient descent a new algorithm, different from the original one, that has a lower loss than the ideal version of the compiled program.

3.1.6 Experiments

We performed two sets of experiments. The first shows the capability of the Neural Compiler to perfectly reproduce any given program. The second shows that our Neural Compiler can adapt and improve the performance of programs. We present results of data-specific optimization being carried out and show decreases in runtime for all the algorithms and additionally, for some algorithms, show that the runtime is a different computational-complexity class altogether. All the code required to reproduce these experiments is available online [1].

3.1.6.1 Compilation

The compiler described in section 3.1.5.3 allows us to go from a program written using our instruction set to a set of weights $\mathbf{\Theta}$ for our Controller.

To illustrate this point, we implemented simple programs that can solve the tasks introduced by [62] and a shortest path problem. One of these implementations can be found in Figure 3.2a, while the others are available in the supplementary materials. These programs are written in a specific language, and are transformed by the Neural Compiler into parameters for the model. As expected, the resulting models solve the original tasks exactly and can generalise to any input sequence.

[1] `https://github.com/albanD/adaptive-neural-compilation`

3.1.6.2 ANC Experiments

In addition to being able to reproduce any given program as was done by [70], we have the possibility of optimising the resulting program further. We exhibit this by compiling program down to our model and optimising their performance. The efficiency gain for these tasks come either from finding simpler, equivalent algorithms or by exploiting some bias in the data to either remove instructions or change the underlying algorithm.

We identify three different levels of interpretability for our model: the first type corresponds to weights containing only Dirac-delta distributions, there is an exact one-to-one mapping between lines in the weight matrices and lines of assembly code. In the second type where all probabilities are Dirac-delta except the ones associated with the execution of the `JEZ` instruction, we can recover an exact algorithm that will use `if` statements to enumerate the different cases arising from this conditional jump. In the third type where any operation other than `JEZ` is executed in a soft way or use a soft argument, it is not possible to recover a program that will be as efficient as the learned one.

We present here briefly the considered tasks and biases, and report the reader to the supplementary materials for a detailed encoding of the input/output tape.

1. **Access**: Given a value k and an array A, return $A[k]$. In the biased version, the value of k is always the same, so the address of the required element can be stored in a constant. This is similar to the optimization known as constant folding.

2. **Swap**: Given an array A and two pointers p and q, swap the elements $A[p]$ and $A[q]$. In the biased version, p and q are always the same so reading them can be avoided.

3. **Increment**: Given an array, increment all its element by 1. In the biased version, the array is of fixed size and the elements of the array have the same value so you do not need to read all of them when going through the array.

4. **Listk**: Given a pointer to the head of a linked list, a number k and a linked list, find the value of the k-th element. In the biased version, the linked list is organised in order in memory, as would be an array, so the address of the k-th value can be computed in constant time. This is the example developed in Figure 3.2.

Table 3.1: Average number of iterations required to solve instances of the problems for the original program, the best learned program and the ideal algorithm for the biased dataset. We also include the success rate of reaching a more efficient algorithm across multiple random restarts.

	Access	**Increment**	**Swap**	**ListK**	**Addition**	**Sort**
Generic	6	40	10	18	20	38
Learned	4	16	6	11	9	18
Ideal	4	34	6	10	6	9.5
Success Rate	37 %	84%	27%	19%	12%	74%

5. **Addition**: Two values are written on the tape and should be summed. No data bias is introduced but the starting algorithm is non-efficient: it performs the addition as a series of increment operation. The more efficient operation would be to add the two numbers.

6. **Sort**: Given an array A, sort it. In the biased version, only the start of the array might be unsorted. Once the start has been arranged, the end of the array can be safely ignored.

For each of these tasks, we perform a grid search on the loss parameters and on our hyper-parameters. Training is performed using Adam [76] and success rates are obtained by running the optimization with 100 different random seeds. We consider that a program has been successfully optimised when two conditions are fulfilled. First, it needs to output the correct solution for all test cases presenting the same bias. Second, the average number of iterations taken to solve a problem must have decreased. Note that if we cared only about the first criterion, the methods presented in Section 3.1.6.1 would already provide a success rate of 100%, without requiring any training.

The results are presented in Table 3.1. For each of these tasks, we manage to find faster algorithms. In the simple cases of **Access** and **Swap**, the *optimal* algorithms for the considered bias are obtained. They are found by incorporating heuristics to the algorithm and storing constants in the initial values of the registers. The learned programs for these tasks are always in the first case of interpretability, this means that we can recover the most efficient algorithm from the learned weights.

While **ListK** and **Addition** have lower success rates, improvements between the original and learned algorithms are still significant. Both were initialised with iterative algorithms with $\mathcal{O}(n)$ complexities. They managed to find constant time $\mathcal{O}(1)$ algorithms to solve the given problems, making the runtime independent of the input. Achieving this means that the equivalence between the two approaches has

been identified, similar to how optimising compilers operate. Moreover, on the **ListK** task, some learned programs corresponds to the second type of interpretability. Indeed these programs use soft jumps to condition the execution on the value of k. Even though these program would not generalise to other values of k, some learned programs for this task achieve a type one interpretability and a study of the learned algorithm reveal that they can generalise to *any* value of k.

Finally, the **Increment** task achieves an unexpected result. Indeed, it is able to outperform our best possible algorithm. By looking at the learned program, we can see that it is actually leveraging the possibility to perform soft writes over multiple elements of the memory at the same time to reduce its runtime. This is the only case where we see a learned program associated with the third type of interpretability. While our ideal algorithm would give a confidence of 1 on the output, this algorithm is unable to do so, but it has a high enough confidence of 0.9 to be considered a correct algorithm.

In practice, for all but the most simple tasks, we observe that further optimization is possible, as some useless instructions remain present. Some transformations of the controller are indeed difficult to achieve through the local changes operated by the gradient descent algorithm. An analysis of these failure modes of our algorithm can be found in the supplementary materials. This motivates us to envision the use of approaches other than gradient descent to address these issues.

3.1.7 Discussion

The work presented here is a first step towards adaptive learning of programs. It opens up several interesting directions of future research. For exemple, the definition of efficiency that we considered in this paper is flexible. We chose to only look at the average number of operations executed to generate the output from the input. We leave the study of other potential measures such as Kolmogorov Complexity and `sloc`, to name a few, for future works.

As shown in the experiment section, our current method is very good at finding efficient solutions for simple programs. For more complex programs, only a solution close to the initialisation can be found. Even though training heuristics could help with the tasks considered here, they would likely not scale up to real applications. Indeed, the main problem we identified is that the gradient-descent based optimization is unable to explore the space of programs effectively, by performing only local transformations. In future work, we want to explore different optimization methods. One approach would be to mix global and local exploration to improve the quality of the solutions. A more ambitious plan would

be to leverage the structure of the problem and use techniques from combinatorial optimization to try and solve the original discrete problem.

3.2 Learning to Superoptimize Programs

3.2.1 Preamble

The remaining of this section contains a paper published at ICLR 2017, namely [77]. This paper is a joint work with Rudy Bunel.

Code super-optimization is the task of transforming any given program to a more efficient version while preserving its input-output behaviour. In some sense, it is similar to the paraphrase problem from natural language processing where the intention is to change the syntax of an utterance without changing its semantics. Code-optimization has been the subject of years of research that has resulted in the development of rule-based transformation strategies that are used by compilers. More recently, however, a class of stochastic search based methods have been shown to outperform these strategies. This approach involves repeated sampling of modifications to the program from a proposal distribution, which are accepted or rejected based on whether they preserve correctness and the improvement they achieve. These methods, however, neither learn from past behaviour nor do they try to leverage the semantics of the program under consideration. Motivated by this observation, we present a novel learning based approach for code super-optimization. Intuitively, our method works by learning the proposal distribution using unbiased estimators of the gradient of the expected improvement. Experiments on benchmarks comprising of automatically generated as well as existing ("Hacker's Delight") programs show that the proposed method is able to significantly outperform state of the art approaches for code super-optimization.

3.2.2 Introduction

Considering the importance of computing to human society, it is not surprising that a very large body of research has gone into the study of the syntax and semantics of programs and programming languages. Code super-optimization is an extremely important problem in this context. Given a program or a snippet of source-code, super-optimization is the task of transforming it to a version that has the same input-output behaviour but can be executed on a target compute architecture more efficiently. Superoptimization provides a natural benchmark for evaluating representations of programs. As a task, it requires the decoupling of the semantics

of the program from its superfluous properties, the exact implementation. In some sense, it is the natural analogue of the paraphrase problem in natural language processing where we want to change syntax without changing semantics.

Decades of research has been done on the problem of code optimization resulting in the development of sophisticated rule-based transformation strategies that are used in compilers to allow them to perform code optimization. While modern compilers implement a large set of rewrite rules and are able to achieve impressive speed-ups, they fail to offer any guarantee of optimality, thus leaving room for further improvement. An alternative approach is to search over the space of all possible programs that are equivalent to the compiler output, and select the one that is the most efficient. If the search is carried out in a brute-force manner, we are guaranteed to achieve super-optimization. However, this approach quickly becomes computationally infeasible as the number of instructions and the length of the program grows.

In order to efficiently perform super-optimization, recent approaches have started to use a stochastic search procedure, inspired by Markov Chain Monte Carlo (MCMC) sampling [9]. Briefly, the search starts at an initial program, such as the compiler output. It iteratively suggests modifications to the program, where the probability of a modification is encoded in a proposal distribution. The modification is either accepted or rejected with a probability that is dependent on the improvement achieved. Under certain conditions on the proposal distribution, the above procedure can be shown, in the limit, to sample from a distribution over programs, where the probability of a program is related to its quality. In other words, the more efficient a program, the more times it is encountered, thereby enabling super-optimization. Using this approach, high-quality implementations of real programs such as the Montgomery multiplication kernel from the OpenSSL library were discovered. These implementations outperformed the output of the `gcc` compiler and even expert-handwritten assembly code.

One of the main factors that governs the efficiency of the above stochastic search is the choice of the proposal distribution. Surprisingly, the state of the art method, Stoke [9], employs a proposal distribution that is neither learnt from past behaviour nor does it depend on the syntax or semantics of the program under consideration. We argue that this choice fails to fully exploit the power of stochastic search. For example, consider the case where we are interested in performing bitwise operations, as indicated by the compiler output. In this case, it is more likely that the optimal program will contain bitshifts than floating point opcodes. Yet, Stoke will assign an equal probability of use to both types of opcodes.

In order to alleviate the aforementioned deficiency of Stoke, we build a reinforcement learning framework to estimate the proposal distribution for optimizing the source code under consideration. The score of the distribution is measured as the expected quality of the program obtained via stochastic search. Using training data, which consists of a set of input programs, the parameters are learnt via the REINFORCE algorithm [72]. We demonstrate the efficacy of our approach on two datasets. The first is composed of programs from "Hacker's Delight" [78]. Due to the limited diversity of the training samples, we show that it is possible to learn a prior distribution (unconditioned on the input program) that outperforms the state of the art. The second dataset contains automatically generated programs that introduce diversity in the training samples. We show that, in this more challenging setting, we can learn a conditional distribution given the initial program that significantly outperforms Stoke.

3.2.3 Related Works

Super-optimization The earliest approaches for super-optimization relied on brute-force search. By sequentially enumerating all programs in increasing length orders [71, 79], the shortest program meeting the specification is guaranteed to be found. As expected, this approach scales poorly to longer programs or to large instruction sets. The longest reported synthesized program was 12 instructions long, on a restricted instruction set [71].

Trading off completeness for efficiency, stochastic methods [9] reduced the number of programs to test by guiding the exploration of the space, using the observed quality of programs encountered as hints. In order to improve the size of solvable instances, [80] combined stochastic optimizers with smart enumerative solvers. However, the reliance of stochastic methods on a generic unspecific exploratory policy made the optimization blind to the problem at hand. We propose to tackle this problem by learning the proposal distribution.

Neural Computing Similar work was done in the restricted case of finding efficient implementation of computation of value of degree k polynomials [81]. Programs were generated from a grammar, using a learnt policy to prioritise exploration. This particular approach of guided search looks promising to us, and is in spirit similar to our proposal, although applied on a very restricted case.

Another approach to guide the exploration of the space of programs was to make use of the gradients of differentiable relaxation of programs. [59] attempted this by simulating program execution using Recurrent Neural Networks. However,

this provided no guarantee that the network parameters were going to correspond to real programs. Additionally, this method only had the possibility of performing local, greedy moves, limiting the scope of possible transformations. On the contrary, our proposed approach operates directly on actual programs and is capable of accepting short-term detrimental moves.

Learning to Optimize Outside of program optimization, applying learning algorithms to improve optimization procedures, either in terms of results achieved or runtime, is a well studied subject. [82] proposed imitation learning based methods to deal with structured output spaces, in a "Learning to search" framework. While this is similar in spirit to stochastic search, our setting differs in the crucial aspect of having a valid cost function instead of searching for one.

More relevant is the recent literature on learning to optimize. [83] and [73] learn how to improve on first-order gradient descent algorithms, making use of neural networks. Our work is similar, as we aim to improve the optimization process. However, as opposed to the gradient descent that they learn on a continuous unconstrained space, our initial algorithm is an MCMC sampler on a discrete domain.

Similarly, training a proposal distribution parameterized by a Neural Network was also proposed by [84] to accelerate inference in graphical models. Similar approaches were successfully employed in computer vision problems where data driven proposals allowed to make inference feasible [85–87]. Other approaches to speeding up MCMC inference include the work of [88], combining it with Variational inference.

3.2.4 Learning Stochastic Super-optimization

3.2.4.1 Stochastic Search as a Program Optimization Procedure

Stoke [9] performs black-box optimization of a cost function on the space of programs, represented as a series of instructions. Each instruction is composed of an opcode, specifying what to execute, and some operands, specifying the corresponding registers. Each given input program $\mathcal{T}$ defines a cost function. For a candidate program $\mathcal{R}$ called *rewrite*, the goal is to optimize the following cost function:

$$\text{cost}\,(\mathcal{R}, \mathcal{T}) = \omega_e \times \text{eq}(\mathcal{R}, \mathcal{T}) + \omega_p \times \text{perf}(\mathcal{R}) \tag{3.2.4.1}$$

The term $\text{eq}(\mathcal{R}; \mathcal{T})$ measures how well the outputs of the rewrite match the outputs of the reference program. This can be obtained either exactly by running a symbolic validator or approximately by running test cases. The term $\text{perf}(\mathcal{R})$ is a measure

of the efficiency of the program. In this paper, we consider runtime to be the measure of this efficiency. It can be approximated by the sum of the latency of all the instructions in the program. Alternatively, runtime of the program on some test cases can be used.

To find the optimum of this cost function, Stoke runs an MCMC sampler using the Metropolis [89] algorithm. This allows us to sample from the probability distribution induced by the cost function:

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp(-\text{cost}\,(\mathcal{R},\mathcal{T}))). \tag{3.2.4.2}$$

The sampling is done by proposing random moves from a different proposal distribution:

$$\mathcal{R}' \sim q(\,\cdot\,|\mathcal{R}). \tag{3.2.4.3}$$

The cost of the new modified program is evaluated and an acceptance criterion is computed. This acceptance criterion

$$\alpha(\mathcal{R},\mathcal{T}) = \min\left(1, \frac{p(\mathcal{R}';\mathcal{T})}{p(\mathcal{R};\mathcal{T})}\right), \tag{3.2.4.4}$$

is then used as the parameter of a Bernoulli distribution from which an accept/reject decision is sampled. If the move is accepted, the state of the optimizer is updated to $\mathcal{R}'$. Otherwise, it remains in $\mathcal{R}$.

While the above procedure is only guaranteed to sample from the distribution $p(\,\cdot\,;\mathcal{T})$ in the limit if the proposal distribution q is symmetric ($q(\mathcal{R}'|\mathcal{R}) = q(\mathcal{R}|\mathcal{R}')$ for all $\mathcal{R}, \mathcal{R}'$), it still allows us to perform efficient hill-climbing for non-symmetric proposal distributions. Moves leading to an improvement are always going to be accepted, while detrimental moves can still be accepted in order to avoid getting stuck in local minima.

3.2.4.2 Learning to Search

We now describe our approach to improve stochastic search by learning the proposal distribution. We begin our description by defining the learning objective (section 3.2.4.2.1), followed by a parameterization of the proposal distribution (section 3.2.4.2.2), and finally the reinforcement learning framework to estimate the parameters of the proposal distribution (section 3.2.4.2.3).

3.2.4.2.1 Objective Function Our goal is to optimize the cost function defined in Eq. (3.2.4.1). Given a fixed computational budget of T iterations to perform program super-optimization, we want to make moves that lead us to the lowest possible cost. As different programs have different runtimes and therefore different associated costs, we need to perform normalization. As normalized loss function, we use the ratio between the best rewrite found and the cost of the initial unoptimized program $\mathcal{R}_0$. Formally, the loss for a set of rewrites $\{\mathcal{R}_t\}_{t=0..T}$ is defined as follows:

$$r(\{\mathcal{R}_t\}_{t=0..T}) = \left(\frac{\min_{t=0..T} \text{cost}\left(\mathcal{R}_t, \mathcal{T}\right)}{\text{cost}\left(\mathcal{R}_0, \mathcal{T}\right)}\right). \tag{3.2.4.5}$$

Recall that our goal is to learn a proposal distribution. Given that our optimization procedure is stochastic, we will need to consider the expected cost as our loss. This expected loss is a function of the parameters θ of our parametric proposal distribution q_θ:

$$\mathcal{L}(\theta) = \mathbb{E}_{\{\mathcal{R}_t\}\sim q_\theta}\left[r(\{\mathcal{R}_t\}_{t=0..T})\right]. \tag{3.2.4.6}$$

3.2.4.2.2 Parameterization of the Move Proposal Distribution The proposal distribution (3.2.4.3) originally used in Stoke [9] takes the form of a hierarchical model. The type of the move is initially sampled from a probability distribution. Additional samples are drawn to specify, for example, the affected location in the programs ,the new operands or opcode to use. Which of these probability distribution get sampled depends on the type of move that was first sampled. The detailed structure of the proposal probability distribution can be found in Appendix A.4.2.

Stoke uses uniform distributions for each of the elementary probability distributions the model samples from. This corresponds to a specific instantiation of the general stochastic search paradigm. In this work, we propose to learn those probability distributions so as to maximize the probability of reaching the best programs. The rest of the optimization scheme remains similar to the one of [9].

Our chosen parameterization of q is to keep the hierarchical structure of the original work of [9], as detailed in Appendix A.4.2, and parameterize all the elementary probability distributions (over the positions in the programs, the instructions to propose or the arguments) independently. The set θ of parameters for q_θ will thus contain a set of parameters for each elementary probability distributions. A fixed proposal distribution is kept through the optimization of a given program, so the proposal distribution needs to be evaluated only once, at the beginning of the optimization and not at every iteration of MCMC.

The stochastic computation graph corresponding to a run of the Metropolis algorithm is given in Figure 3.3. We have assumed the operation of evaluating the cost of a program to be a deterministic function, as we will not model the randomness of measuring performance.

3.2.4.2.3 Learning the Proposal Distribution In order to learn the proposal distribution, we will use stochastic gradient descent on our loss function (3.2.4.6). We obtain the first order derivatives with regards to our proposal distribution parameters using the REINFORCE [72] estimator, also known as the likelihood ratio estimator [90] or the score function estimator [91]. This estimator relies on a rewriting of the gradient of the expectation. For an expectation with regards to a probability distribution $x \sim f_\theta$, the REINFORCE estimator is:

$$\nabla_\theta \sum_x f(x;\theta) r(x) = \sum_x r(x) \nabla_\theta f(x;\theta) = \sum_x f(x;\theta) r(x) \nabla_\theta \log(f(x;\theta)), \quad (3.2.4.7)$$

and provides an unbiased estimate of the gradient.

A helpful way to derive the gradients is to consider the execution traces of the search procedure under the formalism of stochastic computation graphs [92]. We introduce one "cost node" in the computation graphs at the end of each iteration of the sampler. The associated cost corresponds to the normalized difference between the best rewrite so far and the current rewrite after this step:

$$c_t = \min\left(0, \left(\frac{\text{cost}\left(\mathcal{R}_t, \mathcal{T}\right) - \min_{i=0..t-1} \text{cost}\left(\mathcal{R}_i, \mathcal{T}\right)}{\text{cost}\left(\mathcal{R}_0, \mathcal{T}\right)}\right)\right). \quad (3.2.4.8)$$

The sum of all the cost nodes corresponds to the sum of all the improvements made when a new lowest cost was achieved. It can be shown that up to a constant term, this is equivalent to our objective function (3.2.4.5). As opposed to considering only a final cost node at the end of the T iterations, this has the advantage that moves which were not responsible for the improvements would not get assigned any credit.

For each round of MCMC, the gradient with regards to the proposal distribution is computed using the REINFORCE estimator which is equal to

$$\widehat{\nabla_{\theta,i}} \mathcal{L}(\theta) = (\nabla_\theta \log q_\theta(\mathcal{R}_i | \mathcal{R}_{i-1})) \sum_{t>i} c_t. \quad (3.2.4.9)$$

As our proposal distribution remains fixed for the duration of a program optimization, these gradients needs to be summed over all the iterations to obtain the total contribution to the proposal distribution. Once this gradient is estimated, it becomes possible to run standard back-propagation with regards to the features on which the proposal distribution is based on, so as to learn the appropriate feature representation.

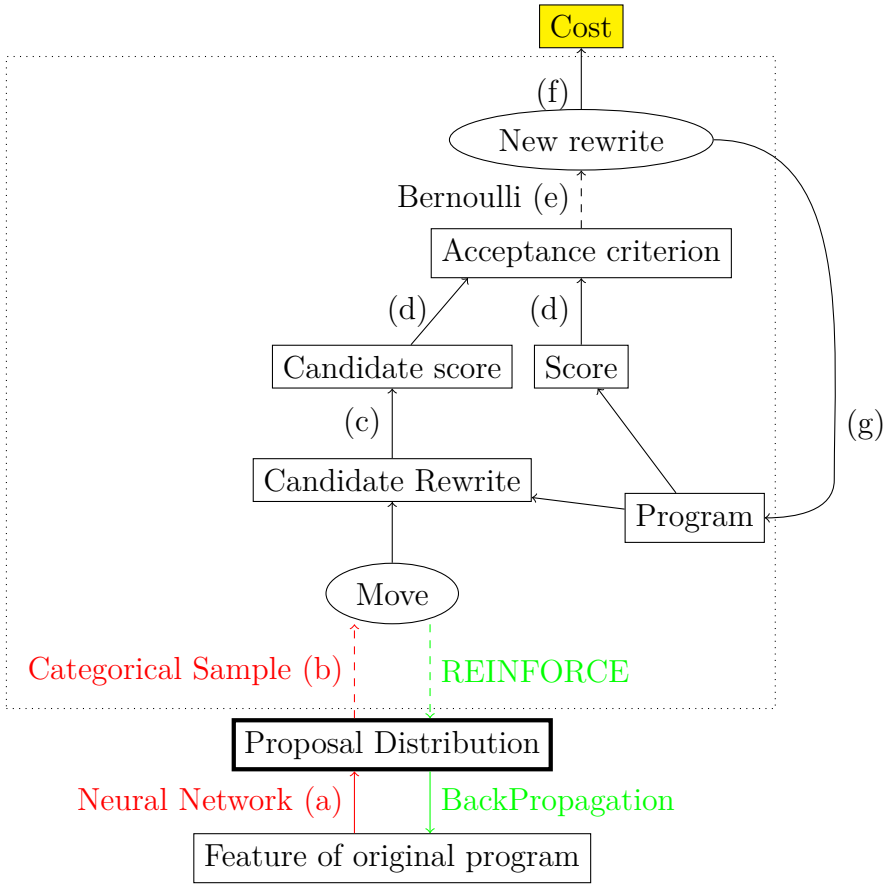


Figure 3.3: *Stochastic computation graph of the Metropolis algorithm used for program super-optimization. Round nodes are stochastic nodes and square ones are deterministic. Red arrows corresponds to computation done in the forward pass that needs to be learned while green arrows correspond to the backward pass. Full arrows represent deterministic computation and dashed arrows represent stochastic ones. The different steps of the forward pass are:*
(a) Based on features of the reference program, the proposal distribution q is computed.
(b) A random move is sampled from the proposal distribution.
(c) The score of the proposed rewrite is experimentally measured.
(d) The acceptance criterion (3.2.4.4) for the move is computed.
(e) The move is accepted with a probability equal to the acceptance criterion.
(f) The cost is observed, corresponding to the best program obtained during the search.
(g) Moves b to f are repeated T times.

3.2.5 Experiments

3.2.5.1 Setup

Implementation Our system is built on top of the Stoke super-optimizer from [9]. We instrumented the implementation of the Metropolis algorithm to allow sampling from parameterized proposal distributions instead of the uniform distributions previously used. Because the proposal distribution is only evaluated

Proposal distribution	MCMC iterations throughput
Uniform	60 000 /second
Categorical	20 000 /second

Table 3.2: Throughput of the proposal distribution estimated by timing MCMC for 10000 iterations

once per program optimization, the impact on the optimization throughput is low, as indicated in Table 3.2.

Our implementation also keeps track of the traces through the stochastic graph. Using the traces generated during the optimization, we can compute the estimator of our gradients, implemented using the Torch framework [93].

Datasets We validate the feasibility of our learning approach on two experiments. The first is based on the Hacker's delight [78] corpus, a collection of twenty five bit-manipulation programs, used as benchmark in program synthesis [9, 94, 95]. Those are short programs, all performing similar types of tasks. Some examples include identifying whether an integer is a power of two from its binary representation, counting the number of bits turned on in a register or computing the maximum of two integers. An exhaustive description of the tasks is given in Appendix A.4.3. Our second corpus of programs is automatically generated and is more diverse.

Models The models we are learning are a set of simple elementary probabilities for the categorical distribution over the instructions and over the type of moves to perform. We learn the parameters of each separate distribution jointly, using a Softmax transformation to enforce that they are proper probability distributions. For the types of move where opcodes are chosen from a specific subset, the probabilities of each instruction are appropriately renormalized. We learn two different type of models and compare them with the baseline of uniform proposal distributions equivalent to Stoke.

Our first model, henceforth denoted the bias, is not conditioned on any property of the programs to optimize. By learning this simple proposal distribution, it is only possible to capture a bias in the dataset. This can be understood as an optimal proposal distribution that Stoke should default to.

The second model is a Multi Layer Perceptron (MLP), conditioned on the input program to optimize. For each input program, we generate a Bag-of-Words representation based on the opcodes of the program. This is embedded through

Model	# of parameters
Uniform	0
Bias	2912
MLP	1.4×10^6

Table 3.3: Size of the different models compared. Uniform corresponds to Stoke [9].

a three hidden layer MLP with ReLU activation unit. The proposal distribution over the instructions and over the type of moves are each the result of passing the outputs of this embedding through a linear transformation, followed by a SoftMax.

The optimization is performed by stochastic gradient descent, using the Adam [76] optimizer. For each estimate of the gradient, we draw 100 samples for our estimator. The values of the hyperparameters used are given in Appendix A.4.1. The number of parameters of each model is given in Table 3.3.

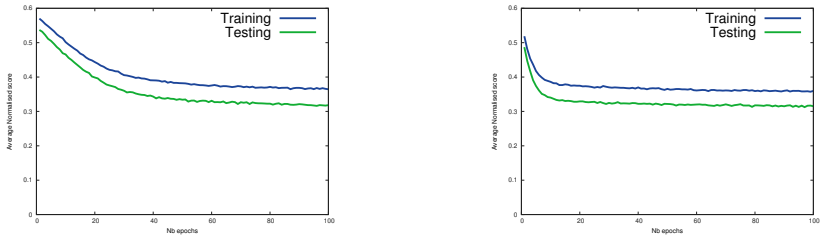
3.2.5.2 Existing Programs

In order to have a larger corpus than the twenty-five programs initially present in "Hacker's Delight", we generate various starting points for each optimization. This is accomplished by running Stoke with a cost function where $\omega_p = 0$ in (3.2.4.1), and keeping only the correct programs. Duplicate programs are filtered out. This allows us to create a larger dataset from which to learn. Examples of these programs at different level of optimization can be found in Appendix A.4.4.

We divide this augmented Hacker's Delight dataset into two sets. All the programs corresponding to even-numbered tasks are assigned to the first set, which we use for training. The programs corresponding to odd-numbered tasks are kept for separate evaluation, so as to evaluate the generalisation of our learnt proposal distribution.

The optimization process is visible in Figure 3.4, which shows a clear decrease of the training loss and testing loss for both models. While simply using stochastic super-optimization allows to discover programs 40% more efficient on average, using a tuned proposal distribution yield even larger improvements, bringing the improvements up to 60%, as can be seen in Table3.4. Due to the similarity between the different tasks, conditioning on the program features does not bring any significant improvements.

In addition, to clearly demonstrate the practical consequences of our learning, we present in Figure 3.5 a superposition of score traces, sampled from the optimization of a program of the test set. Figure 3.5a corresponds to our initialisation, an uniform distribution as was used in the work of [9]. Figure 3.5d corresponds to our



(a) Bias **(b)** Multi-layer Perceptron

Figure 3.4: Proposal distribution training. All models learn to improve the performance of the stochastic optimization. Because the tasks are different between the training and testing dataset, the values between datasets can't directly be compared as some tasks have more opportunity for optimization. It can however be noted that improvements on the training dataset generalise to the unseen tasks.

Model	Training	Test
Uniform	57.01%	53.71%
Bias	36.45 %	31.82 %
MLP	35.96 %	31.51 %

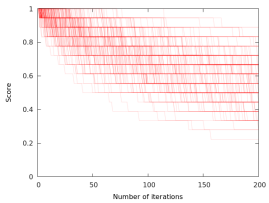
Table 3.4: Final average relative score on the Hacker's Delight benchmark. While all models improve with regards to the initial proposal distribution based on uniform sampling, the model conditioning on program features reach better performances.

optimized version. It can be observed that, while the uniform proposal distribution was successfully decreasing the cost of the program, our learnt proposal distribution manages to achieve lower scores in a more robust manner and in less iterations. Even using only 100 iterations (Figure 3.5e), the learned model outperforms the uniform proposal distribution with 400 iterations (Figure 3.5c).

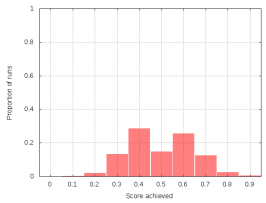
3.2.5.3 Automatically Generated Programs

While the previous experiments shows promising results on a set of programs of interest, the limited diversity of programs might have made the task too simple, as evidenced by the good performance of a blind model. Indeed, despite the data augmentation, only 25 different tasks were present, all variations of the same programs task having the same optimum.

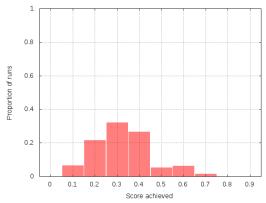
To evaluate our performance on a more challenging problem, we automatically synthesize a larger dataset of programs. Our methods to do so consists in running Stoke repeatedly with a constant cost function, for a large number of iterations. This leads to a fully random walk as every proposed programs will have the same cost, leading to a 50% chance of acceptance. We generate 600 of these



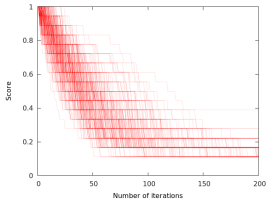
(a) With Uniform proposal Optimization Traces



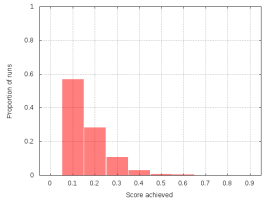
(b) Scores after 200 iterations



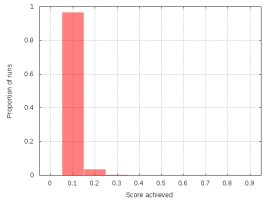
(c) Scores after 400 iterations



(d) With Learned Bias Optimization Traces



(e) Scores after 100 iterations

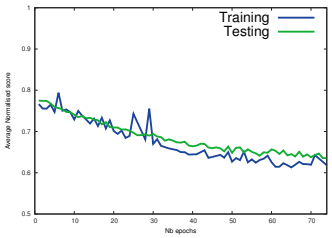


(f) Scores after 200 iterations

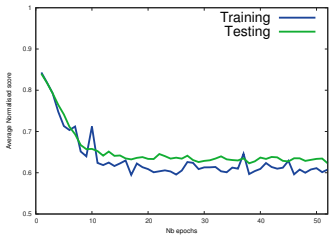
Figure 3.5: Distribution of the improvement achieved when optimising a training sample from the Hacker's Delight dataset. The first column represent the evolution of the score during the optimization. The other columns represent the distribution of scores after a given number of iterations.
(a) to (c) correspond to the uniform proposal distribution, (d) to (f) correspond to the learned bias.

programs, 300 that we use as a training set for the optimizer to learn over and 300 that we keep as a test set.

The performance achieved on this more complex dataset is shown in Figure 3.6 and Table 3.5.



(a) Bias



(b) Multi-layer Perceptron

Figure 3.6: Training of the proposal distribution on the automatically generated benchmark.

Model	Training	Test
Uniform	76.63%	78.15 %
Bias	61.81%	63.56%
MLP	60.13%	62.27%

Table 3.5: Final average relative score. The MLP conditioning on the features of the program perform better than the simple bias. Even the unconditioned bias performs significantly better than the Uniform proposal distribution.

3.2.6 Conclusion

Within this paper, we have formulated the problem of optimizing the performance of a stochastic super-optimizer as a Machine Learning problem. We demonstrated that learning the proposal distribution of a MCMC sampler was feasible and lead to faster and higher quality improvements. Our approach is not limited to stochastic superoptimization and could be applied to other stochastic search problems.

It is interesting to compare our method to the synthesis-style approaches that have been appearing recently in the Deep Learning community [61] that aim at learning algorithms directly using differentiable representations of programs. We find that the stochastic search-based approach yields a significant advantage compared to those types of approaches, as the resulting program can be run independently from the Neural Network that was used to discover them.

Several improvements are possible to the presented methods. In mature domains such as Computer Vision, the representations of objects of interests have been widely studied and as a result are successful at capturing the information of each sample. In the domains of programs, obtaining informative representations remains a challenge. Our proposed approach ignores part of the structure of the program, notably temporal, due to the limited amount of existing data. The synthetic data having no structure, it wouldn't be suitable to learn those representations from it. Gathering a larger dataset of frequently used programs so as to measure more accurately the practical performance of those methods seems the evident next step for the task of program synthesis.

4

Solving Linear Continuous Relaxations

Contents

4.1 Proximal LP Solver for Block Bounded Problems

4.1.1 Preamble

The remaining of this section contains a paper under submission to "Optimization and Engineering". Rudy Bunel helped with the experiments for neural network verification in Section 4.1.7.2. This work was done under the supervision of Philip Torr and Pawan Kumar.

We consider the optimization of *block constrained Linear Programs (LP)*: LPs whose constraints can be grouped into a small number of types of blocks, where two blocks of constraints are of the same type if they share a common set of

coefficients that are applied to different sets of variables. We assume that it is computationally efficient to optimize a linear objective over a single block of constraints. Block bounded LPs are frequently encountered in practice. In this paper, we consider two examples of such LPs. The first considers the roof duality relaxation of the quadratic pseudo-boolean optimization problem which is a classic NP-hard problem and a natural formulation for many combinatorial optimization problems. The second considers the neural network verification problem which aims at proving properties on the input-output mapping associated with a trained neural network. To exploit the structure of the problems, we base our algorithm on Lagrangean Decomposition (LD). We improve the current state-of-the-art solver for LD, which is based on projected supergradient ascent, using the framework of proximal minimization. We show that each proximal problem can be solved efficiently using the conditional gradient algorithm via the analytical computation of the optimal step-size. In contrast to the projected supergradient ascent, our approach relies on a single hyperparameter, namely the weight of the proximal term. We show the efficacy of the proposed solver using the two aforementioned examples of block bounded LPs. We experimentally demonstrate that our solver is able to leverage the structure of these problems and new hardware, such as GPUs, to provide significant speed-up compared to generic solvers.

4.1.2 Introduction

Linear Programming (LP) plays a central role in many fields such as computational biology, operation research, engineering and machine learning. For example, LP solvers can be used to solve relaxations of Integer Programs such as the Quadratic Pseudo-Boolean Optimization (QPBO) problem or prove properties of neural networks (as detailed in Section 4.1.4). In this paper, we focus on a specific class of LP, which we term block bounded LP, that occurs frequently in practice and propose a new solver that is able to leverage their specificities. Block bounded LPs have the following four properties. First, the LP constraints can be grouped into blocks. For these blocks to be interesting, the problem arising from optimizing a linear objective on a single block ignoring the others should be easy to solve. Second, there must be a small number of types of blocks. We consider that two blocks are of the same type if they have the same constraint coefficients applied to different subsets of the variables. Third, exact solutions are not needed and approximate but accurate solutions are sufficient if they are a lower bound of the exact minimal value. Fourth we are interested in solving not just one LP, but a series of related ones. This

paper presents an LP solver that is able to leverage these properties and provide a fast and simple solver without any other assumption on the problem being solved.

The natural approach to solve block problems is Lagrangean Decomposition (LD) [96]. LD introduces the notion of subproblems that are similar to the original LP while ignoring some constraints. It also adds new constraints to enforce that the solutions of all the subproblems are consistent. If every constraint is considered in at least one subproblem, such a formulation will solve the original LP. Any LD algorithm will require us to solve these subproblems independently of each other. We can use the first property and create a subproblem for each block of constraints to ensure that the subproblems are easy to solve. Moreover, using the second property, we can ensure that solving these problems in parallel will be efficient on modern Single Instruction Multiple Data (SIMD) computer architecture because a single algorithm can be used to solve all the subproblems of the same type. The main drawback with using LD is that the overall algorithm used to solved the new formulation is, historically, projected supergradient ascent. While each update is cheap, since the supergradient is easy to compute, in practice, this algorithm is sensitive to the choice of the step-size. In this work we will use the framework of proximal minimization to overcome this problem. This framework is widely used and discussed in detail in [46]. Indeed, by making the original problem strongly convex, its dual formulation becomes smooth and consequently, amenable to hyperparameter-free optimization algorithms such as the Frank Wolfe (FW) aka Conditional Gradient algorithm [97] for which we can compute the step-size in closed form. We will show empirically that the weight associated with the proximal term is easy to tune and allows for a more efficient optimizer. Moreover, using the recent work of [48], we know that the conditional gradient required by the FW algorithm is equivalent to the supergradient used by the projected supergradient ascent algorithm. Thus the algorithm proposed in this paper will have the same runtime and memory complexity per iteration as the existing projected supergradient algorithm while providing faster convergence. Our algorithm performs iterative steps in the dual of the original LP and so maintains a lower bound on the minimal value at every step. Given a required precision by the user, the solver can leverage the third property to return the required solution without the need to solve the LP exactly. Moreover, the proximal term of the algorithm allows to efficiently use good initializations and so speed up the main task using the fourth property. As we will see in the experimental section, such an approach allows us to make full use of recent SIMD hardware such as GPUs.

4.1.3 Related Work

Research on solving general LPs has been active for several decades. The two main methods used in practice are the Simplex algorithm from Dantzig [6] and the interior point methods popularized in the 80s and described, for example, in [7]. State of the art solvers use combinations of these methods with advanced preprocessing. The fastest solvers are commercial ones such as Gurobi [98] and CPLEX [99]. High quality open source alternatives also exist such as GLPK [100], Glop [101] or COIN-CLP [102] to cite only a few. The quality of these solvers is largely influenced by the efficiency of the heuristics used to detect and exploit the problem structure. Using such a solver has many drawbacks for the problems we consider in this paper. The main one is that, even though we have a known structure in our problems, it is not possible to inject that knowledge into the solver. Thus, it may not be able to exploit the specific properties of our problem. For example, it won't be able to easily use the block structure, the fact that multiple similar problems are solved at the same time or that the constraints can be implicit if the right combinatorial algorithm is used to solve the subproblems.

Another line of work closer to this paper is [103] and the practical implementation at [104]. It uses alternating direction method of multipliers (ADMM) to solve any convex problem that can be expressed as a set of subproblems. This solver, though, is not specialized for linear programs and is not able to exploit the simplicity of the objective function, which allows us to solve subproblems in closed form.

On the opposite side, the development of LP solvers highly customised for specific applications is an important avenue of research. It enables to solve large scale problem very effectively by exploiting special structures. For example, problems over Matroids can be solved exactly using a greedy algorithm in a known number of steps [105]. Similarly, the linear relaxation of the maximum a posteriori (MAP) estimation for Markov Random Fields (MRF) has been studied in detail by the machine learning community. The work by [106] presents a maxflow-based algorithm for MAP estimation. The solver is able to reach the exact optimal solution in a known number of steps and is able to provide stronger guarantees than a general LP solver. Even though these algorithms are very efficient, their application is limited in practice as they can only solve one specific problem.

LD as discussed above was introduced in [96] as a high quality relaxation for problems that naturally separate into blocks. This dual formulation provides an interesting view of the original problem as a set of subproblems. The solution of this relaxation has been discussed in [107], which recommends projected supergradient ascent for its simplicity and general applicability. Note that this is an ascent algorithm

as the dual of a minimization problem defines a maximization problem. This idea has been followed in several fields where specialized algorithms have been developed for graph structured combinatorial optimization problems applied to minimum weighted arborescence [108], infrastructure planning under complex economic objectives [109] and MRF optimization [52]. Unfortunately these algorithms have two important drawbacks. First, the implementations are based on specialized derivations for each problem and do not provide a framework to solve LD problems in general. Second, all the implementations use projected supergradient for the optimization, which make them sensitive to step-sizes tuning. In particular, each paper introduces a specific scheduling of the step-size at every iteration of the algorithm.

On the one hand, our solver is less general than general purpose LP solvers as it will only be efficient to solve block bounded LPs. On the other hand, it is more general than solvers specialized for a single task as it can, in theory, be used to solve any LP. In practice, it can be seen as an extension of existing LD-based solvers, generalised for any block bounded LP and using a more efficient optimization algorithm.

4.1.4 Problem Formulation

In this section we present the general formulation for LPs and the specific types of problem we are going to consider. We also introduce all the necessary notations that will then be used in the rest of the paper.

We use bold lower case letters for vectors, bold upper case letters for matrices and bold function names for functions with values in a vector space. In particular, $\mathbf{0}$ represent a vector of zeroes of the appropriate size. Capital letters are used to represent sets of indices or numerical constants and calligraphic capital letters represent sets of sets of indices. Superscripts are used to select an element or a subset of a vector. For example, $\mathbf{x}^j$ is the j-th entry in the vector $\mathbf{x}$ and $\mathbf{x}^C$ is a vector containing $\mathbf{x}^j, \forall j \in C$. Subscripts identify different objects and sets of integers are defined as $[K] = \{1, 2, \ldots, K\}$. For example, $\mathbf{g}_k(\mathbf{x}), \forall k \in [K]$ represent a set of K different functions.

4.1.4.1 General LP

Using these notations, the general LP formulation can be written as:

$$\begin{aligned} &\min_{\mathbf{x}} \ \mathbf{c}^\top \mathbf{x} \\ &\text{s.t. } \mathbf{g}(\mathbf{x}) \geq \mathbf{0}. \end{aligned} \tag{4.1.4.1}$$

Here, $\mathbf{x}$ are the optimization variables, $\mathbf{c}$ the real valued vector that specifies the linear objective and $\mathbf{g}(\mathbf{x})$ is a linear mapping of $\mathbf{x}$. Indeed we will minimize a linear objective subject to a set of linear constraints. To simplify further discussions, we assume that the feasible regions for all the problems we consider are bounded and non-empty. Note that the constraint set is necessarily closed because of the non-strict inequality constraints. To make this formulation clearer, we will use two example LPs throughout the article.

Example: QPBO The first LP we consider is the QPBO problem and its roof duality relaxation. The original QPBO problem can be written as:

$$\begin{aligned} &\min_{\mathbf{y}} \mathbf{y}^\top \mathbf{a} + \mathbf{y}^\top \mathbf{P} \mathbf{y} \\ &\text{s.t. } \mathbf{y} \in \{0,1\}^N. \end{aligned} \tag{4.1.4.2}$$

Here, N is the number of variables in the problem, $\mathbf{a}$ and $\mathbf{P}$ are real valued and encode the problem specifications and $\mathbf{y}$ are the binary variables we want to optimize over. Problems in this formulation are usually encoded as follows. A given variable is considered to be selected if $\mathbf{y}^j = 1$ and $\mathbf{a}^j$ represents the cost of selecting this particular variable, independently of the others. The value $\mathbf{P}^{ij}$ represents the cost of selecting both the i-th and j-th variables at the same time. It can be used, for example, to prevent two variables to be selected at the same time. This problem is hard and so is usually solved using branch and bound (BB). One key element of BB is the bounding method that should return a useful lower bound to the solution of the original problem. For the QPBO problem, a linear relaxation can be obtained in three steps. First relax the binary constraint on $\mathbf{y}$ to the convex hull of the original set: $\mathbf{y} \in [0,1]^N$. Second, introduce new variables $\mathbf{z}$ that will correspond to the product of the original variables as $\mathbf{z}^{ij} = \mathbf{y}^i \mathbf{y}^j$. Finally, since this equality constraint is not linear, we relax it using four new linear constraints to get to the following problem:

$$\begin{aligned} &\min_{\mathbf{y},\mathbf{z}} \mathbf{y}^\top \mathbf{a} + \sum_{i,j} \mathbf{z}^{ij} \mathbf{P}^{ij} \\ &\text{s.t. } \mathbf{y} \in [0,1]^N \\ &\qquad 1 \le i,j \le N \begin{cases} \mathbf{z}^{ij} \ge \mathbf{y}^i + \mathbf{y}^j - 1 \\ \mathbf{z}^{ij} \ge 0 \\ \mathbf{z}^{ij} \le \mathbf{y}^i \\ \mathbf{z}^{ij} \le \mathbf{y}^j. \end{cases} \end{aligned} \tag{4.1.4.3}$$

This relaxation is discussed in more detail in [110]. In particular, other formulations leading to the same relaxation are discussed and persistency properties for this relaxation are derived.

Example: Neural Network Verification The second problem we are interested in is the neural network verification problem. In this paper, we restrict ourselves to neural networks that only contain convolutional or linear layers and Rectified Linear Units (ReLU) as non linearities. The ReLU non linearity is defined as $\text{ReLU}(x) = 0$ if $x < 0$ and $\text{ReLU}(x) = x$ otherwise. These networks can be seen as the composition of several functions, alternating between linear transformations and non linearities. This is an efficient way to obtain complex non linear mappings using simple components and allow easy first-order optimization. Considering a trained network, we want to prove some property of the output given a set of linear constraints on the input. This can be used, for example, to prove that a neural network is robust to adversarial attacks. An adversarial attack aims at finding an input which is both close to a correctly classified reference image and incorrectly classified by the neural network. Mathematically, an input is considered close to a reference image if it lies within a given polytope around the reference image. By optionally adding a new linear layer at the end of the network, this problem can be reduced to proving that the minimum value of the output of the network is non negative subject to the constraint that the input is in the specified polytope and that the variables match the neural network specifications as shown in [111]. Unfortunately, this problem is not convex as the mappings corresponding to the ReLU layers lead to non-convex constraints. To be able to solve this problem, the ReLU constraints are relaxed to the convex hull of the possible values it can take based on estimated minimum and maximum values for the ReLU's input.

We now describe in more detail the problem of finding the minimum value of all the neurons of the L-th layer of the neural network. We denote by $\mathbf{y}_{l-1}$ the vector containing the values of the neurons at the input of the l-th linear layer. The variable $\widetilde{\mathbf{y}}_l$ will contain the value obtained by applying the linear transformation of the l-th layer to $\mathbf{y}_{l-1}$. Note that the input of the network is $\mathbf{y}_0$ and the output is $\widetilde{\mathbf{y}}_L$. We use $\mathbf{W}_l$ and $\mathbf{b}_l$ to represent the weight and bias of the l-th linear layer. For a convolutional layer, $\mathbf{W}_l$ is the linear matrix equivalent to the convolution operation. In this case, it contains the original weights of the convolution repeated for every output point and scattered into a sparse matrix. With these notations, the LP we want to solve is the following:

$$\begin{aligned} &\min_{\mathbf{y}_l, \widetilde{\mathbf{y}}_l} \widetilde{\mathbf{y}}_L \\ &\text{s.t. } \forall l \in [L] \begin{cases} \mathbf{l}_{l-1} \leq \widetilde{\mathbf{y}}_{l-1} \leq \mathbf{u}_{l-1} \\ \widetilde{\mathbf{y}}_l = \mathbf{W}_l \mathbf{y}_{l-1} + \mathbf{b}_l \end{cases} \\ &\qquad \forall l \in [L-1] \begin{cases} \mathbf{y}_l \geq \mathbf{0} \\ \mathbf{y}_l \geq \widetilde{\mathbf{y}}_l \\ \mathbf{y}_l \leq \dfrac{\mathbf{u}_l}{\mathbf{u}_l - \mathbf{l}_l} \odot (\widetilde{\mathbf{y}}_l - \mathbf{l}_l). \end{cases} \end{aligned} \tag{4.1.4.4}$$

Here, $\odot$ represent the element-wise multiplication of vectors to avoid confusion with matrix multiplication. The regular division symbol represents the element-wise division here as no matrix division is used. The first constraint here ensures that the input is within the allowed region. The second constraint ensures the mapping from input to output follows the weights of the linear layers. The remaining constraints correspond to the convex relaxation of each ReLU. Note that the ReLU relaxation described here is only valid if the input range contains 0. When 0 is not within the input range, the ReLU becomes a linear transformation and thus no relaxation is necessary. In the following, we assume that the input range for each ReLU contains 0 to simplify the equations.

For each layer before the L-th layer, this relaxation requires us to know an upper and lower bound for each variable. The quality of this relaxation will be very dependent of the quality of these extrema. Good values for these extrema of a layer L' can be obtained by solving new LPs. The strategy introduced in [112] is to compute these extrema for each layer one after the other. They are computed by solving the same problem as above for the L'-th layer. The two extrema are obtained by minimizing and maximizing the objective. All the LPs described here represent linear objectives over a set of linear constraints and so fit into the framework described in Eq. (4.1.4.1).

4.1.4.2 Block Bounded LP

We now restrict the formulation from Eq. (4.1.4.1) to take into account the block structure of the problems we consider in this paper. To this end, we first consider that there are K types of blocks of constraints in our problem that we index with $k \in [K]$. Each type operates on B_k blocks of variables that we index with $b \in [B_k]$. To simplify the notation going forward, we will use $\forall k, b$ to specify $\forall k \in [K], \forall b \in [B_t]$. We define $C_{k,b}$ as the indices of the variables on which the b-th block of type k works. Moreover $\mathbf{g}_k$ represent the common linear mapping used by all blocks of type k. In other words, $\mathbf{g}_k$ is the set of common linear coefficients applied to each set of variables indexed by $C_{k,b}$. Using these, Eq. (4.1.4.1) can be rewritten as:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{c}^\top \mathbf{x} \\ &\text{s.t. } \forall k, b,\ \mathbf{g}_k(\mathbf{x}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.4.5}$$

Example: QPBO Coming back to the example of the QPBO roof duality problem, we define a single type of blocks of constraints. We define the blocks as being associated with a pair of original variables, for example $\mathbf{y}^i$ and $\mathbf{y}^j$, and the product variable associated with them, $\mathbf{z}^{ij}$. These blocks only need to be defined if the corresponding entry in $\mathbf{P}$ is non-zero and so we define $\{C_{1,b}, \forall b\} \equiv \{(i,j,ij), \forall (i,j) \,|\, \mathbf{P}^{ij} \neq 0\}$. We also define the linear mapping $\mathbf{g}_1 : \mathbb{R}^3 \longrightarrow \mathbb{R}^8$ as the function that produces the following vector output:

$$\mathbf{g}_1(u,v,w) = \begin{pmatrix} u \\ 1-u \\ v \\ 1-v \\ w \\ 1+w-u-v \\ u-w \\ v-w. \end{pmatrix}. \tag{4.1.4.6}$$

We can then rewrite Eq. (4.1.4.3) as:

$$\begin{aligned} &\min_{\mathbf{y},\mathbf{z}} \mathbf{y}^\top \mathbf{a} + \sum_{i,j} \mathbf{z}^{ij} \mathbf{P}^{ij} \\ &\text{s.t. } \forall b, \mathbf{g}_1((\mathbf{y},\mathbf{z})^{C_{1,b}}) \geq \mathbf{0}, \end{aligned} \tag{4.1.4.7}$$

with the notation that $(\mathbf{y},\mathbf{z})^{C_{1,b}} = (\mathbf{y}^i, \mathbf{y}^j, \mathbf{z}^{ij})$ for $C_{1,b} = (i,j,ij)$.

Example: Neural Network Verification For the neural network verification problem, splitting the constraints into blocks is more complex as the problems from different layers lead to different constraints. In this example, we define one type of block of variable for each layer and each type will contain a single block. A block is composed of all the constraints of a given linear layer and the constraints associated with the ReLU before it. We define the linear mappings $\mathbf{g}_l$ for each layer $l \geq 1$ as the following vector functions:

$$\mathbf{g}_l(\widetilde{\mathbf{v}}_l, \mathbf{v}, \widetilde{\mathbf{v}}_{l+1}) = \begin{pmatrix} \widetilde{\mathbf{v}}_l - \mathbf{l}_l \\ \mathbf{u}_l - \widetilde{\mathbf{v}}_l \\ \mathbf{v} \\ \mathbf{v} - \widetilde{\mathbf{v}}_l \\ \dfrac{\mathbf{u}}{\mathbf{u}-\mathbf{l}}(\widetilde{\mathbf{v}}_l - \mathbf{l}) - \mathbf{v}. \\ \widetilde{\mathbf{v}}_{l+1} - \mathbf{W}_{l+1}\mathbf{v} - \mathbf{b}_{l+1} \\ \mathbf{W}_{l+1}\mathbf{v} + \mathbf{b}_{l+1} - \widetilde{\mathbf{v}}_{l+1} \end{pmatrix}. \tag{4.1.4.8}$$

Here, the first two dimensions are associated with the constraints that ensure that the input is within the previously computed extrema. The following two ensure

that the linear transformation is verified and the final three correspond to the relaxation of the ReLU. Since the first layer does not have a ReLU before the linear layer, we define the following linear mapping:

$$\mathbf{g}_0(\widetilde{\mathbf{v}}_0, \mathbf{v}, \widetilde{\mathbf{v}}_1) = \begin{pmatrix} \mathbf{v} - \mathbf{l}_0 \\ \mathbf{u}_0 - \mathbf{v} \\ \widetilde{\mathbf{v}}_1 - \mathbf{W}_1\mathbf{v} - \mathbf{b}_1 \\ \mathbf{W}_1\mathbf{v} + \mathbf{b}_1 - \widetilde{\mathbf{v}}_1 \\ \widetilde{\mathbf{v}}_0 - \mathbf{v} \\ \mathbf{v} - \widetilde{\mathbf{v}}_0. \end{pmatrix}. \tag{4.1.4.9}$$

To simplify notations, we add $\widetilde{\mathbf{y}}_0 = \mathbf{y}_0$ for the first layer that is enforced by the last two constraints defined by $\mathbf{g}_0$. We can then rewrite Eq. (4.1.4.4) as:

$$\begin{aligned} &\min_{\mathbf{y}_l, \widetilde{\mathbf{y}}_l} \widetilde{\mathbf{y}}_L \\ &\text{s.t. } \forall l \in [L]\ \mathbf{g}_{l-1}(\widetilde{\mathbf{y}}_{l-1}, \widetilde{\mathbf{y}}_l) \geq \mathbf{0}. \end{aligned} \tag{4.1.4.10}$$

The reader will notice that contrary to the QPBO problem, the strategy here is to consider the smallest number of blocks, even if they are of different types. Indeed, neural networks rarely have more than a hundred layers and the ones considered in the verification field are even smaller. This reduced number of blocks makes the solution of each block very informative with respect to the overall problem and speeds up the convergence of LD.

4.1.4.3 Min Oracle

The above block-bounded formulation of the problem as a set of subproblems that share variables is especially interesting as we will be able to use approaches from LD to solve it. The details of LD are provided in the next section. For now we describe its essential component, namely, the ability to solve each subproblem efficiently. In particular, we will call an algorithm that, given a vector $\mathbf{c}_{k,b}$, will solve the subproblem indexed by (k, b) as the min oracle. Formally, it means solving the following problem for a given k and b:

$$\begin{aligned} &\min_{\mathbf{x}^{C_{k,b}}} {\mathbf{c}_{k,b}}^\top \mathbf{x}^{C_{k,b}} \\ &\text{s.t. } \mathbf{g}_k(\mathbf{x}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.4.11}$$

To be able to design an efficient algorithm based on LD, we need to be able to solve each of these problems efficiently. As b here only affects which subset of $\mathbf{x}$ is considered but does not change the problem, we only require a min oracle for each type k and all the problems of this type can be solved using the same algorithm. This property of running a single algorithm on different data is key to easily obtain a parallel algorithm that efficiently uses GPUs.

Example: QPBO For the QPBO roof duality problem, a given subproblem can be solved in two steps. First, based on the sign of the coefficients that multiply $\mathbf{y}^i$ and $\mathbf{y}^j$ in the objective function, set their value to 0 if the coefficient is positive and 1 if the coefficient is negative. Second, based on the sign of the coefficient that multiplies $\mathbf{z}^{ij}$, set its value to $\min(\mathbf{y}^i, \mathbf{y}^j)$ if the coefficient is positive and $\max(0, \mathbf{y}^i + \mathbf{y}^j - 1)$ otherwise. Note that if some coefficients are zero in the first step, then the value for the corresponding entry in $\mathbf{y}$ is set to 0 if the coefficient of $\mathbf{z}^{ij}$ is positive and 1 otherwise. This algorithm allows us to solve any subproblem by only checking the sign the 3 linear coefficients. It is interesting to note that this combinatorial algorithm gives a solution that verifies all the constraints described in Eq. (4.1.4.7) without having to evaluate or enumerate them.

Example: Neural Network Verification For the neural network verification problem, we define two different algorithms. One is used to solve the subproblem associated with the first layer, the other is used to solve the subproblems associated with the other layers.

For the first layer, we want to solve the following problem for a given $\mathbf{c}$ and $\widetilde{\mathbf{c}}$ after removing the additional $\widetilde{\mathbf{y}}_0$ that was added:

$$\begin{aligned} \min_{\mathbf{y}_0, \widetilde{\mathbf{y}}_1} \; & \mathbf{c}^\top \mathbf{y}_0 + \widetilde{\mathbf{c}}^\top \widetilde{\mathbf{y}}_1 \\ \text{s.t. } & \mathbf{l}_0 \leq \mathbf{y}_0 \leq \mathbf{u}_0 \\ & \widetilde{\mathbf{y}}_1 = \mathbf{W}_1 \mathbf{y}_0 - \mathbf{b}_1. \end{aligned} \tag{4.1.4.12}$$

We can solve this problem by eliminating the $\widetilde{\mathbf{y}}_1$ variable using the second constraint to obtain the following problem after rearranging the terms:

$$\begin{aligned} \min_{\mathbf{y}_0} \; & (\mathbf{c}^\top + \widetilde{\mathbf{c}}^\top \mathbf{W}_1) \mathbf{y}_0 - \widetilde{\mathbf{c}}^\top \mathbf{b}_1 \\ \text{s.t. } & \mathbf{l}_0 \leq \mathbf{y}_0 \leq \mathbf{u}_0. \end{aligned} \tag{4.1.4.13}$$

This problem can be solved element-wise by setting $\mathbf{y}_0$ to $\mathbf{l}_0$ if $\mathbf{c}^\top + \widetilde{\mathbf{c}}^\top \mathbf{W}_1$ is positive and to $\mathbf{u}_0$ if it is negative. This algorithm is particularly interesting for convolutional layers as $\widetilde{\mathbf{c}}^\top \mathbf{W}_1$ can be computed without computing the complete $\mathbf{W}_1$ matrix using a transposed convolution.

For the other layers, the problem is more complex because of the relaxation of the ReLU constraint. The problem that we want to solve for the l-th layer

can be written as:

$$\begin{aligned}\min_{\widetilde{\mathbf{y}}_l, \mathbf{y}_l, \widetilde{\mathbf{y}}_{l+1}} \quad & {\mathbf{c}_l}^\top \widetilde{\mathbf{y}}_l + {\mathbf{c}_{l+1}}^\top \widetilde{\mathbf{y}}_{l+1} \\ \text{s.t. } & \mathbf{l}_l \leq \widetilde{\mathbf{y}}_l \leq \mathbf{u}_l \\ & \widetilde{\mathbf{y}}_{l+1} = \mathbf{W}_{l+1}\mathbf{y}_l - \mathbf{b}_{l+1} \\ & \mathbf{y}_l \geq \mathbf{0} \\ & \mathbf{y}_l \geq \widetilde{\mathbf{y}}_l \\ & \mathbf{y}_l \leq \frac{\mathbf{u}_l}{\mathbf{u}_l - \mathbf{l}_l} \odot (\widetilde{\mathbf{y}}_l - \mathbf{l}_l).\end{aligned} \tag{4.1.4.14}$$

Here, the variables $\mathbf{y}_l$ are in a single subproblem and so do not have a linear coefficient as we will explain in the next section. To solve this problem, we first eliminate the variable $\widetilde{\mathbf{y}}_{l+1}$ using the second constraint to obtain the following problem.

$$\begin{aligned}\min_{\widetilde{\mathbf{y}}_l, \mathbf{y}_l} \quad & {\mathbf{c}_l}^\top \widetilde{\mathbf{y}}_l + ({\mathbf{c}_{l+1}}^\top \mathbf{W}_{l+1})\mathbf{y}_l - {\mathbf{c}_{l+1}}^\top \mathbf{b}_{l+1} \\ \text{s.t. } & \mathbf{l}_l \leq \widetilde{\mathbf{y}}_l \leq \mathbf{u}_l \\ & \mathbf{y}_l \geq \mathbf{0} \\ & \mathbf{y}_l \geq \widetilde{\mathbf{y}}_l \\ & \mathbf{y}_l \leq \frac{\mathbf{u}_l}{\mathbf{u}_l - \mathbf{l}_l} \odot (\widetilde{\mathbf{y}}_l - \mathbf{l}_l).\end{aligned} \tag{4.1.4.15}$$

We can solve this problem by making the following two observations. First, for a given index j, the problem associated with $\widetilde{\mathbf{y}}_l^j$ and $\mathbf{y}_l^j$ is independent of the other variables. So the problem can be solved as a set of problems of size 2. Second, the constraint set for the pair of variables of index j forms a triangle with vertices $(\mathbf{l}_l^j, 0)$, $(0, 0)$ and $(\mathbf{u}_l^j, \mathbf{u}_l^j)$. Since an optimal solution of an LP always lies at a vertex of the constraint set, the solutions for the small LPs can be obtained by evaluating the objective function at each vertex of the triangle and selecting the minimal value as optimal. Thus, we have an efficient algorithm to solve each type of subproblems that we encounter in the neural network verification problem. Note that the efficient algorithm above is based on the linearity of the objective function.

4.1.5 Problem Reformulation via Decomposition

We now turn to the task of solving the problem defined by Eq. (4.1.4.5) given that we have access to an efficient min oracle. We will first present the general LD approach and then derive a proximal formulation to solve it efficiently.

4.1.5.1 Lagrangean Decomposition

LD was introduced in [96]. The main idea is that when considering a problem composed of a set of subproblems, the original problem can be optimized by solving the subproblems multiple times with modified linear objectives. It corresponds to solving small modifications of the subproblems while trying to force them to agree on what the solution for the shared variables should be. We now formally derive this partial Lagrangean.

First, we introduce a set of variables $\mathbf{x}_{k,b}$ that will each contain one copy of the original variables $\mathbf{x}$ for each subproblem. Using these variables, we can write the following problem, which is equivalent to Eq. (4.1.4.5):

$$\begin{aligned} &\min_{\mathbf{x},\mathbf{x}_{k,b}} \mathbf{c}^\top \mathbf{x} \\ &\text{s.t. } \forall k, b, \begin{cases} \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0} \\ \mathbf{x}_{k,b} = \mathbf{x}. \end{cases} \end{aligned} \tag{4.1.5.1}$$

To simplify the notation when working with multiple subproblems, $\mathbf{x}_{k,b}$ has the same size as $\mathbf{x}$. In practice, all the entries that do not appear in the objective function are ignored.

We introduce Lagrange multipliers $\boldsymbol{\lambda}_{k,b}$ associated with the constraint $\mathbf{x}_{k,b} = \mathbf{x}$. We can then write the partial Lagrangean of Eq. (4.1.5.1) as:

$$\begin{aligned} &\max_{\boldsymbol{\lambda}_{k,b}} \min_{\mathbf{x},\mathbf{x}_{k,b}} \mathbf{c}^\top \mathbf{x} + \sum_{k,b} {\boldsymbol{\lambda}_{k,b}}^\top (\mathbf{x}_{k,b} - \mathbf{x}) \\ &\text{s.t. } \forall k, b, \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.5.2}$$

By setting the gradients with respect to $\mathbf{x}$ to zero, we can eliminate this variable and obtain what we will call our dual problem:

$$\begin{aligned} &\max_{\boldsymbol{\lambda}_{k,b}} \min_{\mathbf{x}_{k,b}} \sum_{k,b} {\boldsymbol{\lambda}_{k,b}}^\top \mathbf{x}_{k,b} \\ &\text{s.t. } \forall k, b, \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0} \\ &\qquad \sum_{k,b} \boldsymbol{\lambda}_{k,b} = \mathbf{c}. \end{aligned} \tag{4.1.5.3}$$

We can make two interesting observations about this problem. First, the inner minimization problem corresponds exactly to the min oracle that we defined in Eq. (4.1.4.11) where the linear objective is given here by the dual variables $\boldsymbol{\lambda}_{k,b}$. Second, when solving the outer problem, a supergradient at a given point is $\mathbf{x}^*_{k,b}$, the optimal solution of the inner minimization problem. In other words, a supergradient can be computed easily as it is the solution given by the min oracle. Using these observations, [107] recommends to solve this problem with projected supergradient ascent on $\boldsymbol{\lambda}_{k,b}$ using the min oracle to compute the supergradient.

Example: QPBO For the QPBO roof duality problem, we will use $\boldsymbol{\lambda}_{1,b}$ and $\boldsymbol{\mu}_{1,b}$ to define the dual variables. Using these, we can rewrite Problem (4.1.4.7) as follows:

$$\begin{aligned} \max_{\boldsymbol{\lambda}_{1,b},\boldsymbol{\mu}_{1,b}} \min_{\mathbf{y}_{1,b},\mathbf{z}_{1,b}} & \sum_b (\boldsymbol{\lambda}_{1,b},\boldsymbol{\mu}_{1,b})^{C_b\top} (\mathbf{y}_{1,b},\mathbf{z}_{1,b})^{C_b} \\ \text{s.t. } \forall b, & \mathbf{g}_1((\mathbf{y}_{1,b},\mathbf{z}_{1,b})^{C_b}) \geq \mathbf{0} \\ & \sum_b \boldsymbol{\lambda}_{1,b} = \mathbf{a} \\ & \sum_b \boldsymbol{\mu}_{1,b} = \mathbf{P}. \end{aligned} \tag{4.1.5.4}$$

We can see that this new problem is very similar to the original one where the dual variables replace the original linear coefficients $\mathbf{a}$ and $\mathbf{P}$. The equivalence with the original problem is ensured by the constraint that the dual variables sum to the original linear coefficients.

4.1.5.2 Proximal Problem

To speed up the algorithm from [107], we will now present the proximal scheme that we use. The main advantage of this scheme is that it makes the problem strongly convex and so will make the dual of this problem smooth. We will show that we can leverage the smoothness of the dual to obtain a projection-and hyperparameter-free optimization algorithm.

To do so, we add a proximal term to Eq. (4.1.5.3) to get the following problem:

$$\begin{aligned} \max_{\boldsymbol{\lambda}_{k,b}} \min_{\mathbf{x}_{k,b}} & \sum_{k,b} \boldsymbol{\lambda}_{k,b}{}^{\top} \mathbf{x}_{k,b} - \sum_{k,b} \frac{\eta}{2} ||\boldsymbol{\lambda}_{k,b} - \widetilde{\boldsymbol{\lambda}}_{k,b}||^2 \\ \text{s.t. } \forall k, b, & \mathbf{g}_k(\mathbf{x}_{k,b}{}^{C_{k,b}}) \geq \mathbf{0} \\ & \sum_{k,b} \boldsymbol{\lambda}_{k,b} = \mathbf{c}. \end{aligned} \tag{4.1.5.5}$$

Here η is a fixed scalar weight and $\widetilde{\boldsymbol{\lambda}}_{k,b}$ is a fixed value of the variable $\boldsymbol{\lambda}_{k,b}$ that is updated periodically. This means that $\widetilde{\boldsymbol{\lambda}}_{k,b}$ verifies the same constraint as $\boldsymbol{\lambda}_{k,b}$, namely $\sum_{k,b} \widetilde{\boldsymbol{\lambda}}_{k,b} = \mathbf{c}$.

Working with a proximal problem will change the optimization algorithm. Indeed, we will solve Eq. (4.1.5.5) for a fixed $\widetilde{\boldsymbol{\lambda}}_{k,b}$ until convergence, or until enough improvement has been achieved. Then $\widetilde{\boldsymbol{\lambda}}_{k,b}$ is updated with the final value of $\boldsymbol{\lambda}_{k,b}$ and the problem is solved again. We will discuss in more detail in the next section how such an algorithm works. In the remainder of this section, we will discuss how to solve Problem (4.1.5.5) efficiently.

4.1.5.3 Dual of Proximal Problem

To fully exploit the new strong convexity of our problem, we will now derive its dual. First, this will give us a smooth optimization problem to work with. Second, we will be able to compute a primal-dual gap to give us a way to monitor the convergence of our algorithm.

To do so, we introduce the Lagrangean variables $\boldsymbol{\beta}$. They are associated with the constraint $\sum_{k,b} \boldsymbol{\lambda}_{k,b} = \mathbf{c}$ and can be used to derive the following problem:

$$\begin{aligned} \min_{\boldsymbol{\beta}} \max_{\boldsymbol{\lambda}_{k,b}} \min_{\mathbf{x}_{k,b}} & \sum_{k,b} {\boldsymbol{\lambda}_{k,b}}^\top \mathbf{x}_{k,b} - \sum_{k,b} \frac{\eta}{2} ||\boldsymbol{\lambda}_{k,b} - \widetilde{\boldsymbol{\lambda}}_{k,b}||^2 + \boldsymbol{\beta}^\top (\sum_{k,b} \boldsymbol{\lambda}_{k,b} - \mathbf{c}) \\ & \text{s.t. } \forall k, b, \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.5.6}$$

Using strong duality of our problem, we can reorder the min and max operations and solve the unconstrained problem on $\boldsymbol{\lambda}_{k,b}$ by setting the gradient of the objective function with respect to this variable to 0. By doing so we obtain the following Karush-Kuhn-Tucker (KKT) conditions:

$$\boldsymbol{\lambda}_{k,b} = \widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{\mathbf{x}_{k,b} + \boldsymbol{\beta}}{\eta}. \tag{4.1.5.7}$$

We use this to simplify Eq. (4.1.5.6) and obtain the following problem:

$$\begin{aligned} \min_{\boldsymbol{\beta}, \mathbf{x}_{k,b}} \; & L(\mathbf{x}_{k,b}, \boldsymbol{\beta}) = \sum_{k,b} (\widetilde{\boldsymbol{\lambda}}_{k,b}^\top \mathbf{x}_{k,b} + \frac{1}{2\eta} ||\mathbf{x}_{k,b} + \boldsymbol{\beta}||^2) \\ & \text{s.t. } \forall k, b, \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.5.8}$$

This problem is interesting for two reasons. First, the optimization over $\boldsymbol{\beta}$ is unconstrained, quadratic and strictly convex and so can be performed in closed form. Moreover, depending on how we want to solve this problem, $\boldsymbol{\beta}$ can be eliminated completely to get a single minimization problem over $\mathbf{x}_{k,b}$. Second, the problem on $\mathbf{x}_{k,b}$ is now strictly convex and even though the objective is not linear, we will see in the next section, that we can do projection-free optimization on it using only the min oracle defined earlier.

This formulation is related to the Augmented Lagrangian Methods discussed in [113]. In particular, it proves that if the Problem (4.1.5.6) is solved exactly, then a finite number of updates of the proximal term is sufficient to solve Problem (4.1.4.5) exactly.

Example: QPBO To have a better idea of what this problem looks like, we can write it for the particular case of the QPBO roof duality relaxation. Using the same notations as before and $\boldsymbol{\beta}$ and $\boldsymbol{\rho}$ for the new dual variables, we have the following problem:

$$\begin{aligned} \min_{\mathbf{y}_b, \mathbf{z}_b, \boldsymbol{\beta}, \boldsymbol{\rho}} \; & \sum_b [(\widetilde{\boldsymbol{\lambda}}_b, \widetilde{\boldsymbol{\mu}}_b)^{C_b\top} (\mathbf{y}_b, \mathbf{z}_b)^{C_b} + \frac{1}{2\eta} ||(\mathbf{y}_b + \boldsymbol{\beta}, \mathbf{z}_b + \boldsymbol{\rho})^{C_b}||^2] \\ \text{s.t. } & \forall b, \mathbf{g}_1((\mathbf{y}_b, \mathbf{z}_b)^{C_b}) \geq \mathbf{0}. \end{aligned} \tag{4.1.5.9}$$

4.1.6 Optimization

We now turn to the core of this paper: the algorithm to solve the problems formulated above. We will first present the different approaches that can be used to solve Eq. (4.1.5.8). Then we will discuss key details of the algorithm that have a large impact in the final performances in practice: the stopping criterion and the choice of η.

4.1.6.1 Algorithm

4.1.6.1.1 Block Coordinate Descent The first approach we present to solve Eq. (4.1.5.8) is to perform Block Coordinate (BC) descent on the block $\boldsymbol{\beta}$ on the one hand and $\mathbf{x}_{k,b}$ on the other hand. Given that the problem we are solving is convex and smooth, BC will converge to the optimal solution.

Block $\boldsymbol{\beta}$ Considering that $\mathbf{x}_{k,b}$ is fixed, optimizing over this block is a simple unconstrained convex minimization. To solve it, we set the gradient with respect to $\boldsymbol{\beta}$ to $\mathbf{0}$.

$$\begin{aligned} \mathbf{0} &= \frac{\partial L}{\partial \boldsymbol{\beta}} \\ \implies \mathbf{0} &= \sum_{k,b} \frac{1}{\eta} (\mathbf{x}_{k,b} + \boldsymbol{\beta}) \\ \implies \forall j,\ \boldsymbol{\beta}^j &= \frac{-1}{N_j} \sum_{t,b \text{ s.t. } j \in C_{t,b}} {\mathbf{x}_{k,b}}^j \\ \implies \boldsymbol{\beta} &= -\mathbf{n} \odot \sum_{k,b} \mathbf{x}_{k,b}. \end{aligned} \tag{4.1.6.1}$$

Here, N_j is the number of subproblems the j-th variable belongs to, $\mathbf{n}$ is the vector such that $\mathbf{n}^j = \dfrac{1}{N_j}$.

Block $\mathbf{x}_{k,b}$ When $\boldsymbol{\beta}$ is fixed, we optimize a convex objective over a compact convex set. An efficient, projection free algorithm to solve these problems is the FW algorithm. It consists of computing the conditional gradient of function and then performing a step in that direction. To compute the conditional gradient, we first compute the gradient of the objective as:

$$\frac{\partial L}{\partial \mathbf{x}_{k,b}} = \widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{1}{\eta}(\mathbf{x}_{k,b} + \boldsymbol{\beta}) = \boldsymbol{\lambda}_{k,b}. \tag{4.1.6.2}$$

Here, the last inequality is given by Eq. (4.1.5.7).

The conditional gradient is computed by minimizing the linearisation of the objective under the same constraint as the original problem. This minimization can be done independently for each subproblem as they each work with a different set of variables. In particular, for a given subproblem k, b, we solve the following problem:

$$\begin{aligned} \widetilde{\mathbf{x}}_{k,b} = \underset{\mathbf{x}_{k,b}}{\operatorname{argmin}} \ & \boldsymbol{\lambda}_{k,b}{}^{\top} \mathbf{x}_{k,b} \\ \text{s.t. } & \mathbf{g}_k(\mathbf{x}_{k,b}{}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.6.3}$$

This problem corresponds exactly to the min oracle defined in Section 4.1.4. Moreover, as was expected given the results from [48], it is equal to the supergradient of Problem (4.1.5.5), which is the primal of the problem we are currently solving.

Finally, a step-size γ needs to be chosen such that the new $\mathbf{x}_{k,b}$ is given by $\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}$. Finding the optimal γ by line search is equivalent to solving the following problem:

$$\min_{\gamma \in [0,1]} L(\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}, \boldsymbol{\beta}). \tag{4.1.6.4}$$

This problem can be solved by first expanding the objective of this problem as:

$$\begin{aligned} & L(\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}, \boldsymbol{\beta}) \\ =& \sum_{k,b} [\widetilde{\boldsymbol{\lambda}}_{k,b}^{\top}(\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}) + \frac{1}{2\eta}||(\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}) + \boldsymbol{\beta}||^2] \\ =& \sum_{k,b} [\gamma\widetilde{\boldsymbol{\lambda}}_{k,b}^{\top}(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}) + \widetilde{\boldsymbol{\lambda}}_{k,b}^{\top}\mathbf{x}_{k,b} + \frac{1}{2\eta}||\gamma(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}) + \mathbf{x}_{k,b} + \boldsymbol{\beta}||^2]. \end{aligned} \tag{4.1.6.5}$$

As illustrated by Eq. (4.1.6.5), the problem boils down to minimizing a one dimensional quadratic function on a compact set. We can solve this in close form by finding the point where the gradient is 0 and then projecting onto the

constraint set. The first step can be done as follows:

$$
\begin{aligned}
0 &= \frac{\partial L(\gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}, \boldsymbol{\beta})}{\partial\gamma} \\
0 &= \sum_{k,b}[\widetilde{\boldsymbol{\lambda}}_{k,b}^{\top}(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}) + \frac{\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}}{\eta}(\gamma(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}) + \mathbf{x}_{k,b} + \boldsymbol{\beta})] \\
\gamma \sum_{k,b} \frac{||\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}||^2}{\eta} &= \sum_{k,b} (\widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{\mathbf{x}_{k,b} + \boldsymbol{\beta}}{\eta})^{\top} (\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}) \\
\gamma &= \frac{\eta \sum_{k,b} \boldsymbol{\lambda}_{k,b}{}^{\top}(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b})}{\sum_{k,b} ||\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}||^2}.
\end{aligned}
\tag{4.1.6.6}
$$

The closed form solution for the line search is given by projecting this value onto the set $[0, 1]$. We define this projection as the function $p_{[0,1]}$.

In practice, we will only perform one step of FW before optimizing the other block as it gives the best experimental results. The complete algorithm is presented in Algorithm (5).

Algorithm 5: Minimization of an LP using a block coordinate approach to optimize the dual of the proximal problem of the LD.

Data: $\mathbf{c}$ and $\mathbf{g}_k$
Initialize $\boldsymbol{\lambda}_{k,b}$ with any valid solution
Initialize $\mathbf{x}_{k,b} = \text{argmin}_{\mathbf{x}_{k,b}} \boldsymbol{\lambda}_{k,b}{}^{\top}\mathbf{x}_{k,b}$
do
  $\widetilde{\boldsymbol{\lambda}}_{k,b} = \boldsymbol{\lambda}_{k,b}$
  do
    Optimal step over beta
    $\boldsymbol{\beta} = -\mathbf{n} \odot \sum_{k,b} \mathbf{x}_{k,b}$
    Compute conditional gradient
    $\boldsymbol{\lambda}_{k,b} = \widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{\mathbf{x}_{k,b} + \boldsymbol{\beta}}{\eta}$ $\widetilde{\mathbf{x}}_{k,b} = \text{argmin}_{\mathbf{x}_{k,b},\, \mathbf{g}_k(\mathbf{x}_{k,b}{}^{C_{k,b}}) \geq \mathbf{0}}$ $\boldsymbol{\lambda}_{k,b}{}^{\top}\mathbf{x}_{k,b}$
    Compute the optimal step-size
    $\gamma = p_{[0,1]}\Big(\frac{\eta \sum_{k,b} \boldsymbol{\lambda}_{k,b}{}^{\top}(\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b})}{\sum_{k,b} ||\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b}||^2}\Big)$
    One step of Frank Wolfe
    $\mathbf{x}_{k,b} = \gamma\widetilde{\mathbf{x}}_{k,b} + (1-\gamma)\mathbf{x}_{k,b}\ \forall k, b$
  while $\gamma > 0$;
while $\widetilde{\boldsymbol{\lambda}}_{k,b} \neq \boldsymbol{\lambda}_{k,b}$;

4.1.6.1.2 Pure Frank Wolfe The second approach we consider to solve this problem is to completely eliminate the $\boldsymbol{\beta}$ variable from the problem using the closed for solution from Eq. (4.1.6.1). By doing so, the resulting problem is a simple minimization of a convex function on a compact set. The new problem

can be written as:

$$\begin{aligned} \min_{\mathbf{x}_{k,b}} \; & L'(\mathbf{x}_{k,b}) = \sum_{k,b} (\widetilde{\boldsymbol{\lambda}}_{k,b}^{\top} \mathbf{x}_{k,b} + \frac{1}{2\eta} ||\mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} \mathbf{x}_{k',b'}||^2) \\ & \text{s.t. } \forall k, b, \, \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.6.7}$$

We can thus apply the FW algorithm to solve this problem without any block coordinate scheme. Similarly to the previous section, we first compute the linearisation of the objective functions as:

$$\begin{aligned} & \frac{\partial L'}{\partial \mathbf{x}_{k,b}} \\ =& \widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{1}{\eta} (\mathbf{1} - \mathbf{n}) \odot (\mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} \mathbf{x}_{k',b'}). \end{aligned} \tag{4.1.6.8}$$

Note that this is very similar to the linearisation in Eq. (4.1.6.2) with only the scaling of the second term with $\mathbf{n}$.

The conditional gradient is obtained by solving the following problem using the min oracle:

$$\begin{aligned} \widetilde{\mathbf{x}}_{k,b} = \underset{\mathbf{x}_{k,b}}{\operatorname{argmin}} \; & \frac{\partial L'}{\partial \mathbf{x}_{k,b}}^{\top} \mathbf{x}_{k,b} \\ & \text{s.t. } \mathbf{g}_k({\mathbf{x}_{k,b}}^{C_{k,b}}) \geq \mathbf{0}. \end{aligned} \tag{4.1.6.9}$$

Finally the optimal step-size can be obtained the same way as before. In particular, the step-size before projection can be computed as:

$$\gamma = \frac{\eta \sum_{k,b} [\widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{1}{\eta} (\mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} \mathbf{x}_{k',b'})]^{\top} (\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b})}{\sum_{k,b} ||\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} (\widetilde{\mathbf{x}}_{k',b'} - \mathbf{x}_{k',b'})||^2}. \tag{4.1.6.10}$$

It is interesting to note that this formulation, contrary to the previous one, solves the problem by only using the FW algorithm. This means that acceleration methods for FW such as the ones presented in [114] can be used, at the cost of extra book keeping, to obtain a linear convergence rate for this algorithm. The complete

algorithm is presented in Algorithm (6).

Algorithm 6: Minimization of an LP using the Frank Wolfe algorithm to optimize the dual of the proximal problem of the LD.

Data: $\mathbf{c}$ and $\mathbf{g}_k$

Initialize $\boldsymbol{\lambda}_{k,b}$ with any valid solution Initialize $\mathbf{x}_{k,b} = \text{argmin}_{\mathbf{x}_{k,b}} \boldsymbol{\lambda}_{k,b}{}^{\top}\mathbf{x}_{k,b}$

do

  $\widetilde{\boldsymbol{\lambda}}_{k,b} = \boldsymbol{\lambda}_{k,b}$

  do

    Compute conditional gradient

    $\dfrac{\partial L'}{\partial \mathbf{x}_{k,b}} = \widetilde{\boldsymbol{\lambda}}_{k,b} + \dfrac{1}{\eta}(\mathbf{1} - \mathbf{n}) \odot (\mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} \mathbf{x}_{k',b'})$

    $\widetilde{\mathbf{x}}_{k,b} = \text{argmin}_{\mathbf{x}_{k,b},\, \mathbf{g}_k(\mathbf{x}_{k,b}{}^{C_{k,b}}) \geq \mathbf{0}} \dfrac{\partial L'}{\partial \mathbf{x}_{k,b}}{}^{\top} \mathbf{x}_{k,b}$

    Compute the optimal step-size

    $\gamma = p_{[0,1]}\Big(\dfrac{\eta \sum_{k,b} [\widetilde{\boldsymbol{\lambda}}_{k,b} + \frac{1}{\eta}(\mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} \mathbf{x}_{k',b'})]^{\top} (\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b})}{\sum_{k,b} ||\widetilde{\mathbf{x}}_{k,b} - \mathbf{x}_{k,b} - \mathbf{n} \odot \sum_{k',b'} (\widetilde{\mathbf{x}}_{k',b'} - \mathbf{x}_{k',b'})||^2}\Big)$

    One step of Frank Wolfe

    $\mathbf{x}_{k,b} = \gamma \widetilde{\mathbf{x}}_{k,b} + (1 - \gamma)\mathbf{x}_{k,b}\ \forall k, b$

  while $\gamma > 0$;

while $\widetilde{\boldsymbol{\lambda}}_{k,b} \neq \boldsymbol{\lambda}_{k,b}$;

4.1.6.2 Stopping Criterion

The algorithm described above is composed of two loops. The outer loop corresponds to the proximal optimization where the proximal term $\widetilde{\boldsymbol{\lambda}}_{k,b}$ is updated at each step. The inner loop solves Problem (4.1.5.8) using one of the two algorithms described in the previous subsection. A key aspect for the speed and precision of the final algorithm are the stopping criterion used for each of these two loops.

Outer Loop Given that our algorithm solves the problem in the dual, at every step of the algorithm we have a lower bound on the optimal value of the original LP. If the user provide a function to project onto the set of constraints $\forall k, b,\ \mathbf{g}_k(\mathbf{x}^{C_{k,b}}) \geq \mathbf{0}$, we can use the difference between the primal and dual value as a criterion for convergence of the algorithm. An LP is a convex problem and exhibits strong duality, meaning that for all problems, this gap will converge to 0. If the user requires a specific precision for the solution, for example that the returned lower bound is less than ϵ away from the optimal value, we can stop the algorithm when the primal-dual gap is below ϵ.

For applications such as neural network verification where the user evaluates whether or not the optimal value is negative, the lower bound given by the dual can be used to stop the solver early if its value becomes positive.

Two other criteria can be used to detect exact convergence of the outer loop. First, if the inner algorithm does not move and returns $\boldsymbol{\lambda}_{k,b} = \widetilde{\boldsymbol{\lambda}}_{k,b}$, it means that the algorithm has converged. Second, we can use the property of LD that if, for a given set of dual variables, the solutions of the subproblems are consistent, meaning that they verify $\forall t, b,\ \mathbf{x}_{k,b} = \mathbf{x}$, then this $\mathbf{x}$ is the optimal solution to the original LP. For a problem with subproblems that have a limited number of possible solutions, this criterion can be useful to quickly identify the optimal solution to the original problem.

Inner Loop For the problem solved in the inner loop, that is Problem (4.1.5.5), we also have both a primal and a dual value, though different from the ones in the previous paragraph as they consider the proximal term. For this problem, we know that the gap will go to 0 as well and the convergence speed is directly linked to the value of η. The precision required for this gap must be small enough to prevent unwanted termination of the outer loop. Indeed, too high a threshold will stop the inner optimization early and might prevent the inner loop from moving at all. This, in turn, would stop the outer loop before reaching the desired precision.

It is also interesting to note that the optimal step-size γ that we compute to update $\mathbf{x}_{k,b}$ can be used to detect convergence as it will be equal to 0 at the optimal solution.

4.1.6.3 Hyperparameter Tuning

The only hyperparameter that needs to be considered in the complete algorithm is the weight of the proximal term η. Contrary to the step-size hyperparameter of the projected gradient algorithm, this term has a well understood influence on the algorithm. On the one hand, the value of η in Problem (4.1.5.5) is exactly the strong convexity coefficient of the objective function. We thus know that this value has a direct impact on the convergence rate of different optimization algorithms applied to this objective function. A large value of η corresponds to an easier optimization problem, while a small η will make the function less strongly convex and thus harder to optimize. On the other hand, this hyperparameter also scales the quadratic part of the objective function and so will change how far the proximal problem is from the original one. In that case, a small η is desirable to ensure that the proximal problem is close enough to the original one. In practice, a trade-off between an easy inner problem and an inner problem close to the original problem needs to be made.

The decision on the value of η can be made in practice as three regimes can be identified:

- η is too large if the inner problems are solved in a few iterations but they converge to very different values from one inner problem to the other.

- η has a correct value if a small number of outer iterations is needed to reach convergence and the inner problem are solved to convergence in a reasonable amount of steps.

- η is too small if the inner problem take a very long time to converge.

These different regimes can easily be identified by monitoring the convergence speed on the inner problems and their final values. We also notice that there is a large range of η that provides similar convergence speed as an increase of η will increase the number of outer steps but speed up the inner iterations. Similarly, a decrease in η will slow down the inner loops but reduce the number of outer steps.

We also observe in practice that the value of η can be reused for different instances of the same problem. Indeed, the value need only be changed when the values of the problem parameters change in scale or the size of the problem changes. For example, for the QPBO problem, we define η as an affine function of the number of variables for some experiments, and a constant value for others as will be detailed in the next section. Similarly, for all the neural network verification experiments, we set η to a constant value for all the experiments as it gives good results for all the networks we consider. This is in contrast to the methods that use the projected supergradient method for which a new learning rate schedule is designed for every task to which it is applied.

4.1.7 Experiments

We now present the experimental comparison of our solver with both general LP solver and specialized solvers. For brevity, we will refer to Algorithm (5) as ProxBC and to Algorithm (6) as ProxFW . We first discuss the implementation details of the algorithms and then compare them both to a state of the art general LP solver, Gurobi [98], and specialized solvers when one exists.

4.1.7.1 Implementation

The implementation has been done in Python using PyTorch [10] and will be made public. Both algorithms presented above can be seen as only performing element-wise or simple reduction operations on matrices. This allows the general implementation to be both simple (few hundred lines of code) and efficient while using transparently CPU, GPU or any supported hardware for acceleration. A GPU-friendly implementation is obtained by using large tensors to store the different vectors. Another speed-up is obtained by taking advantage of the asynchronous nature of the GPUs and evaluating the stopping criterion only after a given number of steps, set to 5 in all experiments.

Using Python allows the implementation to be modular and to require only a limited amount of code to solve a new problem. In particular, to solve a new problem, the user needs to specify how to store as tensors "primal-like" variables such as $\mathbf{c}$ or $\mathbf{x}$ and "dual-like" variables such as $\boldsymbol{\lambda}_{k,b}$ or $\mathbf{x}_{k,b}$. Three functions then need to be implemented:

- minOracle($\mathbf{c}_{k,b}$) that solves the problem defined in Eq. (4.1.4.11).
- toDual($\mathbf{x}$) that converts a primal variable to a dual one as: $\forall k, b,\ \mathbf{x}_{k,b} = \mathbf{n} \odot \mathbf{x}$.
- toPrimal($\mathbf{x}_{k,b}$) that converts a dual variable to a primal one as: $\mathbf{x} = \sum_{k,b} \mathbf{x}_{k,b}$.

4.1.7.2 Solving Neural Network Verification

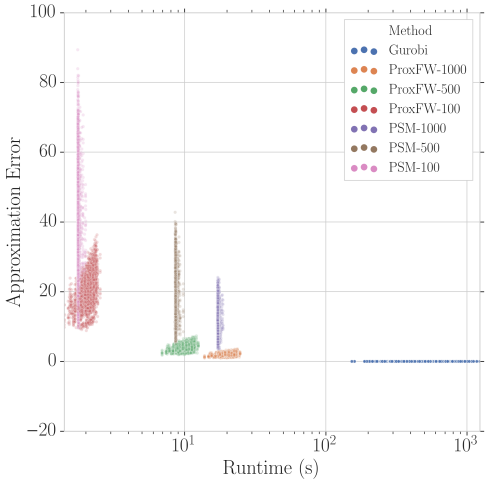
Solvers For the problem of neural network verification, we use the min oracle described in Section 4.1.4.3. The primal variables contain the output of each linear layer while the dual variables contain two copies of them, one associated with the previous layer, one with the next layer. The conversion between primal and dual variables can be done, in one direction, by copying and dividing by 2 the primal variable into the two copies that form the dual variables and, the other direction, by adding element-wise the dual variables to obtain the primal ones. Even though a feasible primal solution is easy to obtain for this problem using a forward pass through the neural network, this primal solution is not of good quality as it ignores most of the dual variables. This is not a strict limitation in practice as the task requires us to solve a large number of such problem but does not require a fixed precision. Indeed, a constant number of inner and outer iterations is used to efficiently estimate the extrema for each layers. In all the following experiments, a fix number of 5 inner iterations is used and the number of outer iterations is changed to perform a trade-off between speed and precision.

The solver's names are suffixed with the total number of iterations performed. The value of η is fixed to 100 for all experiments.

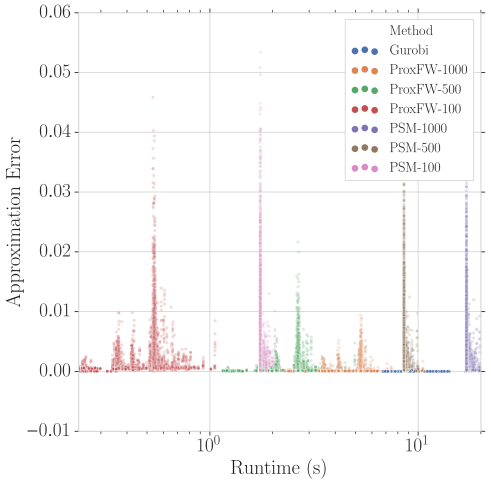
As no specialized solvers exist for this problem, we compare it only with Gurobi using the Python interface and the default parameters. We also compare it to the original projected supergradient method, called PSM below, to see the impact of the proximal minimization on the final speed and precision. Note that the Gurobi solver runs on a 12 core i7-7930K CPU while both the proximal and projected supergradient methods run on a Titan X card.

Experimental Setting To see the behaviour of the solver on different problems, we consider two neural networks in these experiments. Both networks are trained using the CIFAR-10 dataset [115]. The first one that we call the baseline network is trained using classical methods: stochastic gradient descent and cross entropy loss. The second one that we call WK network is trained to be resilient to adversarial attacks following the procedure from [116]. As we will see below, they define LPs with different properties in practice. Considering a perturbation of at most 8/255 in infinity norm of the input, we use the LP solver to compute how high the scores can be for all labels that that are not the ground truth. This is used by practitioners to measure the robustness of the classification for particular samples. We measure both the runtime and the obtained highest score for a subset of the test set of the CIFAR-10 dataset. A large number of LPs is solved for every point as the extrema for each hidden unit needs to be computed. All the problems corresponding to a single layer are solved at the same time for all solvers but Gurobi which does not support solving a batch of problems.

Results In Fig. 4.1a, we plot the maximum over all the incorrect labels of the difference between the computed and the exact highest score as a function of the time required to obtain it for the baseline neural network. We call the difference between the computed and exact score the approximation error. All the differences are positive, meaning that all the solvers will either return the exact value or a lower bound of that value. First, by comparing the Prox and PSM solvers, we can see that they have a similar runtime when running for the same number of iterations but the proximal solver consistently gives more accurate results. Second, Gurobi always gives the exact solution but the runtime is prohibitively long for such problem and there is a larger variance in runtime for this solver. The proximal solver allows us to get accurate solutions an order of magnitude faster. Moreover, approximate solutions can be obtained very quickly by using a small number of



(a) Illustration of trade-off between speed and precision for baseline neural network bound computation. Each point correspond to a test sample from CIFAR-10. While significantly slower, the Gurobi solver gives the exact solution. Both the Prox and PSM solver provide a trade-off between speed and precision. The Prox solver is consistently better than the PSM solver as it is more precise while having the same runtime.



(b) Illustration of trade-off between speed and precision for WK neural network bound computation. Each point correspond to a test sample from CIFAR-10. All the problems are easy to solve and all the solvers reach a very low approximation error. The Prox solver with a small number of iteration is the fastest while providing exact solutions for many samples and so is the best suited for this problem.

iterations like ProxFW-100. This example shows that the proximal solver, even though it does not provide a provably exact solution, is able to give an accurate solution an order of magnitude faster than a general purpose solver. It also shows that the proximal scheme allows the optimization to be more efficient and provides more precise solutions without impacting the runtime of the algorithm compared to the classic projected supergradient algorithm.

We present in Fig. 4.1b the same information as the previous figure for the WK neural network. The LPs defined by this neural network are significantly simpler to solve. Indeed, all solvers are able to solve these problems faster and with low approximation error. In particular, Gurobi is two order of magnitude faster to solve this problem compared to the one using the baseline neural network. We observe in this case that a Prox solver with a small number of iteration is able to find an accurate solution to the problem significantly faster than the Gurobi solver.

4.1.7.3 Solving the Roof Duality Relaxation of QPBO

Solvers For the roof duality relaxation of the QPBO problem, we use the min oracle described in Section 4.1.4.3. For the primal variables, we store $\mathbf{y}$ and

$\mathbf{z}$ in two separate vectors. For the dual variables, we store $\mathbf{y}_{1,b}$ and $\mathbf{z}_{1,b}$ in two separate vectors. The toDual and toPrimal functions are implemented using a mapping to know which pair each node belongs to. We fix η to a value of 0.33 for all the experiments with Barabasi Albert (BA) graphs, detailed below. For the Erdos Renyi (ER) graphs, we set η as a linear function of the problem size N as $\eta = 0.735N - 48.15$. This value was chosen by observing the behaviour of the solver for 6 problems of different sizes and performing a simple linear regression on the manually tuned η for each size. We use the stopping criterion described in Section 4.1.6.2 together with a maximum number of iterations M. We consider three values of M: 20, 250 and 2000 and describe the corresponding results below.

As baseline in this experiment, we consider Gurobi for which we encode the original Problem (4.1.4.3) using the Python interface and use the default parameters. The second baseline we consider is the solver by [106], named MaxFlow below. As opposed to Gurobi, [106] is a specialized solver based on maxflow. Note that neither of the baseline offer the possibility to solve multiple problems at the same time and so they solve them sequentially. Moreover, they both use a CPU for computation while the Prox solver uses a GPU.

Experimental Setting For these experiments, we consider the QPBO instances arising from solving a weighted version of the Maximum Independent Set (MIS) problem. An independent set of a graph is a subset of the vertices such that no edge has both its endpoints in the subset. The MIS problem consists in finding the independent set that has the maximum weight. We generate two types of random graphs on which we will solve this problem. First, BA graphs [117] with the connectivity set to 4. Second, ER graphs [118] with the probability parameter set to 0.4. Each node of the graph has a corresponding weight, chosen uniformly at random from the interval $[0, 1]$.

Precision Results In this experiment, we measure the impact of the maximum number of iterations enforced on the Prox solver for the two problems at hand. As can be seen in Table 4.1, for both algorithms, increasing the maximal number of iteration increases the precision. Moreover, the number of iterations required to reach a given precision will change depending on the problem at hand. We can also see that the ProxBC and ProxFW solvers behave differently for different problems. Indeed, depending on the problem and the number of maximum iterations considered, one solver or the other will be more precise and so they are both useful for different applications.

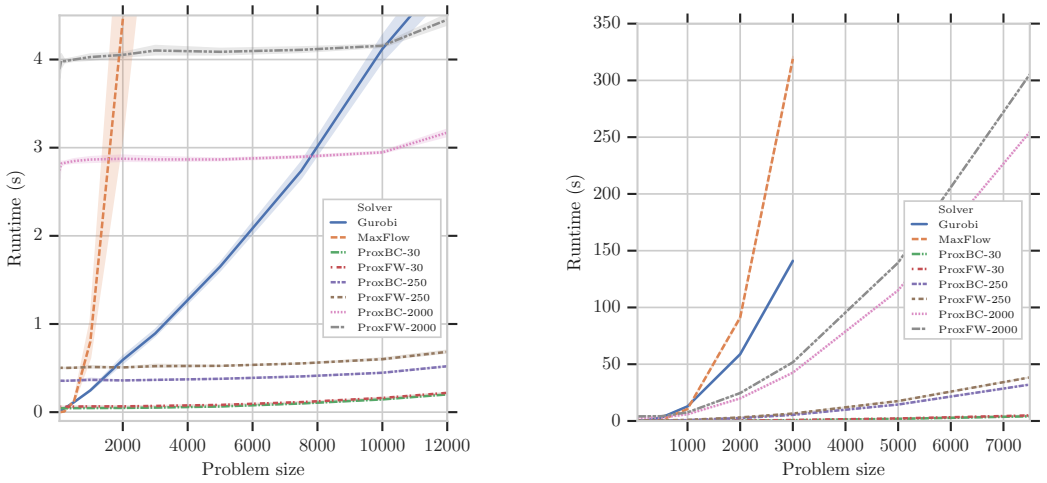
	BA				ER			
Size	100	200	1000	10000	100	200	1000	3000
ProxBC-30	9.41	9.85	9.87	10.1	1.37	1.02	2.04	2.37
ProxBC-250	0.986	1.00	1.05	0.975	0.335	0.208	0.131	0.133
ProxBC-2000	0.315	0.241	0.125	0.123	0.0481	0.0274	0.0209	0.0201
ProxFW-30	8.24	8.22	8.33	8.36	0.967	0.850	2.01	2.34
ProxFW-250	1.39	1.38	1.42	1.30	0.333	0.200	0.130	0.133
ProxFW-2000	0.348	0.279	0.160	0.157	0.0477	0.0273	0.0209	0.0201

Table 4.1: Average relative error given in percent for problems of different sizes. The precision is solver and problem dependent. In all cases, it strictly increases with the number of iterations.

Problem Size Scaling Results We now compare how the different solvers scale when the size of the problems increase. In Fig. 4.2, we plot the runtime to solve one as a function of the size of the problem. First for BA graphs, the number of edges in the graph is equal to the connectivity multiplied by the number of nodes. This means that the number of variables in our sparse formulation is linear in the size of the problem. We observe that the MaxFlow solver is very efficient to get exact solutions for small problems. Unfortunately, it does not scale well with the size of the problem, mainly because the graph it runs the maxflow algorithm on grows quickly and is, therefore, expensive to build explicitly. The Gurobi solver is scaling much better for this problem and is able to solve the large problems exactly in a reasonable amount of time. The Prox solver runtime is only increasing slightly as the GPU is underused for very small problems and is able to solve larger problems with a similar runtime. For this problem, the Prox solver is able to scale much better than both baselines by efficiently using the compute capabilities of the GPU.

Second, for ER graphs, the number of edges is not a linear function of the number of nodes. All the solvers thus have a larger runtime and do not scale as well as for BA graphs. Note that the Gurobi solver exhausts the 64GB of the machine when working with problems larger than 3000. Here again, we see that the Prox solver is able to scale better than the baseline even though it is slower for very small problems.

These experiments show that even though the overhead of the Prox solver is significant and detrimental for small problems, it is able to outperform the baselines in term of runtime by an order of magnitude when solving large problems. We also note that the sparsity of the $\mathbf{P}$ matrix has an important impact on the solvers runtime and our solver is able to efficiently use this sparsity to scale better to large problems.



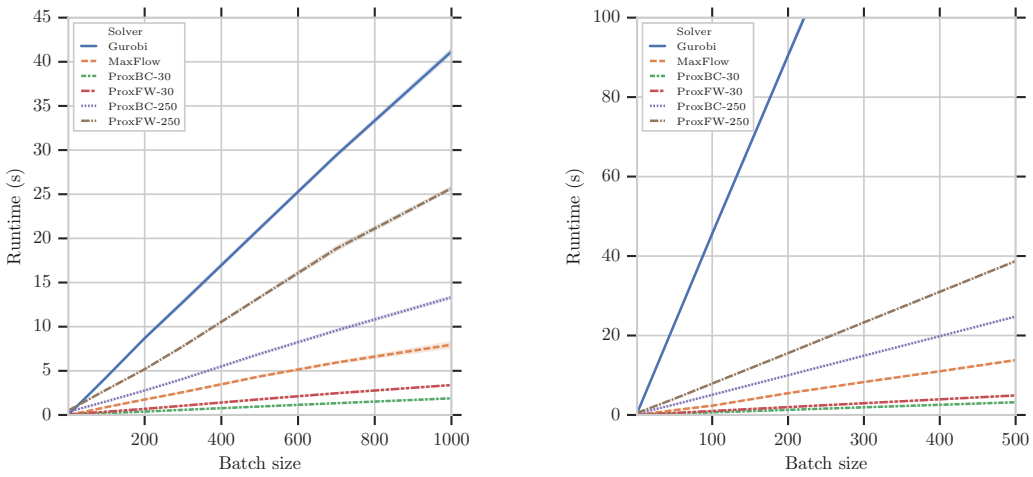
(a) BA graphs **(b)** ER graphs

Figure 4.2: Runtime of the optimizer when the size of the QPBO problems increases. The Prox solver is slower for very small problems but scales much better than both baseline solvers.

Batch Size Scaling Results In these experiments, we explore the behaviour of the different solvers when multiple problems are solved in batch. To do so, we generate batches of independent problems of size 200. The results in Fig. 4.3 show, as expected, that all solvers scale linearly with the batch size. It is important to note though, that the slopes are very different for different solvers. Indeed, while the runtime for Gurobi and MaxFlow are approximatively their runtime for one sample multiplied by the batch size, the Prox solver behaves differently. This is expected for such small problems as the runtime is heavily influenced by the overhead of using the GPU. Indeed, for this solver, the runtime is not a direct function of how many operations need to be done, but also how efficiently data can be read from memory and how efficient the communication with the GPU is. This experiment show that even for smaller problems, the Prox solver is interesting if a large number of problems need to be solved at the same time as the overhead of using GPUs is amortised.

4.1.8 Conclusion

We consider the hard task of designing a solver for a large class of LP while exploiting their specific structure to obtain an efficient solver. We identify problems that separate into blocks for which the corresponding subproblems can be solved efficiently. By combining two well known approaches, namely Lagrangean Decomposition and Proximal optimization, we derive an simple yet efficient solver for block bounded LPs. This novel derivation and the tight coupling of the solver with the task it



(a) BA graphs **(b)** ER graphs

Figure 4.3: Runtime of the optimizer when the number of independent problems (batch size) increases for a problem size of 200. The Prox solver has sub-linear scaling for increasing batch size for such small problems while the baseline quickly reach linear scaling.

solves leads to a state of the art solver as shown in the experimental section. The block bounded problems considered in this paper are common in practice and a simple general solver for them, such as the one presented here, enables researchers to use them as subroutine to solve a broad range of new problems.

We identify two directions to improve further the solver. First, methods related to proximal minimization such as the Alternating Direction Method of Multipliers (ADMM) could be used for the optimization. Different optimization algorithms have different behaviours and will likely allow to speed up the solver for some problems. Second, even though the selection of the η hyperparameter is simple for the examples we consider, no general rule ensure that this is true for all problems. Machine Learning could be used to find the best value without manual search given the strong correlation that we observe in practice between η, on the one hand, and the problem size and coefficients of the linear constraints and objective, on the other hand.

5 Discussion

5.1 Contributions of the Thesis

The intersection of optimization and machine learning is a promising area of research. This thesis adds to the growing literature that backs this idea.

- In Chapter 2, we show that optimization is of decisive importance for machine learning algorithms. Indeed, using the example of Computer Vision tasks modelled with dense CRFs, we show that a specialized algorithm allows to solve the MAP estimation problem while general approaches are either too slow or not accurate enough. To do so, we consider the high quality LP relaxation of the problem. We solve this relaxation using a specialized proximal solver that is based on a fast filtering algorithm and the FW algorithm for which the optimal step-size can be analytically computed. Recent work, such as [11], builds on these results and introduces higher order terms to the CRF to improve the quality of the solutions obtained for semantic segmentation.

- In Chapter 3, we explore two ideas to use machine learning to improve optimization. First, we present an example of how to reformulate an optimization problem into a machine learning one. Indeed, we consider the problem of program synthesis and reformulate it as learning a program that is both fast and correct. The correctness in this case is enforced as a supervised learning tasks where example inputs and outputs are provided to the learning algorithm. We introduce a new differentiable model of the execution of the program which enables us to use Deep Learning tools to solve the learning

problem. Second, we take a different approach and build on top of existing state-of-the-art optimization algorithms and use machine learning to improve them. We show, for the problem of super-optimization, that a state-of-the-art solver such as Stoke [119] can be improved upon by using machine learning. Indeed, using reinforcement learning techniques, we show that the exploration of the space of programs can be improved significantly from the original heuristic.

- In Chapter 4, we present a new solver for block bounded LPs. Such LPs are common when working with machine learning-related problems. We develop this solver such that it can be used within machine learning frameworks easily while being efficient. Indeed, the proposed solver is able to leverage acceleration hardware such as GPUs to scale to larger problems. Its simplicity and ability to easily control the trade-off between speed and precision allows a tight integration within larger algorithms. The solver itself is built using classical approaches such as Lagrangean Decomposition and proximal minimization and uses the FW algorithm with an optimal step-size computed in closed form to perform fast optimization. We demonstrate the efficiency of the solver using two tasks. First, it allows to speed-up neural network verification which requires a large number of LPs to be solved. Second, it allows fast lower bound estimation for the QPBO problem by solving the roof duality relaxation. It is used as a subroutine by a BB algorithms that aims at solving QPBO problems exactly.

5.2 Future Work

This work opens new areas of research. Indeed, a natural extension of the work presented in Chapter 3 is to apply similar approaches to a more general family of optimization problems. We showed that solvers for specific complex tasks can be improved upon using machine learning but no procedure exists to solve a large class of problems while being able to automatically adapt to specific instances. Following ideas developed in this thesis, we consider Integer Linear Programs (ILP) to be an interesting class of NP-hard problems to tackle. We now discuss two possible directions toward this goal.

Learning Heuristics for Branch and Bound First, in the field of optimization, BB techniques are commonly used to solve discrete problems. Such algorithms can be improved upon using machine learning. In particular, the branching, where a variable of the problem is chosen to have its value fixed, hence creating a set of new subproblems, is always based on heuristics. These heuristics are tuned for different problems and crucial to the efficiency of the algorithm. Machine learning techniques can be used to improve these heuristics as discussed, for example, in [120]. Unfortunately, the training procedure is expensive and limited to very small problems. A solver such as the one presented in Chapter 4 can be used to tackle larger problems. Indeed such a solver would allow faster bound computations, allowing larger problems to be solved. Moreover the increase in the speed of the data collection would allow the use of more advanced learning algorithms.

Learning to Relax Discrete Problem Second, a more ambitious approach would be not to use BB to solve such problem but try and find a continuous relaxation which is equivalent to the original ILP. Indeed, the ILP can, for example, be approximated by a sequence of LPs that have an increasing number of constraints and that are increasingly close to the ILP. An example of such a hierarchy is the Sherali-Adams (SA) hierarchy of LPs [121]. Others hierarchies could be considered, such as the Lasserre hierarchy of Semi-Definite Programs (SDP) [122] but the SA hierarchy allows the use of the solver introduced in 4. Indeed, the constraints added at each level of the hierarchy can be seen as new subproblems for our solver. This would lead to minimal changes from one level of the hierarchy to the other, allowing to warm start the problem of a given level using the solution of the previous level for example.

This idea is backed by previous results such as [3] which shows that using heuristics to select a subset of the first level of the SA hierarchy for the MAP estimation LP of CRFs makes the LP relaxation tight, meaning that it is equivalent to the ILP. Recent improvement in machine learning could be used to build on top of such work and replace such hand designed heuristic with a learnt one, specialized for the instances of ILPs at hand. Such approach would give an automatic approach to either, solve the instances of ILPs that are easy in practice or, provide a good approximation for the ones that cannot be solved exactly.

Appendices

A

Papers Supplementary

Contents

A.1 Efficient Continuous Relaxation for Dense CRF

A.1.1 Filter-based Method Approximation

In this paper, the filter based method that we use for our experiments is the one by Adams et al. [21]. In this method, the original computation is approximated by a convolution in a higher dimensional space. The original points are associated to a set of vertices on which the convolution is performed. The considered vertices are the one from the permutohedral lattice. Krähenbühl and Koltun [5] provided an implementation of this method. In their implementation, they added a pixel-wise normalisation of the output of the permutohedral lattice and say that it performs well in practice.

We observe that for the variances considered in this paper and **without** using the normalisation by Krähenbühl and Koltun, the results given by the permutohedral lattice is a constant factor away from the value computed by brute force in most cases. As can be seen in Figure A.1, in the case where we compute $\sum_{a,b} K_{a,b}1$, the left graph, the ratio between the value obtained by brute force and the value obtained using the permutohedral lattice is 0.6 for large enough images. On the other hand, for a different value of the input points where we compute $\sum_{b>a} K_{a,b} - \sum_{b<a} K_{a,b}$, the right graph, we get a ratio of 0.48 between the two results. The case where we consider a variance of 50 is special. We know that the highest the variance value, the worst the approximation of the permutohedral is. If the experience on the full computation is conducted on an image of size 320×213, the ratio between the brute force approach and the permutohedral lattice is 0.633. At the same time is also worth noting that in all these results, if we consider the outputs as vectors, as is done when computing our gradients, the vectors given by the brute force and the ones given by the permutohedral lattice are collinear for all image size and all variances. We can thus expect that for other input values, the direction of gradient provided by the permutohedral lattice is correct, but the norm of this vector may be incorrect.

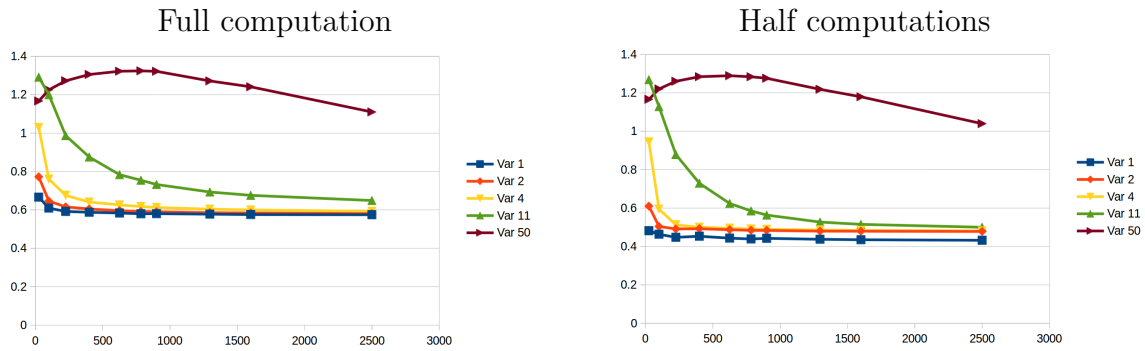


Figure A.1: Permutohedral lattice approximation. The vertical axis is the value computed in a brute force manner over the value computed by the permutohedral lattice. The horizontal axis is the number of pixels in the considered images. The graph on the left shows this ratio when comparing full permutohedral lattice computations with only ones as input. The graph on the right shows this ratio for the divide and conquer approach presented in the LP section. We can see that in both cases, for sufficiently large images, the permutohedral lattice computation is a constant factor away from the brute force value. This constant being a function of the input points values.

A.1.2 Optimal Step Size in the Frank-Wolfe Algorithm

Solving the convex relaxation of the QP is performed using the Franke-Wolfe algorithm [22]. The gradient is computed efficiently using the filter-based method [21]. An efficient method is available to compute the conditional gradient, based on the gradient. Once this conditional gradient is obtained, a step size needs to be determined to updates the value of the current parameters. We show that the optimal step size can be computed and that this does not introduce any additional call to the filter-based method.

The problem to solve is

$$\underset{\alpha\in[0,1]}{\operatorname{argmin}} S_{cvx}(\mathbf{y}+\alpha(\mathbf{s}-\mathbf{y})). \tag{A.1.2.1}$$

The definition of S_{cvx} is

$$S_{cvx}(\mathbf{y}) = (\boldsymbol{\phi}-\mathbf{d})^T\mathbf{y} + \mathbf{y}^T(\boldsymbol{\Psi}+\mathbf{D})\mathbf{y}. \tag{A.1.2.2}$$

Solving for the optimal value of α amounts to solving a second order polynomial:

$$\begin{aligned} S_{cvx}(\mathbf{y}+\alpha(\mathbf{s}-\mathbf{y}) &= (\boldsymbol{\phi}-\mathbf{d})^T(\mathbf{y}+\alpha(\mathbf{s}-\mathbf{y})) \\ &\quad + (\mathbf{y}+\alpha(\mathbf{s}-\mathbf{y}))^T(\boldsymbol{\Psi}+\mathbf{D})(\mathbf{y}+\alpha(\mathbf{s}-\mathbf{y})), \\ &= \alpha^2\left[(\mathbf{s}-\mathbf{y})^T(\boldsymbol{\Psi}+\mathbf{D})(\mathbf{s}-\mathbf{y})\right] \\ &\quad + \alpha\left[(\boldsymbol{\phi}-\mathbf{d})^T(\mathbf{s}-\mathbf{y}) + 2\mathbf{y}^T(\boldsymbol{\Psi}+\mathbf{D})(\mathbf{s}-\mathbf{y})\right] \\ &\quad + \quad\left[(\boldsymbol{\phi}-\mathbf{d})^T\mathbf{y} + \mathbf{y}^T(\boldsymbol{\Psi}+\mathbf{D})\mathbf{y}\right], \end{aligned} \tag{A.1.2.3}$$

whose optimal value is given by

$$\alpha^\star = -\frac{1}{2}\frac{(\boldsymbol{\phi}-\mathbf{d})^T(\mathbf{s}-\mathbf{y}) + 2\mathbf{y}^T(\boldsymbol{\Psi}+\mathbf{D})(\mathbf{s}-\mathbf{y})}{(\mathbf{s}-\mathbf{y})^T(\boldsymbol{\Psi}+\mathbf{D})(\mathbf{s}-\mathbf{y})} \tag{A.1.2.4}$$

The dot products are going to be linear in complexity and efficient. Using the filtering approach, the matrix-vector operation are also linear in complexity. In terms of run-time, they represent the costliest step so minimizing the number of times that we are going to perform them will gives us the best performance for our algorithm. We remind the reader that the expression of the gradient used at an iteration is:

$$\nabla S_{\text{cvx}}(\mathbf{y}) = (\boldsymbol{\phi} - \mathbf{d}) + 2(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y}. \tag{A.1.2.5}$$

so by keeping intermediary results of the gradient's computation, we don't need to compute $(\boldsymbol{\Psi}+\mathbf{D})\mathbf{y}$, having already performed this operation once. The other matrix-vector product that is necessary for obtaining the optimal step-size is $(\boldsymbol{\Psi} + \mathbf{D})\mathbf{s}$. During the first iteration, we will need to compute is using filter-based methods, which means using them twice in the same iteration. However, this computation can be reused. The update rules that we follow are:

$$\mathbf{y}^{t+1} = \mathbf{y}^t + \alpha(\mathbf{s} - \mathbf{y}^t). \tag{A.1.2.6}$$

At the following iteration, to obtain the gradient, we will need to compute:

$$\begin{aligned}(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y}^{t+1} &= (\boldsymbol{\Psi} + \mathbf{D})(\mathbf{y}^t + \alpha(\mathbf{s} - \mathbf{y}^t), \\ &= (1 - \alpha)(\boldsymbol{\Psi} + \mathbf{D})\mathbf{y}^t + \alpha(\boldsymbol{\Psi} + \mathbf{D})\mathbf{s}.\end{aligned} \tag{A.1.2.7}$$

All the matrix-vector product of this equation have already been computed. This means that no call to the filter-based method will be required.

At each iteration, we will only need to perform the expensive matrix-vector products on the conditional gradient. Using linearity and keeping track of our previous computations, we can then obtain all the other terms that we need.

A.1.3 Convex Problem in the Restricted DC Relaxation

Two difference-of-convex decompositions of the objective function are presented in the paper. The first one is based on diagonally dominant matrices to ensure convexity and would be applicable to any QP objective function. However, using this decomposition, the terms involving the pixel-compatibility function, and therefore requiring filter-based convolutions, need to be computed several times per CCCP iteration.

On the other hand, in the case of negative semi-definite compatibility functions, a decomposition suited to the structure of the problem is available. Using this

decomposition, similar to the one proposed by Krähenbühl [23], the convex problem to solve CCCP will be the following:

$$\begin{aligned} \min \quad & (\boldsymbol{\phi}^T - \mathbf{g}^T)\mathbf{y} - \mathbf{y}^T(\boldsymbol{\mu} \otimes \mathbf{I}_N)\mathbf{y}, \\ \text{s.t.} \quad & \mathbf{y} \in \mathcal{M}. \end{aligned} \tag{A.1.3.1}$$

The filter-method has been used to compute the gradient of the concave part $\mathbf{g}$. The Kronecker product with the identity matrix will make this problem completely de-correlated between pixels. This means that instead of solving one problem involving $N \times L$ variables, we will have to solve N problems of L variables, which is much faster. The problem to solve for each pixel are, using the a subscript to refer to the subset of the vector elements that correspond to the random variable a:

$$\begin{aligned} \min \quad & (\boldsymbol{\phi}_a^T - \mathbf{g}_a^T)\mathbf{y}_a - \mathbf{y}_a^T \boldsymbol{\mu} \mathbf{y}_a, \\ \text{s.t.} \quad & \mathbf{y}_a \geq 0 \\ & \mathbf{y}_a^T \mathbb{1} = 1. \end{aligned} \tag{A.1.3.2}$$

These problems can also be solved using the Frank-Wolfe algorithm, with efficient conditional gradient computation and optimal step size. The only difference is that in that case, no filter-based method will need to be used for computation. CCCP on this DC relaxation will therefore be much faster than on the generic case, an improvement gained at the cost of generality.

We also remark that the guarantees of CCCP to provide better results at each iteration does not require to solve the convex problem exactly. It is sufficient to obtain a value of the convex problem lower that the initial estimate. Therefore, the inference may eventually be sped-up by solving the convex problem approximately instead of reaching the optimal solution.

A.1.4 LP Objective Reformulation

This section presents the reformulation of the pairwise part of the LP objective. We first introduce the following equality:

$$\sum_a \sum_{b>a} K_{a,b} y_b(i) = \sum_a \sum_{b<a} K_{a,b} y_a(i), \tag{A.1.4.1}$$

It comes from the symmetry of $\mathbf{K}$.

Using the above formula, considering the reordering has already been done, we can rewrite the pairwise term of (18) as:

$$\begin{aligned}
&\sum_a \sum_{b\neq a} \sum_i K_{a,b} \frac{|y_a(i) - y_b(j)|}{2}, \\
=&\sum_i \sum_a \sum_{b>a} K_{a,b} \frac{y_a(i) - y_b(i)}{2} - \sum_i \sum_a \sum_{b<a} K_{a,b} \frac{y_a(i) - y_b(i)}{2}, \\
=&\sum_i \sum_a \sum_{b>a} K_{a,b} y_a(i) - \sum_i \sum_a \sum_{b<a} K_{a,b} y_a(i).
\end{aligned} \tag{A.1.4.2}$$

It is important to note that in these equations, the ordering between a and b used in the summations is dependent on the considered label i.

A.1.5 LP Divide and Conquer

We are going to present an algorithm to efficiently compute the following:

$$\forall k \quad \sum_{j>k} K_{k,j}, \tag{A.1.5.1}$$

for j and k being between 1 and N. For the sake of simplicity, we are going to consider N as being even. The odd case is very similar. Considering $h = N/2$, we can rewrite the original sum as:

$$\begin{aligned}
\sum_{j>k} K_{k,j} &= \begin{cases} \sum_{j>k} K_{k,j} & \text{if } k > h \\ \sum_{j>k} K_{k,j} & \text{if } k \leq h \end{cases} \\
&= \begin{cases} \sum_{j>k} K_{k,j} & \text{if } k > h \\ \sum_{\substack{j>k \\ j\leq h}} K_{k,j} + \sum_{\substack{j>k \\ j>h}} K_{k,j} & \text{if } k \leq h \end{cases} \\
&= \begin{cases} \underbrace{\sum_{j>k} K_{k,j}}_{A} & \text{if } k > h \\ \underbrace{\sum_{\substack{j>k \\ j\leq h}} K_{k,j}}_{B} + \underbrace{\sum_{j>h} K_{k,j}}_{C} & \text{if } k \leq h \end{cases}
\end{aligned} \tag{A.1.5.2}$$

We can see that both A and B corresponds to the cases where respectively $k, j > h$ and $k, j \leq h$. These two elements can be obtained by recursion using sub-matrices of $\mathbf{K}$ which have half the size of the current size of the problem. To compute the C part, we consider the following variable:

$$v_j = \begin{cases} 0 & \text{if } j \leq h \\ 1 & \text{if } j > h \end{cases} \tag{A.1.5.3}$$

We can now rewrite the C part as $\sum_j K_{k,j} v_j$. We can compute this sum efficiently for all k using the filter-based method. Since this term contribute to the original sum only when $k \leq h$, we will only consider a subset of the output from the filter based method.

So we have a recursive algorithm that will have a depth of $log(N)$ and for which all level takes $\mathcal{O}(N)$ to compute. We can use it to compute the requested sum $\forall k$ in $\mathcal{O}(Nlog(N))$.

A.1.6 LP Generalisation Beyond Potts Models

In this section, we consider the case where the label compatibility $\mu(x_a, x_b)$ is any semi-metric. We recall that $\mu(\cdot,\cdot)$ is a semi-metric if and only if $d(i,i) = 0, \forall i$ and $d(i,j) = d(j,i) > 0, \forall i \neq j$.

To solve this problem, we are going to reduce the semi-metric labelling problem to a *r-hierarchically well-separated tree* (r-HST) labelling problem that we can then reduce to a uniform labelling problem.

As described in [38], an r-HST metric [123] $d^t(\cdot,\cdot)$ is specified by a rooted tree whose edge lengths are non-negative and satisfy the following properties: (i) the edge lengths from any node to all of its children are the same; and (ii) the edge lengths along any path from the root to a leaf decrease by a factor of at least $r > 1$. Given such a tree, known as r-HST, the distance $d^t(i,j)$ is the sum of the edge lengths on the unique path between them

A.1.6.1 Approximate the Semi-Metric with r-HST Metric

Fakcharoenphol et al. [124] present an algorithm to get in polynomial time a mixture of r-HST that approximate any semi-metric using a fixed number of trees. This algorithm generate a collection $\mathcal{D} = d^t(\cdot,\cdot), t = 1, ..., n$ where each d^t is a r-HST metric.

Since this mixture of r-HST metric is an approximation to the original metric, we can approximate the original labelling problem by solving the labelling problem on each of these r-HST metrics and combining them with the method presented in [38]. We are now going to present an efficient algorithm to solve the problem on an r-HST. This algorithm can then be used to solve the problem in the semi-metric case.

A.1.6.2 Solve the r-HST Labelling Problem

We now consider a given r-HST metric d^t and we note $\mathcal{T}$ all the sub-trees corresponding to this metric and we will use T as one of these sub-trees. This problem has been formulated by Kleinberg and Tardos [16], but we are not using their method to solve it because the density of our CRF makes their method unfeasible. We are going to solve their original LP directly:

$$\begin{aligned} \min \quad & \sum_a \sum_i \phi_a(i) y_a(i) + \sum_{a,b \neq a} \sum_T K_{a,b} c_T \frac{|y_a(T) - y_b(T)|}{2} \\ \text{such that} \quad & \sum_i y_a(i) = 1 \quad \forall a \\ & y_a(T) = \sum_{i \in L(T)} y_a(i) \quad \forall a \forall T \\ & y_a(i) \in \{0,1\} \quad \forall a, i \end{aligned} \tag{A.1.6.1}$$

Where $L(T)$ is the set of all labels associated with a sub-tree T.

This problem is the same the Potts model case except for two points:

- In the pairwise term, the labels have been replaced by sub-trees. Since the assignment of the labels to the trees does not change, this won't prevent us from using the same method to compute the gradient.
- There is a factor c_T corresponding to the weights in the tree. This does not prevents us from computing this efficiently since we can move this out of the inner loop with the summation over the trees.

Using the same trick where we sort the $y_a(T)$ for all T, we can rewrite the pairwise part of the above problem and compute its sub-gradient. Using the fact that $\frac{\partial y_a(T)}{\partial y_c(k)}$ is 0 if $a \neq c$ or $k \notin L(T)$ and 1 otherwise and noting T_k all the sub-trees that contains k as one of their label.

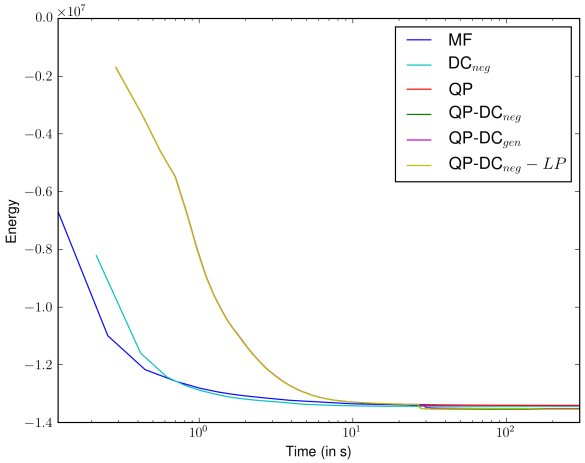
$$\begin{aligned} & \frac{\partial}{\partial y_{c,k}} \Big(\sum_T \sum_{a,b \neq a} K_{a,b} c_T \frac{|y_a(T) - y_b(T)|}{2} \Big) \\ =& \frac{\partial}{\partial y_{c,k}} \Big(\sum_T \sum_{a,b} K_{a,b} c_T \frac{y_a(T) - y_b(T)}{2} \Big) - 2 \sum_T \sum_{a,b<a} K_{a,b} c_T \frac{y_a(T) - y_b(T)}{2} \Big) \\ =& \sum_{T_k} \sum_b c_{T_k} \frac{K_{c,b}}{2} - \sum_{T_k} \sum_a c_{T_k} \frac{K_{a,c}}{2} - 2 \sum_{T_k} \sum_{b<c} c_{T_k} \frac{K_{c,b}}{2} + 2 \sum_{T_k} \sum_{a>c} c_{T_k} \frac{K_{a,c}}{2} \\ =& - \sum_{T_k} \sum_{a<c} c_{T_k} K_{a,c} + \sum_{T_k} \sum_{a>c} c_{T_k} K_{a,c} \\ =& \sum_{T_k} c_{T_k} \Big(\sum_{a>c} K_{a,c} - \sum_{a<c} K_{a,c} \Big) \end{aligned} \tag{A.1.6.2}$$

Since the sorting is done for each sub-tree T, the sums where we consider relative ordering of the indices cannot be switched and thus we cannot simplify this expression further.

We present a method to solve this problem in Algorithm 7. State 3 initialise the subgradient with the unaries. The loop at State 4 is used to compute the participation of each subtree to the total subgradient. To do so, we first precompute the $y_a(T)$ terms for all pixel in the loop starting at State 5. We then sort these $y_a(T)$ in State 8 and reorder the $\mathbf{K}$ matrix accordingly in State 9. Using this, we can use the divide and conquer approach from the Potts model section to compute the participation of this tree to the subgradient for each pixel using the formula from State 10. The vector $\mathbf{gt}$ is of size N and contains one value per pixel. We can now update the subgradient with the partial one on this tree for all labels associated with this tree. The notation $\mathbf{g}_{\cdot,\mathbf{k}}$ corresponds to a single row of the gradient matrix. This is done in the for loop starting at State 11. We can perform one step of subgradient descent in State 17. Finally we need to project the new point on the feasible set at State 17

Algorithm 7: r-HST labelling problem

```
1  Get y^0
2  while not converged do
3  |  Initialise the subgradient g = φ
4  |  for all subtree T do
5  |  |  for all pixel a do
6  |  |  |  y_a(T) = Σ_{i∈L(T)} y_a(i)
7  |  |  end
8  |  |  Sort y_a(T)
9  |  |  Reorder K
10 |  |  Gradient for this subtree gt_c = c_T(Σ_{a>c} K_{a,c} − Σ_{a<c} K_{a,c})
11 |  |  for all label k do
12 |  |  |  if k ∈ T then
13 |  |  |  |  Update g_{·,k} += gt
14 |  |  |  end
15 |  |  end
16 |  end
17 |  y^{t+1} = y^t − βg Project y^{t+1} such that it is a feasible point
18 end
```



Method	Final energy
MF	-1.341e+07
DC_{neg}	-1.344e+07
QP	-1.341e+07
QP-DC_{neg}	**-1.354e+07**
QP-DC_{gen}	-1.352e+07
QP-DC_{neg}-LP	**-1.354e+07**

Figure A.2: Evolution of achieved energies as a function of time on a stereo matching problem (Venus Image).

A.1.7 Model Used in the Experiments Section

All the results that we presented in the paper are valid for pixel compatibility functions composed of a mixture of Gaussian kernels:

$$\sum_m w^{(m)} k(\mathbf{f}_a^{(m)}, \mathbf{f}_b^{(m)}). \tag{A.1.7.1}$$

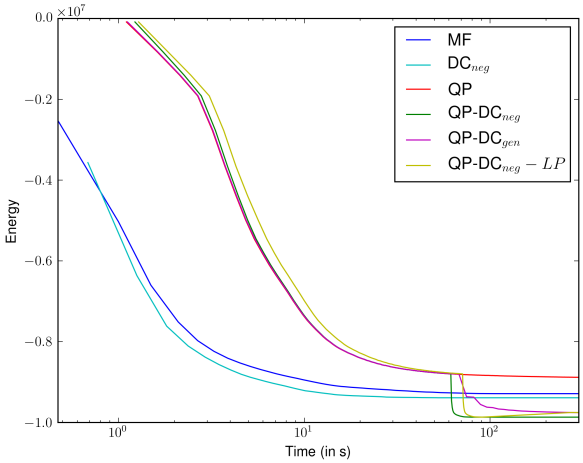
In practice, for our experiments, we use the same form as Krähenbühl and Koltun [5]. It is a mixture of two gaussian kernels defined using the position vectors $\mathbf{p}_a$ and $\mathbf{p}_b$, and the colour vectors $\mathbf{I}_a$ and $\mathbf{I}_b$ associated with each pixel a and b. The complete formula is the following:

$$K_{a,b} = w^{(1)} \exp\left(-\frac{|\mathbf{p}_a - \mathbf{p}_b|^2}{\sigma_1}\right) + w^{(2)} \exp\left(-\frac{|\mathbf{p}_a - \mathbf{p}_b|^2}{\sigma_{2,\text{spc}}} - \frac{|\mathbf{I_a} - \mathbf{I_b}|^2}{\sigma_{2,\text{col}}}\right). \tag{A.1.7.2}$$

We note that this pixel compatibility function contains 5 learnable parameters $w^{(1)}, \sigma_1, w^{(2)}, \sigma_{2,\text{spc}}, \sigma_{2,\text{col}}$.

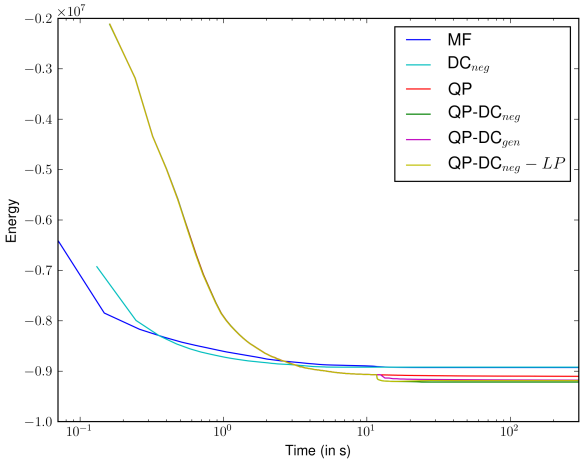
A.1.8 More Results on Stereo Matching

In the following Figures A.5, A.7 and A.6, we observe that the continuous relaxations give consistently better results. We can also note that even though it runs only for a few iterations, the **LP** improves the visual quality of the solution.



Method	Final energy
MF	-9.286e+06
DC$_{neg}$	-9.388e+06
QP	-8.881e+06
QP-DC$_{neg}$	**-9.868e+06**
QP-DC$_{gen}$	-9.757e+06
QP-DC$_{neg}$-LP	-9.758e+06

Figure A.3: Evolution of achieved energies as a function of time on a stereo matching problem (Cones Image). Note that the LP in that case is not improving the results.



Method	Final energy
MF	-8.927e+06
DC$_{neg}$	-8.920e+06
QP	-9.101e+06
QP-DC$_{neg}$	**-9.215e+06**
QP-DC$_{gen}$	-9.177e+06
QP-DC$_{neg}$-LP	-9.186e+06

Figure A.4: Evolution of achieved energies as a function of time on a stereo matching problem (Tsukuba Image). Note that the LP in that case is not improving the results.

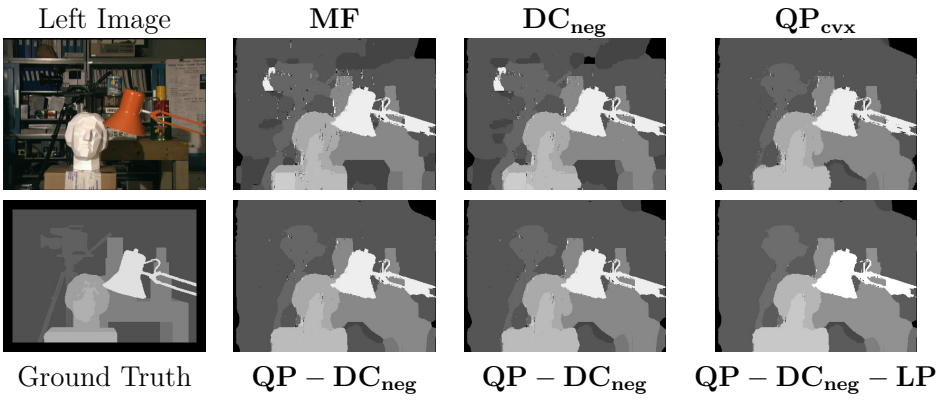


Figure A.5: Stereo matching results on the Tsukuba image and corresponding timings. We see that the LP method allows to improve the smoothness from its initialisation $\mathbf{DC_{neg}}$. For this set of parameters, the mean-field methods performs poorly.

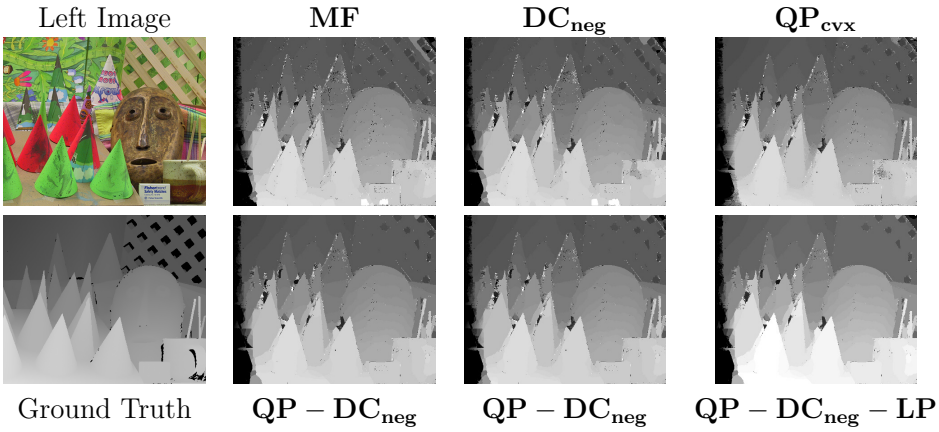


Figure A.6: Stereo matching results on the Cones image and corresponding timings. Here again, we can see that the **MF** solution is significantly better than the **MF5** solution. The continuous relaxations improve even further by reducing the number of artifacts.

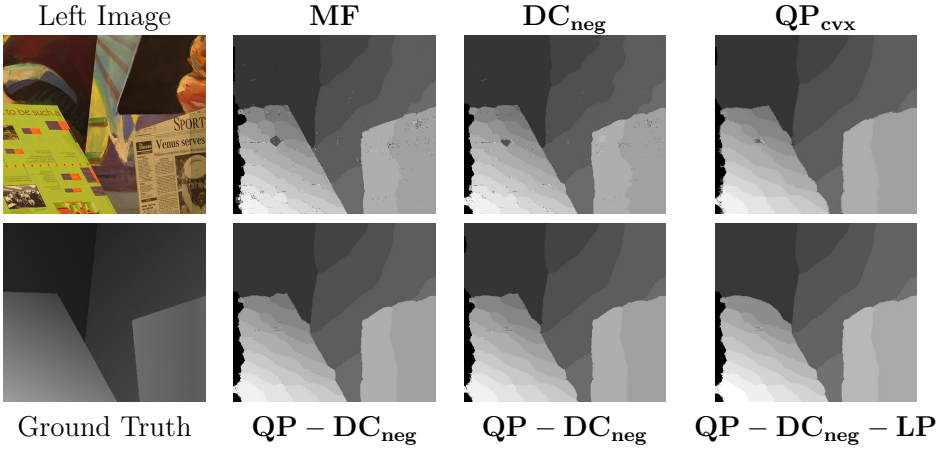


Figure A.7: Stereo matching results on the Venus image. Note that the smoothness of the reconstructions improves with methods reaching lower energies. The LP result does not artefacts anymore, only the non contiguous borders due to the Potts model.

A.1.9 More Results on Segmentation

Additional results for parameters cross-validated for **MF** are presented in Table A.1.

We see that in this case where the parameters are tuned for **MF**, the continuous relaxations still reach lower energy than the mean-field approaches on average. Furthermore, we observe that in almost all images, the energy is strictly lower than the one provided by the mean-field methods. However, with the parameters that were tuned for **MF** using cross-validation, we note that the segmentation performance is poor compared to mean-field approaches, and that the improved energy minimization does not translate to better segmentation.

	Unary	**MF5**	**MF**	$\mathbf{QP_{cvx}}$	$\mathbf{DC_{gen}}$	$\mathbf{DC_{neg}}$	**LP**	**Avg. E**	**Acc**	**IoU**
Unary	-	0	0	0	0	0	0	0	79.04	27.43
MF5	99	-	0	1	0	1	1	-8.37e5	80.42	28.66
MF	99	93	-	4	2	3	3	-1.19e6	**80.95**	**28.86**
$\mathbf{QP_{cvx}}$	99	93	86	-	0	4	2	-1.66e6	77.75	14.94
$\mathbf{DC_{gen}}$	99	94	88	32	-	29	29	**-1.68e6**	77.76	14.96
$\mathbf{DC_{neg}}$	99	93	87	27	2	-	12	-1.67e6	77.76	14.91
LP	99	93	87	29	2	17	-	-1.67e6	77.77	14.93

Table A.1: Percentage of images obtaining strictly lower energy values. Average energy results over the test set and Segmentation performance. Higher percentage is better and lower energy is better. Higher accuracy and IoU are better. Continuous relaxations dominate mean-field approaches on almost all images and improve significantly more compared to the Unary baseline. Parameters tuned for **MF**.

A.2 Efficient Linear Programming for Dense CRFs

A.2.1 Proximal Minimization for LP Relaxation

In this section, we give the detailed derivation of our proximal minimization algorithm for the LP relaxation.

A.2.1.1 Dual Formulation

Let us first restate the definition of the matrices A and B, and analyze their properties. This would be useful to derive the dual of the proximal problem (2.2.4.2a).

Definition A.2.1. *Let $A \in \mathbb{R}^{\mathrm{nm}\times\mathrm{p}}$ and $B \in \mathbb{R}^{\mathrm{nm}\times\mathrm{n}}$ be two matrices such that*

$$(A\boldsymbol{\alpha})_{a:i} = -\sum_{b\neq a}\left(\alpha^1_{ab:i} - \alpha^2_{ab:i} + \alpha^2_{ba:i} - \alpha^1_{ba:i}\right) , \tag{A.2.1.1}$$

$$(B\boldsymbol{\beta})_{a:i} = \beta_a \ .$$

Proposition A.2.1. *Let* $\mathbf{x} \in \mathbb{R}^{\mathrm{nm}}$*. Then, for all* $a \neq b$ *and* $i \in \mathcal{L}$*,*

$$\begin{aligned} \left(A^T\mathbf{x}\right)_{ab:i^1} &= x_{b:i} - x_{a:i} \ , \\ \left(A^T\mathbf{x}\right)_{ab:i^2} &= x_{a:i} - x_{b:i} \ . \end{aligned} \tag{A.2.1.2}$$

Here, the index $ab:i^1$ *denotes the element corresponding to* $\alpha^1_{ab:i}$*.*

Proof. This can be easily proved by inspecting the matrix A. □

Proposition A.2.2. *The matrix* $B \in \mathbb{R}^{\mathrm{nm}\times\mathrm{n}}$ *defined in Eq.* (A.2.1.1) *satisfies the following properties:*

1. *Let* $\mathbf{x} \in \mathbb{R}^{\mathrm{nm}}$*. Then,* $\left(B^T\mathbf{x}\right)_a = \sum_{i\in\mathcal{L}} x_{a:i}$ *for all* $a \in \{1 \dots n\}$*.*
2. $B^TB = mI$*, where* $I \in \mathbb{R}^{\mathrm{n}\times\mathrm{n}}$ *is the identity matrix.*
3. BB^T *is a block diagonal matrix, with each block* $\left(BB^T\right)_a = \mathbf{1}$ *for all* $a \in \{1 \dots n\}$*, where* $\mathbf{1} \in \mathbb{R}^{\mathrm{m}\times\mathrm{m}}$ *is the matrix of all ones.*

Proof. Note that, from Eq. (A.2.1.1), the matrix B simply repeats the elements β_a for m times. In particular, for $m = 3$, the matrix B has the following form:

$$B = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ 1 & \vdots & & & & \vdots \\ 1 & 0 & \cdots & \cdots & \cdots & \vdots \\ 0 & 1 & & & & \vdots \\ \vdots & 1 & & & & \vdots \\ \vdots & 1 & & & & \vdots \\ \vdots & 0 & & & & \vdots \\ \vdots & \vdots & & & & 0 \\ \vdots & \vdots & & & & 1 \\ \vdots & \vdots & & & & 1 \\ 0 & \cdots & \cdots & \cdots & 0 & 1 \end{bmatrix} . \tag{A.2.1.3}$$

Therefore, multiplication by B^T amounts to summing over the labels. From this, the other properties can be proved easily. □

We now derive the Lagrange dual of (2.2.4.2a).

Proposition A.2.3. *Given matrices* $A \in \mathbb{R}^{\mathrm{nm}\times\mathrm{p}}$ *and* $B \in \mathbb{R}^{\mathrm{nm}\times\mathrm{n}}$ *and dual variables* $(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma})$*.*

1. *The Lagrange dual of* (2.2.4.2a) *takes the following form:*

$$\min_{\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}} g(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}) = \frac{\lambda}{2}\|A\boldsymbol{\alpha}+B\boldsymbol{\beta}+\boldsymbol{\gamma}-\boldsymbol{\phi}\|^2 + \left\langle A\boldsymbol{\alpha}+B\boldsymbol{\beta}+\boldsymbol{\gamma}-\boldsymbol{\phi}, \mathbf{y}^k\right\rangle - \langle \mathbf{1},\boldsymbol{\beta}\rangle \ , \tag{A.2.1.4}$$

$$\begin{aligned} \textit{s.t.} \qquad \gamma_{a:i} \quad &\geq 0 \quad \forall\, a \in \{1\dots n\} \quad \forall\, i \in \mathcal{L} \ , \\ \boldsymbol{\alpha} \in \mathcal{C} \quad &= \left\{ \boldsymbol{\alpha} \;\middle|\; \begin{array}{l} \alpha^1_{ab:i} + \alpha^2_{ab:i} = \frac{K_{ab}}{2},\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \\ \alpha^1_{ab:i}, \alpha^2_{ab:i} \geq 0,\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \end{array} \right\} . \end{aligned}$$

2. *The primal variables* $\mathbf{y}$ *satisfy*

$$\mathbf{y} = \lambda\left(A\boldsymbol{\alpha}+B\boldsymbol{\beta}+\boldsymbol{\gamma}-\boldsymbol{\phi}\right)+\mathbf{y}^k \ . \tag{A.2.1.5}$$

Proof. The Lagrangian associated with the primal problem (2.2.4.2a) can be written as [125]:

$$\begin{aligned} \max_{\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}} \min_{\mathbf{y},\mathbf{z}} L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{y},\mathbf{z}) &= \sum_a\sum_i \phi_{a:i}\, y_{a:i} + \sum_{a,b\neq a}\sum_i \frac{K_{ab}}{2} z_{ab:i} + \frac{1}{2\lambda}\sum_a\sum_i \left(y_{a:i}-y^k_{a:i}\right)^2 \\ &\quad + \sum_{a,b\neq a}\sum_i \alpha^1_{ab:i}\left(y_{a:i}-y_{b:i}-z_{ab:i}\right) + \sum_{a,b\neq a}\sum_i \alpha^2_{ab:i}\left(y_{b:i}-y_{a:i}-z_{ab:i}\right) \\ &\quad + \sum_a \beta_a\left(1-\sum_i y_{a:i}\right) - \sum_a\sum_i \gamma_{a:i}\, y_{a:i} \ , \end{aligned} \tag{A.2.1.6}$$

$$\begin{aligned} \text{s.t.} \qquad \alpha^1_{ab:i}, \alpha^2_{ab:i} \quad &\geq 0 \quad \forall\, a \neq b \quad \forall\, i \in \mathcal{L} \ , \\ \gamma_{a:i} \quad &\geq 0 \quad \forall\, a \in \{1\dots n\} \quad \forall\, i \in \mathcal{L} \ . \end{aligned}$$

Note that the dual problem is obtained by minimizing the Lagrangian over the primal variables $(\mathbf{y},\mathbf{z})$. With respect to $\mathbf{z}$, the Lagrangian is linear and when $\nabla_{\mathbf{z}} L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{y},\mathbf{z}) \neq 0$, the minimization in $\mathbf{z}$ yields $-\infty$. This situation is not useful as the dual function is unbounded. Therefore we restrict ourselves to the case where $\nabla_{\mathbf{z}} L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{y},\mathbf{z}) = 0$. By differentiating with respect to $\mathbf{z}$ and setting the derivatives to zero, we obtain

$$\alpha^1_{ab:i} + \alpha^2_{ab:i} = \frac{K_{ab}}{2} \quad \forall\, a \neq b \quad \forall\, i \in \mathcal{L} \ . \tag{A.2.1.7}$$

The minimum of the Lagrangian with respect to $\mathbf{y}$ is attained when $\nabla_{\mathbf{y}} L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{y},\mathbf{z}) = 0$. Before differentiating with respect to $\mathbf{y}$, let us rewrite the Lagrangian using Eq. (A.2.1.7) and reorder the terms:

$$\begin{aligned} L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma},\mathbf{y},\mathbf{z}) &= \sum_a\sum_i (\phi_{a:i}-\beta_a-\gamma_{a:i})\, y_{a:i} + \frac{1}{2\lambda}\sum_a\sum_i \left(y_{a:i}-y^k_{a:i}\right)^2 + \sum_{a,b\neq a}\sum_i \left(\alpha^1_{ab:i}-\alpha^2_{ab:i}\right) y_{a:i} \\ &\quad + \sum_{a,b\neq a}\sum_i \left(\alpha^2_{ba:i}-\alpha^1_{ba:i}\right) y_{a:i} + \sum_a \beta_a \ . \end{aligned} \tag{A.2.1.8}$$

Now, by differentiating with respect to $\mathbf{y}$ and setting the derivatives to zero, we get

$$\frac{1}{\lambda}\left(y_{a:i} - y^k_{a:i}\right) = -\sum_{b\neq a}\left(\alpha^1_{ab:i} - \alpha^2_{ab:i} + \alpha^2_{ba:i} - \alpha^1_{ba:i}\right) + \beta_a + \gamma_{a:i} - \phi_{a:i} \quad \forall\, a \in \{1\dots n\} \quad \forall\, i \in \mathcal{L}\ . \tag{A.2.1.9}$$

Using Eq. (A.2.1.1), the above equation can be written in vector form as

$$\frac{1}{\lambda}\left(\mathbf{y} - \mathbf{y}^k\right) = A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}\ . \tag{A.2.1.10}$$

This proves Eq. (A.2.1.5). Now, using Eqs. (A.2.1.7) and (A.2.1.10), the dual problem can be written as

$$\min_{\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}} g(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\gamma}) = \frac{\lambda}{2}\|A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}\|^2 + \left\langle A\boldsymbol{\alpha} + B\boldsymbol{\beta} + \boldsymbol{\gamma} - \boldsymbol{\phi}, \mathbf{y}^k\right\rangle - \langle \mathbf{1}, \boldsymbol{\beta}\rangle\ , \tag{A.2.1.11}$$

$$\text{s.t.} \qquad \gamma_{a:i} \geq 0 \quad \forall\, a \in \{1\dots n\} \quad \forall\, i \in \mathcal{L}\ ,$$

$$\boldsymbol{\alpha} \in \mathcal{C} = \left\{ \boldsymbol{\alpha} \;\middle|\; \begin{array}{l} \alpha^1_{ab:i} + \alpha^2_{ab:i} = \frac{K_{ab}}{2},\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \\ \alpha^1_{ab:i}, \alpha^2_{ab:i} \geq 0,\ \forall\, a \neq b,\ \forall\, i \in \mathcal{L} \end{array} \right\}\ .$$

Here, $\mathbf{1}$ denotes the vector of all ones of appropriate dimension. Note that we converted our problem to a minimization one by changing the sign of all the terms. This proves Eq. (A.2.1.4). □

A.2.1.2 Optimizing Over $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$

In this section, for a fixed value of $\boldsymbol{\alpha}^t$, we optimize over $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

Proposition A.2.4. *If $\nabla_\beta g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}, \boldsymbol{\gamma}) = 0$, then $\boldsymbol{\beta}$ satisfy*

$$\boldsymbol{\beta} = B^T\left(A\boldsymbol{\alpha}^t + \boldsymbol{\gamma} - \boldsymbol{\phi}\right)/m\ . \tag{A.2.1.12}$$

Proof. By differentiating the dual objective g with respect to $\boldsymbol{\beta}$ and setting the derivatives to zero, we obtain the above equation. Note that, from Proposition A.2.2, $B^T\mathbf{y}^k = \mathbf{1}$ since $\mathbf{y}^k \in \mathcal{M}$ (defined in Eq. (2.2.3.4)), and $B^TB = mI$. Both these identities are used to simplify the above equation. □

Let us now define a matrix D and analyze its properties. This will be useful to simplify the optimization over $\boldsymbol{\gamma}$.

Definition A.2.2. *Let $D \in \mathbb{R}^{\mathrm{nm}\times\mathrm{nm}}$ be a matrix that satisfy*

$$D = I - \frac{BB^T}{m}\ , \tag{A.2.1.13}$$

where B is defined in Eq. (A.2.1.1).

Proposition A.2.5. *The matrix D satisfies the following properties:*

1. *D is block diagonal, with each block matrix $D_a = I - \mathbf{1}/m$, where $I \in \mathbb{R}^{\mathrm{m}\times\mathrm{m}}$ is the identity matrix and $\mathbf{1} \in \mathbb{R}^{\mathrm{m}\times\mathrm{m}}$ is the matrix of all ones.*

2. *$D^T D = D$.*

Proof. From Proposition A.2.2, the matrix BB^T is block diagonal with each block $\left(BB^T\right)_a = \mathbf{1}$. Therefore D is block diagonal with each block matrix $D_a = I - \mathbf{1}/m$. Note that the block matrices D_a are identical. The second property can be proved using simple matrix algebra. □

Now we turn to the optimization over $\boldsymbol{\gamma}$.

Proposition A.2.6. *The optimization over $\boldsymbol{\gamma}$ decomposes over pixels where for a pixel a, this QP has the form*

$$\begin{aligned} \min_{\boldsymbol{\gamma}_a} \quad & \frac{1}{2}\boldsymbol{\gamma}_a^T Q \boldsymbol{\gamma}_a + \left\langle \boldsymbol{\gamma}_a, Q\left((A\boldsymbol{\alpha}^t)_a - \boldsymbol{\phi}_a\right) + \mathbf{y}_a^k \right\rangle \ , \\ s.t. \quad & \boldsymbol{\gamma}_a \geq \mathbf{0} \ . \end{aligned} \tag{A.2.1.14}$$

Here, $\boldsymbol{\gamma}_a$ denotes the vector $\{\gamma_{a:i} \mid i \in \mathcal{L}\}$ and $Q = \lambda\,(I - \mathbf{1}/m) \in \mathbb{R}^{\mathrm{m}\times\mathrm{m}}$, with I the identity matrix and $\mathbf{1}$ the matrix of all ones.

Proof. By substituting $\boldsymbol{\beta}$ in the dual problem (A.2.1.4) with Eq. (A.2.1.12), the optimization problem over $\boldsymbol{\gamma}$ takes the following form:

$$\min_{\boldsymbol{\gamma}} \ g(\boldsymbol{\alpha}^t, \boldsymbol{\gamma}) = \frac{\lambda}{2}\|D(A\boldsymbol{\alpha}^t + \boldsymbol{\gamma} - \boldsymbol{\phi})\|^2 + \left\langle D(A\boldsymbol{\alpha}^t + \boldsymbol{\gamma} - \boldsymbol{\phi}), \mathbf{y}^k \right\rangle + \frac{1}{m}\left\langle \mathbf{1}, A\boldsymbol{\alpha}^t + \boldsymbol{\gamma} - \boldsymbol{\phi} \right\rangle \ , \tag{A.2.1.15}$$

$$\text{s.t.} \qquad \boldsymbol{\gamma} \quad \geq \mathbf{0} \ ,$$

where $D = I - \frac{BB^T}{m}$.

Note that, since $\mathbf{y}^k \in \mathcal{M}$, from Proposition A.2.2, $B^T\mathbf{y}^k = \mathbf{1}$. Using this fact, the identity $D^T D = D$, and by removing the constant terms, the optimization problem over $\boldsymbol{\gamma}$ can be simplified:

$$\begin{aligned} \min_{\boldsymbol{\gamma}} \ & g(\boldsymbol{\alpha}^t, \boldsymbol{\gamma}) = \frac{\lambda}{2}\boldsymbol{\gamma}^T D \boldsymbol{\gamma} + \langle \boldsymbol{\gamma}, \lambda D(A\boldsymbol{\alpha}^t - \boldsymbol{\phi}) + \mathbf{y}^k \rangle \ , \\ \text{s.t.} \quad & \boldsymbol{\gamma} \quad \geq \mathbf{0} \ . \end{aligned} \tag{A.2.1.16}$$

Furthermore, since D is block diagonal from Proposition A.2.5, we obtain

$$\min_{\boldsymbol{\gamma}\geq\mathbf{0}} \ g(\boldsymbol{\alpha}^t, \boldsymbol{\gamma}) = \sum_a \min_{\boldsymbol{\gamma}_a \geq \mathbf{0}} \ \frac{\lambda}{2}\boldsymbol{\gamma}_a^T D_a \boldsymbol{\gamma}_a + \langle \boldsymbol{\gamma}_a, \lambda D_a\left((A\boldsymbol{\alpha}^t)_a - \boldsymbol{\phi}_a\right) + \mathbf{y}_a^k \rangle \ , \tag{A.2.1.17}$$

where the notation $\boldsymbol{\gamma}_a$ denotes the vector $\{\gamma_{a:i} \mid i \in \mathcal{L}\}$. By substituting $Q = \lambda D_a$, the QP associated with each pixel a can be written as

$$\min_{\boldsymbol{\gamma}_a \geq \mathbf{0}} \quad \frac{1}{2}\boldsymbol{\gamma}_a^T Q \boldsymbol{\gamma}_a + \left\langle \boldsymbol{\gamma}_a, Q\left((A\boldsymbol{\alpha}^t)_a - \boldsymbol{\phi}_a\right) + \mathbf{y}_a^k \right\rangle \ . \tag{A.2.1.18}$$

□

Each of these m dimensional quadratic programs (QP) are optimized using the iterative algorithm of [47]. Before we give the update equation, let us first write our problem in the form used in [47]. For a given $a \in \{1 \dots n\}$, this yields

$$\min_{\boldsymbol{\gamma}_a \geq \mathbf{0}} \frac{1}{2}\boldsymbol{\gamma}_a^T Q \boldsymbol{\gamma}_a - \langle \boldsymbol{\gamma}_a, \mathbf{h}_a \rangle \ , \tag{A.2.1.19}$$

where

$$\begin{aligned} Q &= \lambda\left(I - \frac{\mathbf{1}}{m}\right) \ , \\ \mathbf{h}_a &= -Q\left((A\boldsymbol{\alpha}^t)_a - \boldsymbol{\phi}_a\right) - \mathbf{y}_a^k \ . \end{aligned} \tag{A.2.1.20}$$

Hence, at each iteration, the element-wise update equation has the following form:

$$\gamma_{a:i} = \gamma_{a:i}\left[\frac{2\,(Q^-\boldsymbol{\gamma}_a)_i + h^+_{a:i} + c}{(|Q|\boldsymbol{\gamma}_a)_i + h^-_{a:i} + c}\right] \ , \tag{A.2.1.21}$$

where $Q^- = \max(-Q, 0)$, $|Q| = \mathrm{abs}(Q)$, $h^+_{a:i} = \max(h_{a:i}, 0)$ and $h^-_{a:i} = \max(-h_{a:i}, 0)$ and $0 < c \ll 1$. These max and abs operations are element-wise. We refer the interested reader to [47] for more detail on this update rule.

Note that, even though the matrix Q has m^2 elements, the multiplication by Q can be performed in $\mathcal{O}(m)$. In particular, the multiplication by Q can be decoupled into a multiplication by the identity matrix and a matrix of all ones, both of which can be performed in linear time. Similar observations can be made for the matrices Q^- and $|Q|$. Hence, the time complexity of the above update is $\mathcal{O}(m)$. Once the optimal $\boldsymbol{\gamma}$ for a given $\boldsymbol{\alpha}^t$ is computed, the corresponding optimal $\boldsymbol{\beta}$ is given by Eq. (A.2.1.12).

A.2.1.3 Conditional Gradient Computation

Proposition A.2.7. *The conditional gradient* $\mathbf{s}$ *satisfy*

$$(A\mathbf{s})_{a:i} = -\sum_b \left(K_{ab}\mathbf{1}[\tilde{y}^t_{a:i} \geq \tilde{y}^t_{b:i}] - K_{ab}\mathbf{1}[\tilde{y}^t_{a:i} \leq \tilde{y}^t_{b:i}]\right) \ , \tag{A.2.1.22}$$

where $\tilde{\mathbf{y}}^t = \lambda\,(A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}) + \mathbf{y}^k$ *using Eq.* (A.2.1.5).

Proof. The conditional gradient with respect to $\boldsymbol{\alpha}$ is obtained by solving the following linearization problem:

$$\mathbf{s} = \underset{\hat{\mathbf{s}} \in \mathcal{C}}{\operatorname{argmin}} \left\langle \hat{\mathbf{s}}, \nabla_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) \right\rangle \ , \tag{A.2.1.23}$$

where

$$\nabla_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) = A^T \tilde{\mathbf{y}}^t \ , \tag{A.2.1.24}$$

with $\tilde{\mathbf{y}}^t = \lambda \left(A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi} \right) + \mathbf{y}^k$ using Eq. (A.2.1.5).

Note that the feasible set $\mathcal{C}$ is separable, i.e., it can be written as $\mathcal{C} = \prod_{a,\, b \neq a,\, i \in \mathcal{L}} \mathcal{C}_{ab:i}$, with
$\mathcal{C}_{ab:i} = \{(\alpha^1_{ab:i}, \alpha^2_{ab:i}) \mid \alpha^1_{ab:i} + \alpha^2_{ab:i} = K_{ab}/2, \alpha^1_{ab:i}, \alpha^2_{ab:i} \geq 0\}$. Therefore, the conditional gradient can be computed separately, corresponding to each set $\mathcal{C}_{ab:i}$. This yields

$$\begin{aligned} \min_{\hat{s}^1_{ab:i}, \hat{s}^2_{ab:i}} \quad & \hat{s}^1_{ab:i} \nabla_{\alpha^1_{ab:i}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) + \hat{s}^2_{ab:i} \nabla_{\alpha^2_{ab:i}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) \ , \\ \text{s.t.} \quad & \hat{s}^1_{ab:i} + \hat{s}^2_{ab:i} = K_{ab}/2 \ , \\ & \hat{s}^1_{ab:i}, \hat{s}^2_{ab:i} \geq 0 \ , \end{aligned} \tag{A.2.1.25}$$

where, using Proposition A.2.1, the gradients can be written as:

$$\begin{aligned} \nabla_{\alpha^1_{ab:i}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) &= \tilde{y}^t_{b:i} - \tilde{y}^t_{a:i} \ , \\ \nabla_{\alpha^2_{ab:i}} g(\boldsymbol{\alpha}^t, \boldsymbol{\beta}^t, \boldsymbol{\gamma}^t) &= \tilde{y}^t_{a:i} - \tilde{y}^t_{b:i} \ . \end{aligned} \tag{A.2.1.26}$$

Hence, the minimum is attained at:

$$\begin{aligned} s^1_{ab:i} &= \begin{cases} K_{ab}/2 & \text{if } \tilde{y}^t_{a:i} \geq \tilde{y}^t_{b:i} \\ 0 & \text{otherwise} \ , \end{cases} \\ s^2_{ab:i} &= \begin{cases} K_{ab}/2 & \text{if } \tilde{y}^t_{a:i} \leq \tilde{y}^t_{b:i} \\ 0 & \text{otherwise} \ . \end{cases} \end{aligned} \tag{A.2.1.27}$$

Now, from Eq. (A.2.1.1), $A\mathbf{s}$ takes the following form:

$$(A\mathbf{s})_{a:i} = -\sum_{b \neq a} \left(\frac{K_{ab}}{2} \mathbf{1}[\tilde{y}^t_{a:i} \geq \tilde{y}^t_{b:i}] - \frac{K_{ab}}{2} \mathbf{1}[\tilde{y}^t_{a:i} \leq \tilde{y}^t_{b:i}] + \frac{K_{ba}}{2} \mathbf{1}[\tilde{y}^t_{b:i} \leq \tilde{y}^t_{a:i}] - \frac{K_{ba}}{2} \mathbf{1}[\tilde{y}^t_{b:i} \geq \tilde{y}^t_{a:i}] \right) \ , \tag{A.2.1.28}$$

$$= -\sum_{b} \left(K_{ab} \mathbf{1}[\tilde{y}^t_{a:i} \geq \tilde{y}^t_{b:i}] - K_{ab} \mathbf{1}[\tilde{y}^t_{a:i} \leq \tilde{y}^t_{b:i}] \right) \ .$$

Here, we used the symmetry of the kernel matrix K to obtain this result. Note that the second equation is a summation over $b \in \{1 \dots n\}$. This is true due to the identity $K_{aa} \mathbf{1}[\tilde{y}^t_{a:i} \geq \tilde{y}^t_{a:i}] - K_{aa} \mathbf{1}[\tilde{y}^t_{a:i} \leq \tilde{y}^t_{a:i}] = 0$ when $b = a$. □

A.2.1.4 Optimal Step Size

Proposition A.2.8. *The optimal step size δ satisfy*

$$\delta = P_{[0,1]}\left(\frac{\langle A\boldsymbol{\alpha}^t - A\mathbf{s}^t, \tilde{\mathbf{y}}^t\rangle}{\lambda\|A\boldsymbol{\alpha}^t - A\mathbf{s}^t\|^2}\right) \ . \tag{A.2.1.29}$$

Here, $P_{[0,1]}$ denotes the projection to the interval $[0,1]$, that is, clipping the value to lie in $[0,1]$.

Proof. The optimal step size δ gives the maximum decrease in the objective function g given the descent direction $\mathbf{s}^t$. This can be formulated as the following optimization problem:

$$\min_{\delta} \quad \frac{\lambda}{2}\left\|A\boldsymbol{\alpha}^t + \delta\left(A\mathbf{s}^t - A\boldsymbol{\alpha}^t\right) + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}\right\|^2 + \left\langle A\boldsymbol{\alpha}^t + \delta\left(A\mathbf{s}^t - A\boldsymbol{\alpha}^t\right) + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}), \mathbf{y}^k\right\rangle - \langle \mathbf{1} \tag{A.2.1.30}$$

$$\text{s.t.} \quad \delta \in [0,1] \ .$$

Note that the above function is optimized over the scalar variable δ and the minimum is attained when the derivative is zero. Hence, setting the derivative to zero, we have

$$0 = \lambda\left\langle \delta\left(A\mathbf{s}^t - A\boldsymbol{\alpha}^t\right) + A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}, A\mathbf{s}^t - A\boldsymbol{\alpha}^t\right\rangle + \left\langle \mathbf{y}^k, A\mathbf{s}^t - A\boldsymbol{\alpha}^t\right\rangle \ , \tag{A.2.1.31}$$

$$\delta = \frac{\langle A\boldsymbol{\alpha}^t - A\mathbf{s}^t, \lambda\left(A\boldsymbol{\alpha}^t + B\boldsymbol{\beta}^t + \boldsymbol{\gamma}^t - \boldsymbol{\phi}\right) + \mathbf{y}^k\rangle}{\lambda\|A\boldsymbol{\alpha}^t - A\mathbf{s}^t\|^2} \ ,$$

$$\delta = \frac{\langle A\boldsymbol{\alpha}^t - A\mathbf{s}^t, \tilde{\mathbf{y}}^t\rangle}{\lambda\|A\boldsymbol{\alpha}^t - A\mathbf{s}^t\|^2} \ .$$

In fact, if the optimal δ is out of the interval $[0,1]$, the value is simply truncated to be in $[0,1]$. □

A.2.2 Fast Conditional Gradient Computation

In this section, we give the technical details of the original filtering algorithm and then our modified filtering algorithm. To this end, we consider the following computation

$$\forall\, a \in \{1 \ldots n\}, \quad v'_a = \sum_b k(\mathbf{f}_a, \mathbf{f}_b)\, v_b\, \mathbf{1}[y_a \geq y_b] \ , \tag{A.2.2.1}$$

with $y_a, y_b \in [0,1]$ for all $a, b \in \{1 \ldots n\}$. Note that the above equation is the same as Eq. (14), except for the multiplication by the scalar v_b. In Section 4, the value v_b was assumed to be 1, but here we consider the general case where $v_b \in \mathbb{R}$.

A.2.2.1 Original Filtering Algorithm

Let us first introduce some notations below. We denote the set of lattice points of the original permutohedral lattice with $\mathcal{P}$ and the neighbouring feature points of lattice point l by $N(l)$. This neighbourhood is shown in Fig. 1 in the main paper. Furthermore, we denote the neighbouring lattice points of a feature point a by $\bar{N}(a)$. In addition, the barycentric weight between the lattice point l and feature point b is denoted with w_{lb}. Furthermore, the value at feature point b is denoted by v_b and the value at lattice point l is denoted by $\bar{v}_l$. Finally, the set of feature point scores is denoted by $\mathcal{Y} = \{y_b \mid b \in \{1 \dots n\}\}$, their set of values is denoted by $\mathcal{V} = \{v_b \mid b \in \{1 \dots n\}\}$ and the set of lattice point values is denoted by $\bar{\mathcal{V}} = \{\bar{v}_l \mid l \in \mathcal{P}\}$. The pseudocode of the algorithm is given in Algorithm 8.

Algorithm 8: Original filtering algorithm [21]

Data: Permutohedral lattice $\mathcal{P}$, set of feature point values $\mathcal{V}$

$\mathcal{V}' \leftarrow \mathbf{0} \quad \bar{\mathcal{V}} \leftarrow \mathbf{0} \quad \bar{\mathcal{V}}' \leftarrow \mathbf{0}$; // Initialization

forall $l \in \mathcal{P}$ **do**

  forall $b \in N(l)$ **do**

    $\bar{v}_l \leftarrow \bar{v}_l + w_{lb}\, v_b$; // Splatting

  end

end

$\bar{\mathcal{V}}' \leftarrow k \otimes \bar{V}$; // Blurring

forall $a \in \{1 \dots n\}$ **do**

  forall $l \in \bar{N}(a)$ **do**

    $v'_a \leftarrow v'_a + w_{la}\, \bar{v}'_l$; // Slicing

  end

end

A.2.2.2 Modified Filtering Algorithm

As mentioned in the main paper, the interval $[0, 1]$ is discretized into H bins. Note that each bin $h \in \{0 \dots H-1\}$ is associated with an interval which is identified as: $\left[\frac{h}{H-1}, \frac{h+1}{H-1}\right)$. Note that, the last bin (with bin id $H-1$) is associated with the interval $[1, \cdot)$. Since $y_b \leq 1$, this bin contains the feature points whose scores are exactly 1. Given the score y_b of the feature point b, its bin/level can be identified as

$$h_b = \lfloor y_b * (H-1) \rfloor \ , \tag{A.2.2.2}$$

where $\lfloor \cdot \rfloor$ denotes the standard floor function.

Furthermore, during splatting, the values v_b are accumulated to the neighbouring lattice point only if the lattice point is above or equal to the feature point level.

We denote the value at lattice point l at level h by $\bar{v}_{l:h}$. Formally, the barycentric interpolation at lattice point l at level h can be written as

$$\bar{v}_{l:h} = \sum_{\substack{b \in N(l) \\ h_b \leq h}} w_{lb}\, v_b \ . \tag{A.2.2.3}$$

Then, blurring is performed independently at each discrete level h. Finally, during slicing, the resulting values are interpolated at the feature point level. Our modified algorithm is given in Algorithm 9. In this algorithm, we denote the set of values corresponding to all the lattice points at level h as $\bar{\mathcal{V}}_h = \{v_{l:h} \mid l \in \mathcal{P}\}$.

Algorithm 9: Modified filtering algorithm

Data: Permutohedral lattice $\mathcal{P}$, set of feature point values $\mathcal{V}$, discrete levels H, set of scores $\mathcal{Y}$

$\mathcal{V}' \leftarrow \mathbf{0} \quad \bar{\mathcal{V}} \leftarrow \mathbf{0} \quad \bar{\mathcal{V}}' \leftarrow \mathbf{0}$; // Initialization
forall $l \in \mathcal{P}$ **do**
  forall $b \in N(l)$ **do**
    $h_b \leftarrow \lfloor y_b * (H-1) \rfloor$; // Splatting
    forall $h \in \{h_b \ldots H-1\}$ **do**
      /* Splat at the feature point's level and above */
      $\bar{v}_{l:h} \leftarrow \bar{v}_{l:h} + w_{lb}\, v_b$
    end
  end
end
forall $h \in \{0 \ldots H-1\}$ **do**
  $\bar{\mathcal{V}}'_h \leftarrow k \otimes \bar{\mathcal{V}}_h$; // Blurring at each level independently
end
forall $a \in \{1 \ldots n\}$ **do**
  $h_a \leftarrow \lfloor y_a * (H-1) \rfloor$; // Slicing
  forall $l \in \bar{N}(a)$ **do**
    $v'_a \leftarrow v'_a + w_{la}\, \bar{v}'_{l:h_a}$; // Slice at the feature point level
  end
end

Note that the above algorithm is given for the constraint $\mathbf{1}[y_a \geq y_b]$ (Eq. (14)). However, it is fairly easy to modify it for the $\mathbf{1}[y_a \leq y_b]$ constraint. In particular, one needs to change the interval identified by the bin h to: $\left(\frac{h-1}{H-1}, \frac{h}{H-1}\right]$. Using this fact, one can easily derive the splatting and slicing equations for the $\mathbf{1}[y_a \leq y_b]$ constraint. The algorithm given above introduces an approximation to the gradient computation that depends on the number of discrete bins H. However, this approximation can be eliminated by using a dynamic data structure which we briefly explain in the next section.

A.2.2.2.1 Adaptive Version of the Modified Filtering Algorithm Here, we briefly explain the adaptive version of our modified algorithm, which replaces the fixed discretization with a dynamic data structure. Effectively, discretization boils down to storing a vector of length H at each lattice point. Instead of such a fixed-length vector, one can use a dynamic data structure that grows with the number of different scores encountered at each lattice point in the splatting and blurring steps. In the worst case, i.e., when all the neighbouring feature points have different scores, the maximum number of values to store at a lattice point is

$$H = \max_{l} |N^2(l)| \ , \tag{A.2.2.4}$$

where $N^2(l)$ denotes the union of neighbourhoods of the lattice point l and its neighbouring lattice points (the vertices of the shaded hexagon in Fig. 1 in the main paper). In our experiments, we observed that $|N^2(l)|$ is usually less than 100, with an average around 10. Empirically, however, we found this dynamic version to be slightly slower than the static one. We conjecture that this is due to the static version benefitting from better compiler optimization. Furthermore, both the versions obtained results with similar precesion and therefore we used the static one for all our experiments.

A.2.3 Additional Experiments

Let us first explain the pixel compatibility function used in the experiments. We then turn to additional experiments.

A.2.3.1 Pixel Compatibility Function Used in the Experiments

As mentioned in the main paper, our algorithm is applicable to any pixel compatibility function that is composed of a mixture of Gaussian kernels. In all our experiments, we used two kernels, namely spatial kernel and bilateral kernel, similar to [5, 14]. Our pixel compatibility function can be written as

$$K_{ab} = w^{(1)} \exp\left(-\frac{|\mathbf{p}_a - \mathbf{p}_b|^2}{\sigma_1}\right) + w^{(2)} \exp\left(-\frac{|\mathbf{p}_a - \mathbf{p}_b|^2}{\sigma_{2:s}} - \frac{|\mathbf{I}_a - \mathbf{I}_b|^2}{\sigma_{2:c}}\right) \ , \tag{A.2.3.1}$$

where $\mathbf{p}_a$ denotes the (x, y) position of pixel a measured from top left and $\mathbf{I}_a$ denotes the (r, g, b) values of pixel a. Note that there are 5 learnable parameters: $w^{(1)}, \sigma_1, w^{(2)}, \sigma_{2:s}, \sigma_{2:c}$. These parameters are cross validated for different algorithms on each data set. The final cross validated parameters for MF and DC_{neg} are given in Table A.2. To perform this cross-validation, we ran Spearmint for 2 days for each algorithm on both datasets. Note that, due to this time limitation, we were able

Data set	Algorithm	$w^{(1)}$	σ_1	$w^{(2)}$	$\sigma_{2:s}$	$\sigma_{2:c}$
MSRC	MF	7.467846	1.000000	4.028773	35.865959	11.209644
	DC_{neg}	2.247081	3.535267	1.699011	31.232626	7.949970
Pascal	MF	100.000000	1.000000	74.877398	50.000000	5.454272
	DC_{neg}	0.500000	3.071772	0.960811	49.785678	1.000000

Table A.2: *Parameters tuned for MF and DC_{neg} on the MSRC and Pascal validation sets using Spearmint [42].*

to run approximately 1000 Spearmint iterations on MSRC but only 100 iterations on Pascal. This is due to bigger images and larger validation set on the Pascal dataset. Hence, it resulted in less accurate energy parameters.

A.2.3.2 Additional Segmentation Results

In this section we provide additional segmentation results.

A.2.3.2.1 Results on Parameters Tuned for MF The results for the parameters tuned for MF on the MSRC and Pascal datasets are given in Table A.3. In Fig. A.8, we show the assignment energy as a function of time for an image in MSRC (the tree image in Fig. A.9) and for an image in Pascal (the sheep image in Fig. A.9). Furthermore, we provide some of the segmentation results in Fig. A.9.

Interestingly, for the parameters tuned for MF, even though our algorithm obtains much lower energies, MF yields the best segmentation accuracy. In fact, one can argue that the parameters tuned for MF do not model the segmentation problem accurately, but were tuned such that the inaccurate MF inference yields good results. Note that, in the Pascal dataset, when tuned for MF, the Gaussian mixture coefficients are very high (see Table A.2). In such a setting, DC_{neg} ended up classifying all pixel in most images as background. In fact, SG-LP_ℓ was able to improve over DC_{neg} in only 1% of the images, whereas all our versions improved over DC_{neg} in roughly 25% of the images. Furthermore, our accelerated versions could not get any advantage over the standard version and resulted in similar run times. Note that, in most of the images, the *uncertain* pixels are in fact the entire image, as shown in Fig. A.9.

A.2.3.2.2 Summary We have evaluated all the algorithms using two different parameter settings. Therefore, we summarize the best segmentation accuracy obtained by each algorithm and the corresponding parameter setting in Table A.4. Note that, on MSRC, the best parameter setting for DC_{neg} corresponds to the

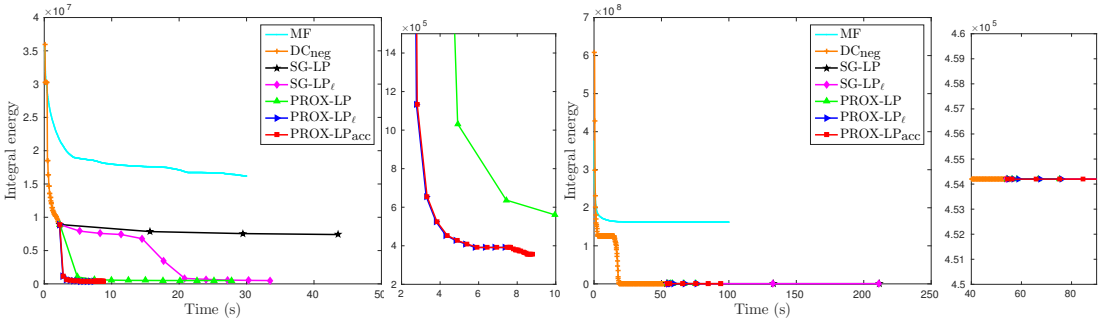


Figure A.8: *Assignment energy as a function of time for MF parameters for an image in (**left**) MSRC and (**right**) Pascal. A zoomed-in version is shown next to each plot. Except for MF, all the algorithms were initialized with DC_{neg}. For the MSRC image, PROX-LP clearly outperforms SG-LP$_\ell$ by obtaining much lower energies in fewer iterations, and the accelerated versions of our algorithm obtain roughly the same energy as PROX-LP but significantly faster. For the Pascal image, however, no LP algorithm is able to improve over DC_{neg}. Note that, in the Pascal dataset, for the MF parameters, DC_{neg} ended up classifying all pixel in most images as background (which yields low energy values) and no LP algorithm is able to improve over it.*

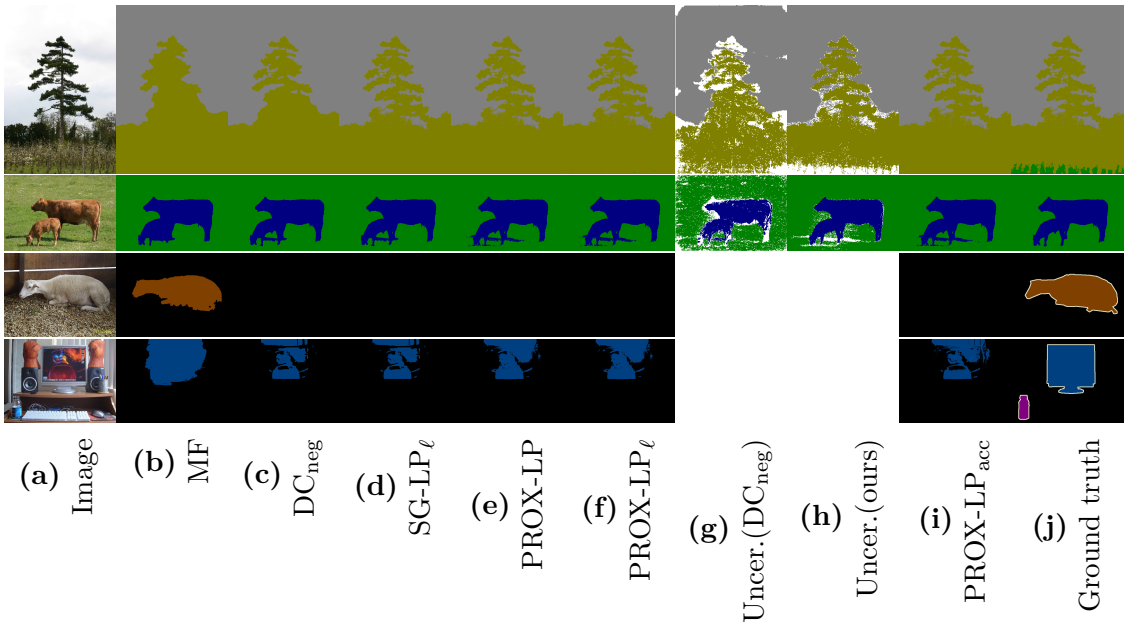


Figure A.9: *Results with MF parameters, for an image in (**top**) MSRC and (**bottom**) Pascal. The uncertain pixels identified by DC_{neg} and PROX-LP$_{acc}$ are marked in white. Note that, in MSRC all versions of our algorithm obtain visually good segmentations similar to MF (or better). In Pascal, the segmentation results are poor except for MF, even though they obtain much lower energies. We argue that, in this case, the energy parameters do not model the segmentation problem accurately.*

		MF5	MF	DC_{neg}	SG-LP_ℓ	PROX-LP	PROX-LP_ℓ	PROX-LP_{acc}	Ave. E ($\times 10^4$)	Ave. T (s)	Acc.	IoU
MSRC	MF5	-	0	0	0	0	0	0	2366.6	**0.2**	81.14	54.60
	MF	95	-	18	15	2	1	2	1053.6	13.0	**83.86**	**59.75**
	DC_{neg}	95	77	-	0	0	0	0	812.7	2.8	83.50	59.67
	SG-LP_ℓ	95	80	48	-	2	0	1	800.1	37.3	83.51	59.68
	PROX-LP	95	93	95	93	-	35	46	265.6	27.3	83.01	58.74
	PROX-LP_ℓ	95	94	94	94	59	-	43	**261.2**	13.9	82.98	58.62
	PROX-LP_{acc}	95	93	93	93	49	46	-	295.9	7.9	83.03	58.97
Pascal	MF5	-	-	1	1	0	0	0	40779.8	**0.8**	80.42	28.66
	MF	93	-	3	3	0	0	1	20354.9	21.7	**80.95**	**28.86**
	DC_{neg}	93	87	-	0	0	0	0	2476.2	39.1	77.77	14.93
	SG-LP_ℓ	93	87	1	-	0	0	0	2474.1	414.7	77.77	14.92
	PROX-LP	94	90	24	24	-	4	9	1475.6	81.0	78.04	15.79
	PROX-LP_ℓ	94	90	24	24	5	-	9	**1458.9**	82.7	78.04	15.79
	PROX-LP_{acc}	94	89	28	27	18	18	-	1623.7	83.9	77.86	15.18

Table A.3: *Results on the MSRC and Pascal datasets with the parameters tuned for MF. We show: the percentage of images where the row method strictly outperforms the column one on the final integral energy, the average integral energy over the test set, the average run time, the segmentation accuracy and the intersection over union score. Note that all versions of our algorithm obtain much lower energies than the baselines. However, as expected, lower energy does not correspond to better segmentation accuracy, mainly due to the less accurate energy parameters. Furthermore, the accelerated versions of our algorithm are similar in run time and obtain similar energies compared to PROX-LP.*

parameters tuned for MF. This is a strange result but can be explained by the fact that, as mentioned in the main paper, cross-validation was performed using the less accurate ground truth provided with the original dataset, but evaluation using the accurate ground truth annotations provided by [5].

Furthermore, in contrast to MSRC, the segmentation results of our algorithm on the Pascal dataset is not the state-of-the-art, even with the parameters tuned for DC_{neg}. This may be explained by the fact, that due to the limited cross-validation, the energy parameters obtained for the Pascal dataset is not accurate. Therefore, even though our algorithm obtained lower energies that was not reflected in the segmentation accuracy. Similar behaviour was observed in [14, 33].

A.2.3.3 Effect of the Proximal Regularization Constant

We plot the assignment energy as a function of time for an image in MSRC (the same image used to generate Fig. 2.6) by varying the proximal regularization constant λ. Here, we used the parameters tuned for DC_{neg}. The plot is shown in

Algorithm	MSRC			Pascal		
	Parameters	Ave. T (s)	Acc.	Parameters	Ave. T (s)	Acc.
MF5	MF	**0.2**	81.14	MF	**0.8**	80.42
MF	MF	13.0	83.86	MF	21.7	**80.95**
DC_{neg}	MF	2.8	83.50	DC_{neg}	3.7	80.43
SG-LP_{ℓ}	MF	37.3	83.51	DC_{neg}	84.4	80.49
PROX-LP	DC_{neg}	23.5	83.99	DC_{neg}	106.7	80.63
PROX-LP_{ℓ}	DC_{neg}	6.3	83.94	DC_{neg}	22.1	80.65
PROX-LP_{acc}	DC_{neg}	3.7	**84.16**	DC_{neg}	14.7	80.58

Table A.4: *Best segmentation results of each algorithm with their respective parameters, the average time on the test set and the segmentation accuracy. In MSRC, the best segmentation accuracy is obtained by PROX-LP_{acc} and in Pascal it is by MF. Note that, on MSRC, the best parameter setting for DC_{neg} corresponds to the parameters tuned for MF. This is due to the fact that cross-validation was performed on the less accurate ground truth but evaluation on the accurate ground truth annotations provided by [5]. Furthermore, the low segmentation performance of our algorithm on the Pascal dataset is may be due to less accurate energy parameters resulted from limited cross-validation.*

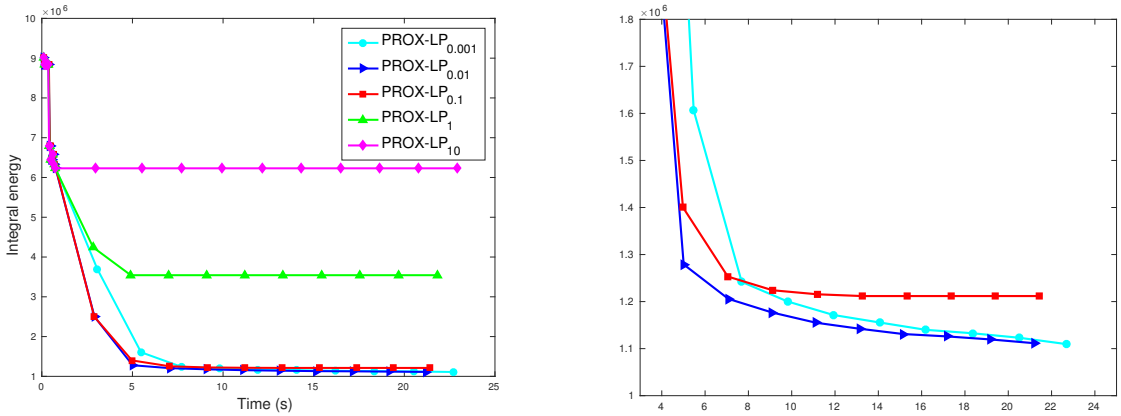
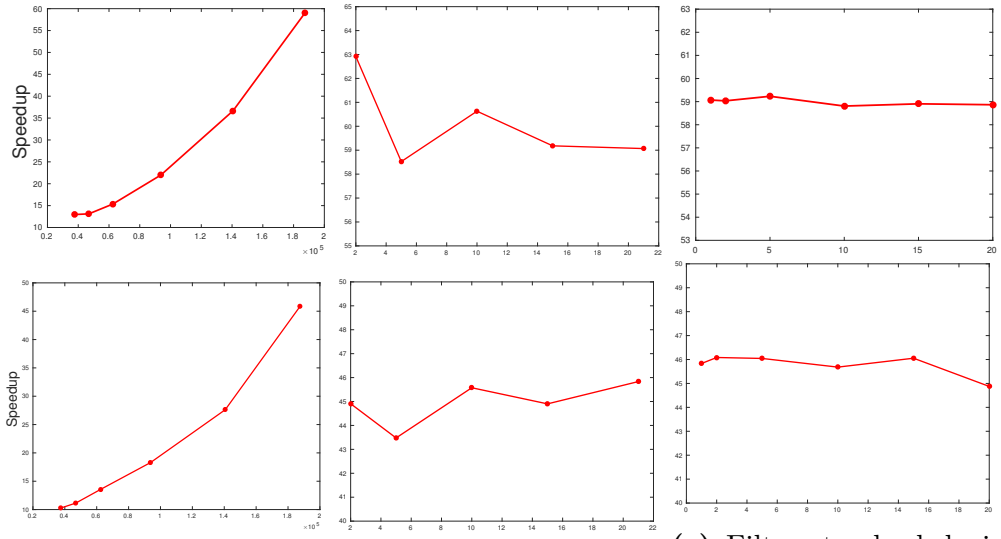


Figure A.10: *Assignment energy as a function of time for an image in MSRC, for different values of λ. The zoomed plot is shown on the right. Note that, for $\lambda = 0.1, 0.01, 0.001$, PROX-LP obtains similar energies in approximately the same run time.*

Fig. A.10. In summary, for a wide range of λ, PROX-LP obtains similar energies with approximately the same run time.

A.2.3.4 Modified Filtering Algorithm

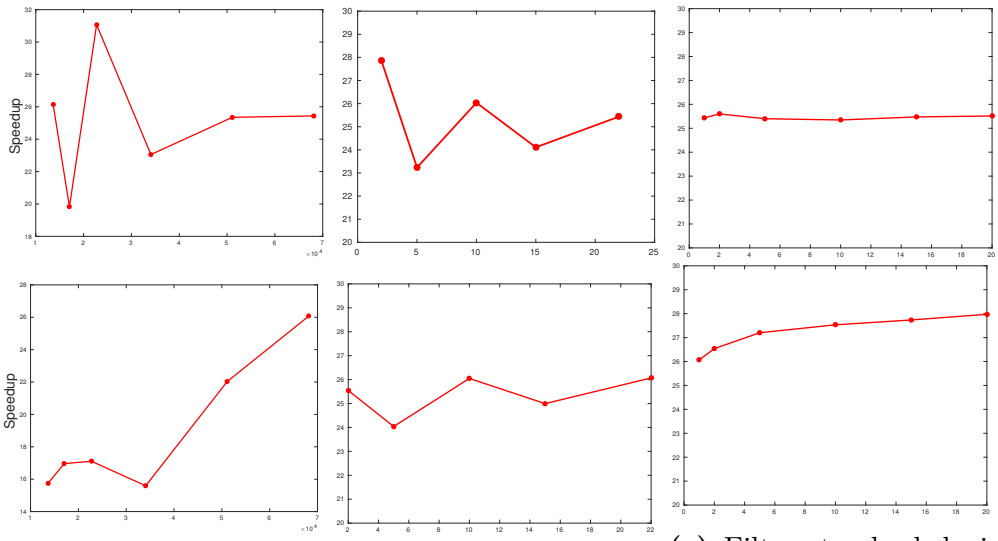
We compare our modified filtering method, described in Section 4, with the divide-and-conquer strategy of [14]. To this end, we evaluated both algorithms on one of the Pascal VOC test images (the sheep image in Fig. 2.7), but varying the image size, the number of labels and the Gaussian kernel standard deviation. The respective plots are shown in Fig. A.11. Note that, as claimed in the main paper,



(a) Number of pixels **(b)** Number of labels **(c)** Filter standard deviation

Figure A.11: *Speedup of our modified filtering algorithm over the divide-and-conquer strategy of [14] on a Pascal image,* ***top:*** *spatial kernel ($d = 2$),* ***bottom:*** *bilateral kernel ($d = 5$). Note that our speedup grows with the number of pixels and is approximately constant with respect to the number of labels and filter standard deviation.*

speedup with respect to the standard deviation is roughly constant. Similar plots for an MSRC image (the tree image in Fig. 2.7) are shown in Fig. A.12. In this case, speedup is around $15 - 32$, with around $23 - 32$ in the operating region of all versions of our algorithm.



(a) Number of pixels **(b)** Number of labels **(c)** Filter standard deviation

Figure A.12: *Speedup of our modified filtering algorithm over the divide-and-conquer strategy of [14] on a MSRC image,* ***top:*** *spatial kernel (d = 2),* ***bottom:*** *bilateral kernel (d = 5). Note that our speedup grows with the number of pixels and is approximately constant with respect to the number of labels and filter standard deviation.*

A.3 Adaptive Neural Compilation

A.3.1 Detailed Model Description

In this section, we are going to precisely define a non differentiable version model that we use in the main paper. This model can be seen as a recurrent network. Indeed, it takes as input an initial memory tape, performs a certain number of iterations and outputs a final memory tape. The memory tape is an array of M cells, where a cell is an element holding a single integer value. The internal state of this recurrent model are the memory, the registers and the instruction register. The registers are another set of R cells that are internal to the model. The instruction register is a single cell used in a specific way described later. These internal states are noted $\mathcal{M}^t = \{m_1^t, m_2^t, \ldots, m_M^t\}$, $\mathcal{R}^t = \{r_1^t, r_2^t, \ldots, r_R^t\}$ and $\mathcal{IR}^t$ for the memory, the registers and the instruction register respectively.

Figure 1 of the main paper describes in more detail how the different elements interact with each other. At each iteration, the Controller takes as input the value of the instruction register $\mathcal{IR}^t$ and outputs four values:

$$\mathtt{e}^t, \mathtt{a}^t, \mathtt{b}^t, \mathtt{o}^t = \text{Controller}(\mathcal{IR}^t). \tag{A.3.1.1}$$

The first value $\mathtt{e}^t$ is used to select one of the instruction of the Machine to execute at this iteration. The second and third values $\mathtt{a}^t$ and $\mathtt{b}^t$ will identify which registers

to use as the first and second argument for the selected instruction. The fourth value $\mathtt{o}^t$ identity the output register where to write the result of the executed instruction. The Machine then takes as input these four values and the internal state and computes the updated value of the internal state and a *stop* flag:

$$\mathcal{M}^{t+1}, \mathcal{R}^{t+1}, \mathcal{IR}^{t+1}, stop = \text{Machine}(\mathcal{M}^t, \mathcal{R}^t, \mathcal{IR}^t, \mathtt{e}^t, \mathtt{a}^t, \mathtt{b}^t, \mathtt{o}^t). \tag{A.3.1.2}$$

The *stop* flag is a binary flag. When its value is 1, it means that the model will stop the execution and the current memory state will be returned.

The Machine The machine is a deterministic function that increments the instruction register and executes the command given by the Controller to update the current internal state. The set of instructions that can be executed by the Machine can be found in Table 3.1b. Each instruction takes two values as arguments and returns a value. Additionally, some of these instructions have side effects. This mean that they do not just output a value, they perform another task. This other task can be for example to modify the content of the memory. All the considered side effects can be found in Table 3.1b. By convention, instructions that don't have a value to return and that are used only for their side-effect will return a value of 0.

The Controller The Controller is a function that takes as input a single value and outputs four different values. The Controller's internal parameters, the initial values for the registers and the initial value of the instruction register define uniquely a given Controller.

The usual choice in the literature is to use an LSTM network[61, 62, 65] as controller. Our choice was to instead use a simpler model. Indeed, our Controller associates a command to each possible value of the instruction register. Since the instruction register's value will increase by one at each iteration, this will enforce the Controller to encode in its weights what to do at each iteration. If we were using a recurrent controller the same instruction register could potentially be associated to different sets of outputs and we would lose this one to one mapping.

To make this clearer, we first rewrite the instruction register as an indicator vector with a 1 at the position of its value:

$$I_i = \begin{cases} 1 & \text{if } i = IR^t \\ 0 & \text{otherwise} \end{cases} . \tag{A.3.1.3}$$

In this case, we can write a single output a^t of the Controller as the result of a linear function of I:

$$\mathtt{a}^t = \mathtt{W}_\mathtt{a} * I, \tag{A.3.1.4}$$

where W_a is the 1xM matrix containing the value that need to be chosen as first arguments for each possible value of the instruction register and $*$ represent a matrix vector multiplication.

A.3.1.1 Mathematical Details of the Differentiable Model

In order to make the model differentiable, every value and every choice are replaced by probability distributions over the possible choices. The main paper introduces how, using convex combinations of probability, the execution of the Machine is made differentiable. We present here the mathematical formulation of this procedure for the case of the side-effects, which was eluded in the paper for space reasons.

STOP In the discrete model, the execution is halted when the STOP instruction is executed. However, in the differentiable model, the STOP instruction may be executed with a probability smaller than 1. To take this into account, when executing the model, we keep track of the probability that the program should have terminated before this iteration based on the probability associated to the STOP instruction at each iteration. Once this probability goes over a threshold $\eta_{\text{stop}} \in]0, 1]$, the execution is halted.

READ The mechanism is entirely the same as the one used to compute the arguments based on the registers and a probability distribution over the registers.

JEZ We note $\boldsymbol{\mathcal{IR}}^{t+1}_{jez}$ and $\boldsymbol{\mathcal{IR}}^{t+1}_{njez}$ the new value of $\boldsymbol{\mathcal{IR}}^{t}$ if we had respectively executed or not the JEZ instruction. We also have e^t_{jez} the probability of executing this instruction at iteration t. The new value of the instruction register is:

$$\boldsymbol{\mathcal{IR}}^{t+1} = \boldsymbol{\mathcal{IR}}^{t+1}_{njez} \cdot (1 - e^t_{jez}) + \boldsymbol{\mathcal{IR}}^{t+1}_{jez} \cdot e^t_{jez} \tag{A.3.1.5}$$

$\boldsymbol{\mathcal{IR}}^{t+1}_{jez}$ is himself computed based on several probability distribution. If we consider that the instruction JEZ is executed with probabilistic arguments **cond** and **label**, its value is given by

$$\boldsymbol{\mathcal{IR}}^{t+1}_{jez} = \textbf{label} \cdot \text{cond}_0 + \texttt{INC}(\boldsymbol{\mathcal{IR}}^{t}) \cdot (1 - \text{cond}_0) \tag{A.3.1.6}$$

With a probability equals to the one that the first argument is equal to zero, the new value of $\boldsymbol{\mathcal{IR}}^{t}$ is **label**. With the complement, it is equal to the incremented version of its current value, as the machine automatically increments the instruction register.

WRITE The mechanism is fairly similar to the one of the JEZ instruction. We note $\mathbf{M}^{t+1}_{WRITE}$ and $\mathbf{M}^{t+1}_{nWRITE}$ the new value of $\mathbf{M}^t$ if we had respectively executed or not the WRITE instruction. We also have e^t_{write} the probability of executing this instruction at iteration t. The new value of the memory matrix register is:

$$\mathbf{M}^{t+1} = \mathbf{M}^{t+1}_{nWRITE} \cdot (1 - e^t_{write}) + \mathbf{M}^{t+1}_{WRITE} \cdot e^t_{WRITE} \tag{A.3.1.7}$$

As with the JEZ instruction, the value of $\mathbf{M}^{t+1}_{WRITE}$ is dependent on the two probability distribution given as input: **addr** and **val**. The probability that the i-th cell of the memory tape contains the value j after the update is:

$$M^{t+1}_{i,j} = \text{addr}_i \cdot \text{val}_j + (1 - \text{addr}_i) \cdot M^t_{i,j} \tag{A.3.1.8}$$

Note that this can done using linear algebra operations so as to update everything in one global operation.

$$\mathbf{M}^{t+1} = \Big(((\mathbf{1} - \mathbf{addr})\mathbf{1}^T) \otimes \mathbf{M}^t\Big) + (\mathbf{addr}\ \mathbf{val}^T) \tag{A.3.1.9}$$

A.3.2 Specification of the Loss

This loss contains four terms that will balance the correctness of the learnt algorithm, proper usage of the stop signal and speed of the algorithms. The parameters defining the models are the weight of the Controller's function and the initial value of the registers. When running the model with the parameters θ, we consider that the execution ran for T time steps. We consider the memory to have a size M and that each number can be an integer between 0 and $M - 1$. $\mathbf{M}^t$ was the state of the memory at the t-th step. $\mathbf{T}$ and $\mathbf{C}$ are the target memory and the 0-1 mask of the elements we want to consider. All these elements are matrices where for example $\mathbf{M}^t_{i,j}$ is the probability of the i-th entry of the memory to take the value j at the step t. We also note $p_{\text{stop},t}$ the probability outputted by the Machine that it should have stopped before iteration t.

Correctness The first term corresponds to the correctness of the given algorithm. For a given input, we have the expected output and a mask. The mask allows us to know which elements in the memory we should consider when comparing the solutions. For the given input, we will compare the values specified by the mask of the expected output with the final memory tape provided by the execution. We compare them with the $\mathcal{L}_2$ distance in the probability space. Using the notations from above, we can write this term as:

$$L_c(\theta) = \sum_{i,j} \mathbf{C}_{i,j}(\mathbf{M}^T_{i,j}(\theta) - \mathbf{T}_{i,j})^2. \tag{A.3.2.1}$$

If we optimised only this first term, nothing would encourage the learnt algorithm to use the STOP instruction and halt as soon as it finished.

Halting To prevent programs to take an infinite amount of time without stopping, we defined a maximum number of iterations T_{max} after which the execution is halted. During training, we also add a penalty if the Controller didn't halt before this limit:

$$L_{sT_{max}}(\theta) = (1 - p_{\text{stop}-T}(\theta)) \cdot [T == T_{max}] \tag{A.3.2.2}$$

Efficiency If we consider only the above mentioned losses, the program will make sure to halt by itself but won't do it as early as possible. We incentivise this behaviour by penalising each iteration taken by the program where it does not stop:

$$L_t(\theta) = \sum_{t \in [1, T-1]} (1 - p_{\text{stop},t}(\theta)). \tag{A.3.2.3}$$

Confidence Moreover, we want the algorithm to have a good confidence to stop when it has found the correct output. To do so, we add the following term which will penalise probability of stopping if the current state of the memory is not the expected one:

$$L_s t(\theta) = \sum_{t \in [2, T]} \sum_{i,j} (p_{\text{stop},t}(\theta) - p_{\text{stop},t-1}(\theta)) \mathbf{C}_{i,j} (\mathbf{M}^t_{i,j}(\theta) - \mathbf{T}_{i,j})^2. \tag{A.3.2.4}$$

The increase in probability $(p_{\text{stop},t} - p_{\text{stop},t-1})$ corresponds to the probability of stopping exactly at iteration t. So, this is equivalent to the expected error made.

Total Loss The complete loss that we use is then the following:

$$L(\theta) = \alpha L_c(\theta) + \beta L_{sT_{max}}(\theta) + \gamma L_s t(\theta) + \delta L_t(\theta). \tag{A.3.2.5}$$

A.3.3 Distributed Representation of the Program

For the most of out experiments, the learned weights are fully interpretable as they first in the first type of interpretability. However, in some specific cases, under the pressure of our loss encouraging a smaller number of iterations, an interesting behavior emerges.

Remarks It is interesting to note that the decompiled version is not straightforward to interpret. Indeed when we reach a program that has non Dirac-delta distributions in its weights, we cannot perform the inverse of the one-to-one mapping performed by the compiler. In fact, it relies on this blurriness to be able to execute the program with a smaller number of instruction. Notably, by having some blurriness on the `JEZ` instruction, the program can hide additional instructions, by creating a distributed state. We now explain the mechanism used to achieve this.

Creating a Distributed State Consider the following program and assume that the initial value of $\mathcal{IR}$ is 0:

Initial Registers:

$R_1 = 0; R_2 = 1; R_3 = 4, R_4 = 0$

Program:

$0 : R_1 = \textbf{READ}\ (R_1, R_4)$

$1 : R_4 = \textbf{JEZ}\ (R_1, R_3)$

$2 : R_4 = \textbf{WRITE}(R_1, R_1)$

$3 : R_4 = \textbf{WRITE}(R_1, R_3)$

If you take this program and execute it for three iterations, it will: read the first value of the tape into R_1. Then, if this value is zero, it will jump to State 4, otherwise it will just increment $\mathcal{IR}$. This means that depending on the value that was in R_1, the next instruction that will be executed will be different (in this case, the difference between State 3 and State 4 is which registers they will be writing from). This is our standard way of implementing conditionals.

Imagine that, after learning, the second instruction in our example program has 0.5 probability of being a `JEZ` and 0.5 probability of being a `ZERO`. If the content of R_1 is a zero, according to the `JEZ`, we should jump to State 4, but this instruction is executed with a probability of 0.5. We also have 0.5 probability of executing the `ZERO` instruction, which would lead to State 3.

Therefore, $\mathcal{IR}$ is not a Dirac-delta distribution anymore but points to State 3 with probability 0.5 and State 4 with probability 0.5.

Exploiting a Distributed State To illustrate, we will discuss how the Controller computes **a** for a model with 3 registers. The Table A.5 show an example of some weights for such a controller.

	R_1	R_2	R_3
State 1	20	5	-20
State 2	-20	5	20

Table A.5: Controller Weights

If we are in State 1, the output of the controller is going to be

$$out = \text{softmax}([20, 5, -20]) = [0.9999..., 3e^{-7}, 4e^{-18}] \tag{A.3.3.1}$$

If we are in State 2, the output of the controller is going to be

$$out = \text{softmax}([-20, 5, 20]) = [4e^{-18}, 3e^{-7}, 0.9999...] \tag{A.3.3.2}$$

In both cases, the output of the controller is therefore going to be almost discrete. In State 1, R_1 would be chosen and in State 2, R_3 would be chosen.

However, in the case where we have a distributed state with probability 0.5 over State 1 and 0.5 over State 2, the output would be:

$$\begin{aligned} out &= \text{softmax}(0.5 * [-20, 5, 20] + 0.5[20, 5, -20]) \\ &= \text{softmax}([0, 10, 0]) \\ &= [4e^{-5}, 0.999, 4e^{-5}]. \end{aligned} \tag{A.3.3.3}$$

Note that the result of the distributed state is actually different from the result of the discrete states. Moreover it is still a discrete choice of the second register.

Because this program contains distributed elements, it is not possible to perform the one-to-one mapping between the weights and the lines of code. Though every instruction executed by the program, except for the `JEZ`, are binary. This means that this model can be translated to a regular program that will take exactly the same runtime, but will require more lines of codes than the number of lines in the matrix.

A.3.4 Alternative Learning Strategies

A critique that can be made to this method is that we will still initialise close to a local minimum. Another approach might be to start from a random initialisation but adding a penalty on the value of the weights such that they are encourage to be close to the generic algorithm. This can be seen as $\mathcal{L}_2$ regularisation but instead of pushing the weights to 0, we push then with the value corresponding to the generic algorithm. If we start with a very high value of this penalty but use an annealing schedule where its importance is very quickly reduced, this is going to be equivalent to the previous method.

A.3.5 Possible Extension

A.3.5.1 Making the Objective Function Differentiable

These experiments showed that we can transform any program that perform a mapping between an input memory tape to an output memory tape to a set of parameters and execute it using our model. The first point we want to make here is that this means that we take any program and transform it into a differentiable function easily. For example, if we want to learn a model that given a graph and

two nodes a and b, will output the list of nodes to go through to go from a to b in the shortest amount of time. We can easily define the loss of the length of the path outputted by the model. Unfortunately, the function that computes this length from the set of nodes is not differentiable. Here we could implement this function in our model and use it between the prediction of the model and the loss function to get an end to end trainable system.

A.3.5.2 Beyond Mimicking and Towards Open Problems

It would even be possible to generalise our learning procedure to more complex problems for which we don't have a ground truth output. For example, we could consider problems where the exact answer for a given input is not computable or not unique. If the goodness of a solution can be computed easily, this value could be used as training objective. Any program giving a solution could be used as initialisation and our framework would improve it, making it generate better solutions.

A.3.6 Example Tasks

This section will present the programs that we use as initialisation for the experiment section.

A.3.6.1 Access

In this task, the first element in the memory is a value k. Starting from the second element, the memory contains a zero-terminated list. The goal is to access the k-th element in the list that is zero-indexed. The program associated with this task can be found in Listing A.1.

```
1 var k = 0
2 k = READ(0)
3 k = INC(k)
4 k = READ(k)
5 WRITE(0, k)
6 STOP()
```

Listing A.1: Access Task

Example input:	6	9	1	2	7	9	8	1	3	5
Output:	1	9	1	2	7	9	8	1	3	5

A.3.6.2 Copy

In this task, the first element in the memory is a pointer p. Starting from the second element, the memory contains a zero-terminated list. The goal is to copy this list at the given pointer. The program associated with this task can be found in Listing A.2.

```
1  var read_addr = 0
2  var read_value = 0
3  var write_addr = 0
4
5  write_addr = READ(0)
6  l_loop: read_value = READ(read_addr)
7  JEZ(read_value, l_stop)
8  WRITE(write_addr, read_value)
9  read_addr = INC(read_addr)
10 write_addr = INC(write_addr)
11 JEZ(0, l_loop)
12
13 l_stop: STOP()
```

Listing A.2: Copy Task

Example input:	9	11	3	1	5	14	0	0	0	0	0	0	0	0	0
Output:	9	11	3	1	5	14	0	0	0	11	3	1	5	14	0

A.3.6.3 Increment

In this task, the memory contains a zero-terminated list. The goal is to increment each value in the list by 1. The program associated with this task can be found in Listing A.3.

```
1  var read_addr = 0
2  var read_value = 0
3
4  l_loop: read_value = READ(read_addr)
5  JEZ(read_value, l_stop)
6  read_value = INC(read_value)
7  WRITE(read_addr, read_value)
8  read_addr = INC(read_addr)
9  JEZ(0, l_loop)
10
11 l_stop: STOP()
```

Listing A.3: Increment Task

Example input:	1	2	2	3	0	0	0
Output:	2	3	3	4	0	0	0

A.3.6.4 Reverse

In this task, the first element in the memory is a pointer p. Starting from the second element, the memory contains a zero-terminated list. The goal is to copy this list at the given pointer in the reverse order. The program associated with this task can be found in Listing A.4.

```
var read_addr = 0
var read_value = 0
var write_addr = 0

write_addr = READ(write_addr)
l_count_phase: read_value = READ(read_addr)
JEZ(read_value, l_copy_phase)
read_addr = INC(read_addr)
JEZ(0, l_count_phase)

l_copy_phase: read_addr = DEC(read_addr)
JEZ(read_addr, l_stop)
read_value = READ(read_addr)
WRITE(write_addr, read_value)
write_addr = INC(write_addr)
JEZ(0, l_copy_phase)

l_stop: STOP()
```

Listing A.4: Reverse Task

Example input:	5	7	2	13	14	0	0	0	0	0	0	0	0	0	0
Output:	5	7	2	13	14	14	13	2	7	0	0	0	0	0	0

A.3.6.5 Permutation

In this task, the memory contains two zero-terminated list one after the other. The first contains a set of indices. the second contains a set of values. The goal is to fill the first list with the values in the second list at the given index. The program associated with this task can be found in Listing A.5.

```
var read_addr = 0
var read_value = 0
var write_offset = 0

l_count_phase: read_value = READ(write_offset)
write_offset = INC(write_offset)
JEZ(read_value, l_copy_phase)
JEZ(0, l_count_phase)

l_copy_phase: read_value = DEC(read_addr)
JEZ(read_value, l_stop)
read_value = ADD(write_offset, read_value)
read_value = READ(read_value)
WRITE(read_addr, read_value)
read_addr = INC(read_addr)
JEZ(0, l_copy_phase)
l_stop: STOP()
```

Listing A.5: Permutation Task

Example input:	2	1	3	0	13	4	6	0	0	0	0	0	0	0	0
Output:	4	13	6	0	13	4	6	0	0	0	0	0	0	0	0

A.3.6.6 Swap

In this task, the first two elements in the memory are pointers p and q. Starting from the third element, the memory contains a zero-terminated list. The goal is to swap the elements pointed by p and q in the list that is zero-indexed. The program associated with this task can be found in Listing A.6.

```
var p = 0
var p_val = 0
var q = 0
var q_val = 0

p = READ(0)
q = READ(1)
p_val = READ(p)
q_val = READ(q)
WRITE(q, p_val)
WRITE(p, q_val)
STOP()
```

Listing A.6: Swap Task

Example input:	1	3	7	6	7	5	2	0	0	0
Output:	1	3	7	5	7	6	2	0	0	0

A.3.6.7 ListSearch

In this task, the first three elements in the memory are a pointer to the head of the linked list, the value we are looking for v and a pointer to a place in memory where to store the result. The rest of the memory contains the linked list. Each element in the linked list is two values, the first one is the pointer to the next element, the second is the value contained in this element. By convention, the last element in the list points to the address 0. The goal is to return the pointer to the first element whose value is equal to v. The program associated with this task can be found in Listing A.7.

```
var p_out = 0
var p_current = 0
var val_current = 0
var val_searched = 0

val_searched = READ(1)
p_out = READ(2)
l_loop: p_current = READ(p_current)
val_current = INC(p_current)
val_current = READ(val_current)
val_current = SUB(val_current, val_searched)
JEZ(val_current, l_stop)
JEZ(0, l_loop)
l_stop: WRITE(p_out, p_current)
STOP()
```

Listing A.7: ListSearch Task

Example input:	11	10	2	9	4	3	10	0	6	7	13	5	12	0	0
Output:	11	10	5	9	4	3	10	0	6	7	13	5	12	0	0

A.3.6.8 ListK

In this task, the first three elements in the memory are a pointer to the head of the linked list, the number of hops we want to do k in the list and a pointer to a place in memory where to store the result. The rest of the memory contains the linked list. Each element in the linked list is two values, the first one is the pointer to the next element, the second is the value contained in this element. By convention, the last element in the list points to the address 0. The goal is to return the value of the k-th element of the linked list. The program associated with this task can be found in Listing A.8.

```
1  var p_out = 0
2  var p_current = 0
3  var val_current = 0
4  var k = 0
5
6  k = READ(1)
7  p_out = READ(2)
8  l_loop: p_current = READ(p_current)
9  k = DEC(k)
10 JEZ(k, l_stop)
11 JEZ(0, l_loop)
12 l_stop: p_current = INC(p_current)
13 p_current = READ(p_current)
14 WRITE(p_out, p_current)
15 STOP()
```

Listing A.8: ListK Task

Example input:	3	2	2	9	15	0	0	0	1	15	17	7	13	0	0	11
Output:	3	2	17	9	15	0	0	0	1	15	17	7	13	0	0	11

10	0	0	0
10	0	0	0

A.3.6.9 Walk BST

In this task, the first two elements in the memory are a pointer to the head of the BST and a pointer to a place in memory where to store the result. Starting at the third element, there is a zero-terminated list containing the instructions on how to traverse in the BST. The rest of the memory contains the BST. Each element in the BST has three values, the first one is the value of this node, the second is the pointer to the left node and the third is the pointer to the right element. By convention, the leafs points to the address 0. The goal is to return the value of the node we get at after following the instructions. The instructions are 1 or 2 to go respectively to the left or the right. The program associated with this task can be found in Listing A.9.

```
var p_out = 0
var p_current = 0
var p_instr = 0
var instr = 0

p_current = READ(0)
p_out = READ(1)
instr = READ(2)

l_loop: JEZ(instr, l_stop)
p_current = ADD(p_current, instr)
p_current = READ(p_current)
p_instr = INC(p_instr)
JEZ(0, l_loop)

l_stop: p_current = READ(p_current)
WRITE(p_out, p_current)
STOP()
```

Listing A.9: WalkBST Task

	Example input:								12	1	1	2	0	0	15	0	9	23	0	0	11	15	6
	Output:								12	10	1	2	0	0	15	0	9	23	0	0	11	15	6
8	0	24	0	0	0	0	0	0	10	0	0	0	0	0									
8	0	24	0	0	0	0	0	0	10	0	0	0	0	0									

A.3.6.10 Merge

In this task, the first three elements in the memory are pointers to respectively, the first list, the second list and the output. The two lists are zero-terminated sorted lists. The goal is to merge the two lists into a single sorted zero-terminated list that starts at the output pointer. The program associated with this task can be found in Listing A.10.

```
var p_first_list = 0
var val_first_list = 0
var p_second_list = 0
var val_second_list = 0
var p_output_list = 0
var min = 0

p_first_list = READ(0)
p_second_list = READ(1)
p_output_list = READ(2)

l_loop: val_first_list = READ(p_first_list)
val_second_list = READ(p_second_list)
JEZ(val_first_list, l_first_finished)
JEZ(val_second_list, l_second_finished)
min = MIN(val_first_list, val_second_list)
min = SUB(val_first_list, min)
JEZ(min, l_first_smaller)

WRITE(p_output_list, val_first_list)
p_output_list = INC(p_output_list)
p_first_list = INC(p_first_list)
JEZ(0, l_loop)

l_first_smaller: WRITE(p_output_list, val_second_list)
p_output_list = INC(p_output_list)
p_second_list = INC(p_second_list)
JEZ(0, l_loop)

l_first_finished: p_first_list = ADD(p_second_list, 0)
val_first_list = ADD(val_second_list, 0)

l_second_finished: WRITE(p_output_list, val_first_list)
p_first_list = INC(p_first_list)
p_output_list = INC(p_output_list)
val_first_list = READ(p_first_list)
JEZ(val_first_list, l_stop)
JEZ(0, l_second_finished)

l_stop: STOP()
```

Listing A.10: Merge Task

	Example input:	3	8	11	27	17	16	1	0	29	26	0	0	0	0	0
	Output:	3	8	11	27	17	16	1	0	29	26	0	29	27	26	17

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	0	0	0	0	0	0	0	0	0	0	0	0	0

A.3.6.11 Dijkstra

In this task, we are provided with a graph represented in the input memory as follow. The first element is a pointer p_out indicating where to write the results. The following elements contain a zero-terminated array with one entry for each vertex in the graph. Each entry is a pointer to a zero-terminated list that contains a pair of values for each outgoing edge of the considered node. Each pair of value contains first the index in the first array of the child node and the second value contains the cost of this edge. The goal is to write a zero-terminated list at the address provided by p_out that will contain the value of the shortest path from the first node in the list to this node. The program associated with this task can be found in Listings A.11 and A.12.

```
var min = 0
var argmin = 0

var p_out = 0
var p_out_temp = 0
var p_in = 1
var p_in_temp = 1

var nnodes = 0

var zero = 0
var big = 99

var tmp_node = 0
var tmp_weight = 0
var tmp_current = 0
var tmp = 0

var didsmth = 0

p_out = READ(p_out)
p_out_temp = ADD(p_out, zero)

tmp_current = INC(zero)
l_loop_nnodes:tmp = READ(p_in_temp)
JEZ(tmp, l_found_nnodes)
WRITE(p_out_temp, big)
p_out_temp = INC(p_out_temp)
WRITE(p_out_temp, tmp_current)
p_out_temp = INC(p_out_temp)
p_in_temp = INC(p_in_temp)
nnodes = INC(nnodes)
JEZ(zero, l_loop_nnodes)

l_found_nnodes:WRITE(p_out, zero)
JEZ(zero, l_find_min)
l_min_return:p_in_temp = ADD(p_in, argmin)
p_in_temp = READ(p_in_temp)

l_loop_sons:tmp_node = READ(p_in_temp)
JEZ(tmp_node, l_find_min)
tmp_node = DEC(tmp_node)
p_in_temp = INC(p_in_temp)
tmp_weight = READ(p_in_temp)
p_in_temp = INC(p_in_temp)
```

Listing A.11: Dijkstra Algorithm (Part 1)

```
57 p_out_temp = ADD(p_out, tmp_node)
58 p_out_temp = ADD(p_out_temp, tmp_node)
59 tmp_current = READ(p_out_temp)
60 tmp_weight = ADD(min, tmp_weight)
61
62 tmp = MIN(tmp_current, tmp_weight)
63 tmp = SUB(tmp_current, tmp)
64 JEZ(tmp, l_loop_sons)
65 WRITE(p_out_temp, tmp_weight)
66 JEZ(zero, l_loop_sons)
67
68 l_find_min:p_out_temp = DEC(p_out)
69 tmp_node = DEC(zero)
70 min = ADD(big, zero)
71 argmin = DEC(zero)
72
73 l_loop_min:p_out_temp = INC(p_out_temp)
74 tmp_node = INC(tmp_node)
75 tmp = SUB(tmp_node, nnodes)
76 JEZ(tmp, l_min_found)
77
78 tmp_weight = READ(p_out_temp)
79
80 p_out_temp = INC(p_out_temp)
81 tmp = READ(p_out_temp)
82 JEZ(tmp, l_loop_min)
83
84 tmp = MAX(min, tmp_weight)
85 tmp = SUB(tmp, tmp_weight)
86 JEZ(tmp, l_loop_min)
87 min = ADD(tmp_weight, zero)
88 argmin = ADD(tmp_node, zero)
89 JEZ(zero, l_loop_min)
90
91 l_min_found:tmp = SUB(min, big)
92 JEZ(tmp, l_stop)
93 p_out_temp = ADD(p_out, argmin)
94 p_out_temp = ADD(p_out_temp, argmin)
95 p_out_temp = INC(p_out_temp)
96 WRITE(p_out_temp, zero)
97 JEZ(zero, l_min_return)
98
99 l_stop:STOP()
```

Listing A.12: Dijkstra Algorithm (Part 2)

Example omitted for space reasons

A.3.7 Learned Optimization: Case Study

Here we present an analysis of the optimization achieved by the ANC. We take the example of the **ListK** task and study the difference between the learned program and the initialisation used.

A.3.7.1 Representation

The representation chosen is under the form of the intermediary representation described in Figure (2b) of the main paper. Based on the parameters of the Controller, we can recover the approximate representation described in Figure (2b) of the main paper: 1For each possible "discrete state" of the instruction register, we can compute the commands outputted by the controller. We report the most probable value for each distribution, as well as the probability that the compiler would assign to this value. If no value has a probability higher than 0.5, we only report a neutral token (`R-`, `-`, `NOP`).

A.3.7.2 Biased ListK

Figure A.13 represents the program that was used as initialisation to the optimization problem. This is the direct result from the compilation performed by the Neural Compiler of the program described in Listing A.8. A version with a probability of 1 for all necessary instructions would have been easily obtained but not amenable to learning.

Figure A.14 similarly describes the program that was obtained after learning.

As a remainder, the bias introduced in the ListK task is that the linked list is well organised in memory. In the general case, the element could be in any order. An input memory tape to the problem of asking for the third element in the linked list containing $\{4, 5, 6, 7\}$ would be:

9	3	2	0	0	11	5	0	7	5	4	7	6	0	0	0	0	0	0	0

or

5	3	2	0	0	7	4	15	5	0	7	0	0	0	0	9	6	0	0	0

In the biased version of the task, all the elements are arranged in order and contiguously positioned on the tape. The only valid representation of this problems is:

3	3	2	5	4	7	5	9	6	0	7	0	0	0	0	0	0	0	0	0

A.3.7.3 Solutions

Because of the additional structure of the problem, the bias in the data, a more efficient algorithm to find the solution exists. Let us dive into the comparison of the two different solutions.

Both use their first two states to read the parameters of the given instance of the task. Which element of the list should be returned is read at line (`0:`) and where to write the returned value is read at line (`1:`). Step (`2:`) to (`6:`) are dedicated to putting the address of the k-th value of the linked list into the registers `R1`. Step (`7:`) to (`9:`) perform the same task in both solution: reading the value at the address contained in `R1`, writing it at the desired position and stopping.

The difference between the two programs lies in how they put the address of the k-th value into `R1`.

Generic The initialisation program, used for initialisation, works in the most general case so needs to perform a loop where it put the address of the next element in the linked list in `R1` (`2:`), decrement the number of jumps remaining to be made (`3:`), checking whether the wanted element has been reached (`4:`) and going back to the start of the loop if not (`5:`). Once the desired element is reached, `R1` is incremented so as to point on the value of the linked list element.

Specific On the other hand, in the biased version of the problem, the position of the desired value can be analytically determined. The function parameters occupy the first three cells of the tape. After those, each element of the linked list will occupy two cells (one for the pointer to the next address and one for the value). Therefore, the address of the desired value is given by

$$\begin{aligned} \texttt{R1} &= 3 + (2 * (k - 1) + 1) - 1 + 1 \\ &= 3 + 2 * k - 1 \end{aligned} \tag{A.3.7.1}$$

(the -1 comes from the fact that the address are 0-indexed and the final +1 from the fact that we are interested in the position of the value and not of the pointer.)

The way this is computed is as follows:

- `R1` $= 3 + k$ by adding the constant 3 to the registers `R2` containing K. (`2:`)
- `R2` $= k - 1$ (`3:`)
- `R1` $= 3 + 2 * k - 1$ by adding the now reduced value of `R2`. (`6:`)

The algorithm implemented by the learned version is therefore much more efficient for the biased dataset, due to its capability to ignore the loop.

A.3.7.4 Failure analysis

An observation that can be made is that in the learned version of the program, Step (`4:`) and (`5:`) are not contributing to the algorithms. They execute instructions that have no side effect and store the results into the registers `R7` that is never used later in the execution.

The learned algorithm could easily be more efficient by not performing these two operations. However, such an optimization, while perhaps trivial for a standard compiler, capable of detecting unused values, is fairly hard for our optimisers to discover. Because we are only doing gradient descent, the action of "moving some instructions earlier in the program" which would be needed here to make the useless instructions disappear, is fairly hard, as it involves modifying several rows of the program at once in a coherent manner.

```
R1 = 0 (0.88)
R2 = 1 (0.88)
R3 = 2 (0.88)
R4 = 6 (0.88)
R5 = 0 (0.88)
R6 = 2 (0.88)
R7 = - (0.05)

Initial State: 0 (0.88)

0:  R2 (0.96)   = READ (0.93)   [ R2 (0.96)     , R- (0.14)     ]
1:  R3 (0.96)   = READ (0.93)   [ R3 (0.96)     , R- (0.14)     ]
2:  R1 (0.96)   = READ (0.93)   [ R1 (0.96)     , R- (0.14)     ]
3:  R2 (0.96)   = DEC  (0.93)   [ R2 (0.96)     , R- (0.14)     ]
4:  R7 (0.96)   = JEZ  (0.93)   [ R2 (0.96)     , R4 (0.96)     ]
5:  R7 (0.96)   = JEZ  (0.93)   [ R5 (0.96)     , R6 (0.96)     ]
6:  R1 (0.96)   = INC  (0.93)   [ R1 (0.96)     , R- (0.14)     ]
7:  R1 (0.96)   = READ (0.93)   [ R1 (0.96)     , R- (0.14)     ]
8:  R7 (0.96)   = WRIT (0.93)   [ R3 (0.96)     , R1 (0.96)     ]
9:  R7 (0.96)   = STOP (0.93)   [ R- (0.14)     , R- (0.14)     ]
10:     R- (0.16)   = NOP  (0.11)   [ R- (0.16)     , R- (0.16)     ]
11:     R- (0.16)   = NOP  (0.11)   [ R- (0.18)     , R- (0.16)     ]
12:     R- (0.17)   = NOP  (0.1)    [ R- (0.17)     , R- (0.17)     ]
13:     R- (0.16)   = NOP  (0.11)   [ R- (0.17)     , R- (0.16)     ]
14:     R- (0.17)   = NOP  (0.11)   [ R- (0.16)     , R- (0.16)     ]
15:     R- (0.16)   = NOP  (0.1)    [ R- (0.16)     , R- (0.17)     ]
16:     R- (0.17)   = NOP  (0.11)   [ R- (0.16)     , R- (0.18)     ]
17:     R- (0.18)   = NOP  (0.1)    [ R- (0.18)     , R- (0.17)     ]
18:     R- (0.17)   = NOP  (0.1)    [ R- (0.16)     , R- (0.16)     ]
19:     R- (0.15)   = NOP  (0.11)   [ R- (0.16)     , R- (0.17)     ]
```

Figure A.13: Initialisation used for the learning of the ListK task.

```
R1 = 3 (0.99)
R2 = 1 (0.99)
R3 = 2 (0.99)
R4 = 10 (0.53)
R5 = 0 (0.99)
R6 = 2 (0.99)
R7 = 7 (0.99)

Initial State: 0 (0.99)

0:  R2 (0.99)   = READ (0.99)   [ R2 (1)    , R1 (0.97)     ]
1:  R6 (0.99)   = READ (0.99)   [ R3 (0.99)     , R6 (0.5)      ]
2:  R1 (1)      = ADD  (0.99)   [ R1 (0.99)     , R2 (1)    ]
3:  R2 (1)      = DEC  (0.99)   [ R2 (1)    , R1 (0.99)     ]
4:  R7 (0.99)   = MAX  (0.99)   [ R2 (0.99)     , R1 (0.51)     ]
5:  R7 (0.99)   = INC  (0.99)   [ R6 (0.7)      , R1 (0.89)     ]
6:  R1 (1)      = ADD  (0.99)   [ R1 (0.99)     , R2 (1)    ]
7:  R1 (0.99)   = READ (0.99)   [ R1 (0.99)     , R1 (0.53)     ]
8:  R7 (0.99)   = WRIT (0.99)   [ R6 (1)    , R1 (0.99)     ]
9:  R7 (0.9)    = STOP (0.99)   [ R6 (0.98)     , R1 (0.99)     ]
10:     R2 (0.99)   = STOP (0.96)   [ R1 (0.52)     , R1 (0.99)     ]
11:     R1 (0.98)   = ADD  (0.73)   [ R4 (0.99)     , R2 (0.99)     ]
12:     R3 (0.98)   = ADD  (0.64)   [ R6 (0.99)     , R1 (0.99)     ]
13:     R3 (0.87)   = STOP (0.65)   [ R3 (0.52)     , R1 (0.99)     ]
14:     R3 (0.89)   = STOP (0.62)   [ R6 (0.99)     , R2 (0.62)     ]
15:     R3 (0.99)   = STOP (0.65)   [ R3 (0.99)     , R2 (0.71)     ]
16:     R3 (0.99)   = NOP  (0.45)   [ R6 (0.99)     , R1 (0.99)     ]
17:     R2 (0.99)   = INC  (0.56)   [ R6 (0.7)      , R1 (0.98)     ]
18:     R3 (0.99)   = STOP (0.65)   [ R3 (0.99)     , R1 (0.99)     ]
19:     R3 (0.98)   = STOP (0.98)   [ R2 (0.62)     , R1 (0.79)     ]
```

Figure A.14: Learnt program for the listK task

A.4 Learning to Superoptimize Programs

A.4.1 Hyperparameters

A.4.1.1 Architectures

The output size of 9 corresponds to the types of move. The output size of 2903 correspond to the number of possible instructions that Stoke can use during a rewrite. This is smaller that the 3874 that are possible to find in an original program.

Outputs	Bias (9) SoftMax	Bias (2903) SoftMax

Table A.6: Architecture of the Bias

Embedding	Linear (3874 $\rightarrow$ 100) + ReLU Linear (100 $\rightarrow$ 300) + ReLU Linear (300 $\rightarrow$ 300) + ReLU	
Outputs	Linear (300 $\rightarrow$ 9) SoftMax	Linear (300 $\rightarrow$ 2903) SoftMax

Table A.7: Architecture of the Multi Layer Perceptron

A.4.1.2 Training Parameters

All of our models are trained using the Adam [76] optimizer, with its default hyper-parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. We use minibatches of size 32.

The learning rate were tuned by observing the evolution of the loss on the training datasets for the first iterations. The picked values are given in Table A.8. Those learning rates are divided by the size of the minibatches.

	Hacker's Delight	Synthetic
Bias	1	10
MLP	0.01	0.1

Table A.8: Values of the Learning rate used.

A.4.2 Structure of the Proposal Distribution

The sampling process of a move is a hierarchy of sampling step. The easiest way to represent it is as a generative model for the program transformations. Depending on what type of move is sampled, different series of sampling steps have to be performed. For a given move, all the probabilities are sampled independently so

the probability of proposing the move is the product of the probability of picking each of the sampling steps. The generative model is defined in Figure A.15. It is going to be parameterized by the the parameters of each specific probability distribution it samples from. The default Stoke version uses uniform probabilities over all of those elementary distributions.

```
def proposal(current_program):
    move_type = sample(categorical(all_move_type))
    if move_type == 1: % Add empty Instruction
       pos = sample(categorical(all_positions(current_program)))
       return (ADD_NOP, pos)

    if move_type == 2: % Delete an Instruction
       pos = sample(categorical(all_positions(current_program)))
       return (DELETE, pos)

    if move_type == 3: % Instruction Transform
       pos = sample(categorical(all_positions(current_program)))
       instr = sample(categorical(set_of_all_instructions))
       arity = nb_args(instr)
       for i = 1, arity:
           possible_args = possible_arguments(instr, i)
           % get one of the arguments that can be used as i-th
           % argument for the instruction 'instr'.
           operands[i] = sample(categorical(possible_args))
       return (TRANSFORM, pos, instr, operands)

    if move_type == 4: % Opcode Transform
       pos = sample(categorical(all_positions(current_program)))
       args = arguments_at(current_program, pos)
       instr = sample(categorical(possible_instruction(args)))
       % get an instruction compatible with the arguments
       % that are in the program at line pos.
       return(OPCODE_TRANSFORM, pos, instr)

    if move_type == 5: % Opcode Width Transform
       pos = sample(categorical(all_positions(current_program))
       curr_instr = instruction_at(current_program, pos)
       instr = sample(categorical(same_memonic_instr(curr_instr))
       % get one instruction with the same memonic that the
       % instruction 'curr_instr'.
       return (OPCODE_TRANSFORM, pos, instr)

    if move_type == 6: % Operand transform
       pos = sample(categorical(all_positions(current-program))
       curr_instr = instruction_at(current_program, pos)
       arg_to_mod = sample(categorical(args(curr_instr)))
       possible_args = possible_arguments(curr_instr, arg_to_mod)
       new_operand = sample(categorical(possible_args))
       return (OPERAND_TRANSFORM, pos, arg_to_mod, new_operand)

    if move_type == 7: % Local swap transform
       block_idx = sample(categorical(all_blocks(current_program)))
       possible_pos = pos_in_block(current_program, block_idx)
       pos_1 = sample(categorical(possible_pos))
       pos_2 = sample(categorical(possible_pos))
       return (SWAP, pos_1, pos_2)

    if move_type == 8: % Global swap transform
       pos_1 = sample(categorical(all_positions(current_program)))
       pos_2 = sample(categorical(all_positions(current_program)))
       return (SWAP, pos_1, pos_2)

    if move_type == 9: % Rotate transform
       pos_1 = sample(categorical(all_positions(current_program)))
       pos_2 = sample(categorical(all_positions(current_program)))
       return (ROTATE, pos_1, pos_2)
```

Figure A.15: Generative Model of a Transformation.

A.4.3 Hacker's Delight Tasks

The 25 tasks of the Hacker's delight [78] datasets are the following:

1. Turn off the right-most one bit
2. Test whether an unsigned integer is of the form $2^(n-1)$
3. Isolate the right-most one bit
4. Form a mask that identifies right-most one bit and trailing zeros
5. Right propagate right-most one bit
6. Turn on the right-most zero bit in a word
7. Isolate the right-most zero bit
8. Form a mask that identifies trailing zeros
9. Absolute value function
10. Test if the number of leading zeros of two words are the same
11. Test if the number of leading zeros of a word is strictly less than of another work
12. Test if the number of leading zeros of a word is less than of another work
13. Sign Function
14. Floor of average of two integers without overflowing
15. Ceil of average of two integers without overflowing
16. Compute max of two integers
17. Turn off the right-most contiguous string of one bits
18. Determine if an integer is a power of two
19. Exchanging two fields of the same integer according to some input
20. Next higher unsigned number with same number of one bits
21. Cycling through 3 values
22. Compute parity

23. Counting number of bits

24. Round up to next highest power of two

25. Compute higher order half of product of x and y

Reference implementation of those programs were obtained from the examples directory of the stoke repository [119].

A.4.4 Examples of Hacker's Delight Optimization

The first task of the Hacker's Delight corpus consists in turning off the rightmost one bit of a register.

When compiling the code in Listing A.16a, `llvm` generates the code shown in Listing A.16b. A typical example of an equivalent version of the same program obtained by the data-augmentation procedure is shown in Listing A.16c. Listing A.16d contains the optimal version of this program.

Note that such optimization are already feasible using the stoke system of [9].

```
#include <stdint.h>

int32_t p01(int32_t x) {
  int32_t o1 = x - 1;
  return x & o1;
}
```

(a) Source.

```
  pushq %rbp
  movq %rsp, %rbp
  movl %edi, -0x4(%rbp)
  movl -0x4(%rbp), %edi
  subl $0x1, %edi
  movl %edi, -0x8(%rbp)
  movl -0x4(%rbp), %edi
  andl -0x8(%rbp), %edi
  movl %edi, %eax
  popq %rbp
  retq
  nop
  nop
  nop
```

(b) Optimization starting point.

```
  blsrl %edi, %esi
  sets %ch
  xorq %rax, %rax
  sarb $0x2, %ch
  rorw $0x1, %di
  subb $0x3, %dil
  mull %ebp
  subb %ch, %dh
  rcrb $0x1, %dil
  cmovbel %esi, %eax
  retq
```

(c) Alternative equivalent program.

```
  blsrl %edi, %eax
  retq
```

(d) Optimal solution.

Figure A.16: Program at different stage of the optimization.

References

[1] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[2] Joseph B. Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem". In: *Proceedings of the American Mathematical Society* (1956).

[3] David Sontag, Talya Meltzer, Amir Globerson, Yair Weiss, and Tommi Jaakkola. "Tightening LP Relaxations for MAP using Message-Passing". In: *UAI* (2008).

[4] Vladimir Kolmogorov and Martin J. Wainwright. "On the Optimality of Tree-reweighted Max-product Message-passing". In: *CUAI* (2005).

[5] Philipp Krahenbuhl and Vladlen Koltun. "Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials". In: *Neural Information Processing Systems* (2011).

[6] George Bernard Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[7] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2016.

[8] Stephen A. Cook. "The Complexity of Theorem-proving Procedures". In: *ACM Symposium on Theory of Computing* (1971).

[9] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization". In: *SIGPLAN Not.* (2013).

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS Autodiff Workshop* (2017).

[11] Thomas Joy, Alban Desmaison, Thalaiyasingam Ajanthan, Rudy Bunel, Mathieu Salzmann, Pushmeet Kohli, Philip H. S. Torr, and M Pawan Kumar. "Efficient Relaxations for Dense CRFs with Sparse Higher Order Potentials". In: *SIAM journal on imaging sciences* (2019).

[12] N. Siddharth, Brooks Paige, Jan-Willem Van de Meent, Alban Desmaison, Frank Wood, Noah D. Goodman, Pushmeet Kohli, and Philip H.S. Torr. "Learning Disentangled Representations with Semi-Supervised Deep Generative Models". In: *arXiv:1706.00400* (2017).

[13] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, and Philip H. S. Torr. "Playing Doom with SLAM-Augmented Deep Reinforcement Learning". In: *CoRR* (2016).

[14] Alban Desmaison, Rudy Bunel, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. "Efficient Continuous Relaxations for Dense CRF". In: *European Conference on Computer Vision* (2016).

[15] Pradeep Ravikumar and John Lafferty. "Quadratic Programming Relaxations for Metric Labeling and Markov Random Field MAP Estimation". In: *ICML* (2006).

[16] Jon Kleinberg and Éva Tardos. “Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields”. In: *JACM* (2002).

[17] M. Pawan Kumar, Vladimir Kolmogorov, and Philip H. S. Torr. “An Analysis of Convex Relaxations for MAP Estimation”. In: *Neural Information Processing Systems* (2008).

[18] Chandra Chekuri, Sanjeev Khanna, Joseph Naor, and Leonid Zosin. “Approximation algorithms for the metric labeling problem via a new linear programming formulation”. In: *SODA* (2001).

[19] Marshall F. Tappen, Ce Liu, Edward H. Adelson, and William T. Freeman. “Learning Gaussian Conditional Random Fields for Low-Level Vision”. In: *Conference on Computer Vision and Pattern Recognition* (2007).

[20] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[21] Andrew Adams, Jongmin Baek, and Myers Abraham. “Fast High-Dimensional Filtering Using the Permutohedral Lattice”. In: *Eurographics* (2010).

[22] Marguerite Frank and Philip Wolfe. “An algorithm for quadratic programming”. In: *Naval Research Logistics Quarterly* (1956).

[23] Philipp Krahenbuhl and Vladlen Koltun. “Parameter learning and convergent inference for dense random fields”. In: *International Conference on Machine Learning* (2013).

[24] Pierre Baque, Timur Bagautdinov, Francois Fleuret, and Pascal Fua. “Principled Parallel Mean-Field Inference for Discrete Random Fields”. In: *Conference on Computer Vision and Pattern Recognition* (2016).

[25] Vibhav Vineet, Jonathan Warrell, and Philip H. S. Torr. “Filter-Based Mean-Field Inference for Random Fields with Higher-Order Terms and Product Label-Spaces”. In: *International Journal of Computer Vision* (2014).

[26] Lubor Ladicky, Chris Russell, Pushmeet Kohli, and Philip Torr. “Graph cut based inference with co-occurrence statistics”. In: *European Conference on Computer Vision* (2010).

[27] Pushmeet Kohli, Pawan Kumar, and Philip Torr. “P3 & beyond: Solving energies with higher order cliques”. In: *Conference on Computer Vision and Pattern Recognition* (2007).

[28] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *Conference on Computer Vision and Pattern Recognition* (2015).

[29] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan Yuille. “Semantic image segmentation with deep convolutional nets and fully connected CRFs”. In: *International Conference on Learning Representations* (2015).

[30] Alexander Schwing and Raquel Urtasun. “Fully Connected Deep Structured Networks”. In: *CoRR* (2015).

[31] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip H. S. Torr. "Conditional Random Fields as Recurrent Neural Networks". In: *International Conference on Computer Vision* (2015).

[32] Yimeng Zhang and Tsuhan Chen. "Efficient inference for fully-connected CRFs with stationarity". In: *Conference on Computer Vision and Pattern Recognition* (2012).

[33] Peng Wang, Chunhua Shen, and Anton van den Hengel. "Efficient SDP Inference for Fully-Connected CRFs Based on Low-Rank Decomposition". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[34] Michel Goemans and David Williamson. "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming". In: *JACM* (1995).

[35] Simon Lacoste-Julien and Martin Jaggi. "Block-Coordinate Frank-Wolfe Optimization for Structural SVMs". In: *International Conference on Machine Learning* (2013).

[36] Alan L. Yuille and Anand Rangarajan. "The Concave-Convex Procedure (CCCP)". In: *Neural Information Processing Systems* (2002).

[37] Bharath K. Sriperumbudur and Gert R. Lanckriet. "On the convergence of concave-convex procedure". In: *Neural Information Processing Systems* (2009).

[38] M. Pawan Kumar and Daphne Koller. "MAP estimation of semi-metric MRFs via hierarchical graph cuts". In: *UAI* (2009).

[39] Laurent Condat. "Fast projection onto the simplex and the l_1 ball". In: *Mathematical Programming* (2015).

[40] Daniel Scharstein and Richard Szeliski. "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms". In: *International Journal of Computer Vision* (2002).

[41] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* (2010).

[42] Jasper Snoek, Hugo Larochelle, and Ryan Adams. "Practical bayesian optimization of machine learning algorithms". In: *Neural Information Processing Systems* (2012).

[43] Yuri Boykov, Olga Veksler, and Ramin Zabih. "Fast Approximate Energy Minimization via Graph Cuts". In: *PAMI* (2001).

[44] Thalaiyasingam Ajanthan, Alban Desmaison, Rudy Bunel, Mathieu Salzmann, Philip HS Torr, and M Pawan Kumar. "Efficient Linear Programming for Dense CRFs". In: *Conference on Computer Vision and Pattern Recognition* (2017).

[45] M. Pawan Kumar, Vladimir Kolmogorov, and Philip H. S. Torr. "An Analysis of Convex Relaxations for MAP Estimation of Discrete MRFs". In: *Journal of Machine Learning Research* (2009).

[46] Neal Parikh and Stephen Boyd. "Proximal Algorithms". In: *Foundations and Trends in Optimization* (2014).

[47] Xiao Xiao and Donghui Chen. “Multiplicative Iteration for Nonnegative Quadratic Programming”. In: *Numerical Linear Algebra with Applications* (2014).

[48] Francis Bach. “Duality between subgradient and conditional gradient methods”. In: *SIAM Journal on Optimization* (2015).

[49] Hao Zheng, Zhanlei Yang, Wenju Liu, Jizhong Liang, and Yanpeng Li. “Improving deep neural networks using softplus units”. In: *International Joint Conference on Neural Networks* (2015).

[50] Chandra Chekuri, Sanjeev Khanna, Joseph Naor, and Leonid Zosin. “A Linear Programming Formulation and Approximation Algorithms for the Metric Labeling Problem”. In: *SIAM Journal on Discrete Mathematics* (2005).

[51] Vladimir Kolmogorov. “Convergent Tree-Reweighted Message Passing for Energy Minimization”. In: *TPAMI* (2006).

[52] Nikos Komodakis, Nikos Paragios, and Georgios Tziritas. “MRF Energy Minimization and Beyond via Dual Decomposition”. In: *IEEE TPAMI* (2011).

[53] Martin J. Wainwright, Tommi S. Jaakkola, and Alan S. Willsky. “MAP Estimation via Agreement on Trees: Message-passing and Linear Programming”. In: *Transactions on Information Theory* (2005).

[54] Ofer Meshi, Mehrdad Mahdavi, and Alex Schwing. “Smooth and strong: MAP inference with linear convergence”. In: *Neural Information Processing Systems* (2015).

[55] Pradeep Ravikumar, Alekh Agarwal, and Martin J. Wainwright. “Message-passing for Graph-structured Linear Programs: Proximal Projections, Convergence and Rounding Schemes”. In: *International Conference on Machine Learning* (2008).

[56] Anton Osokin, Jean-Baptiste Alayrac, Isabella Lukasewitz, Puneet Dokania, and Simon Lacoste-Julien. “Minding the Gaps for Block Frank-Wolfe Optimization of Structured SVMs”. In: *International Conference on Machine Learning* (2016).

[57] Neel Shah, Vladimir Kolmogorov, and Christoph H. Lampert. “A multi-plane block-coordinate Frank-Wolfe algorithm for training structural SVMs with a costly max-oracle”. In: *Conference on Computer Vision and Pattern Recognition* (2015).

[58] Rahul G. Krishnan, Simon Lacoste-Julien, and David Sontag. “Barrier Frank-Wolfe for marginal inference”. In: *Neural Information Processing Systems* (2015).

[59] Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. “Adaptive Neural Compilation”. In: *Neural Information Processing Systems* (2016).

[60] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. “Conditionally correct superoptimization”. In: *OOPSLA* (2015).

[61] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”. In: *CoRR* (2014).

[62] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. “Neural Random-Access Machines”. In: *CoRR* (2015).

[63] Łukasz Kaiser and Ilya Sutskever. “Neural GPUs learn algorithms”. In: *ICLR* (2015).

[64] Armand Joulin and Tomas Mikolov. “Inferring algorithmic patterns with stack-augmented recurrent nets”. In: *NIPS* (2015).

[65] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. “Learning to transduce with unbounded memory”. In: *Neural Information Processing Systems* (2015).

[66] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. “Adding Gradient Noise Improves Learning for Very Deep Networks”. In: *International Conference on Learning Representations* (2016).

[67] Hava Siegelmann. “Neural programming language”. In: *AAAI* (1994).

[68] Frédéric Gruau, Jean-Yves Ratajszczak, and Gilles Wiber. “A neural compiler”. In: *Theoretical Computer Science* (1995).

[69] João Pedro Neto, Hava Siegelmann, and Félix Costa. “Symbolic processing in neural networks”. In: *Journal of the Brazilian Computer Society* (2003).

[70] Scott Reed and Nando de Freitas. “Neural Programmer-Interpreters”. In: *International Conference on Learning Representations* (2016).

[71] Henry Massalin. “Superoptimizer: a look at the smallest program”. In: *ACM SIGPLAN* (1987).

[72] Ronald J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Machine Learning* (1992).

[73] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. “Learning to learn by gradient descent by gradient descent”. In: *Neural Information Processing Systems* (2016).

[74] Wojciech Zaremba and Ilya Sutskever. “Reinforcement learning neural Turing machines”. In: *CoRR* (2015).

[75] Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. “Learning Simple Algorithms from Examples”. In: *CoRR* (2015).

[76] Diederik P. Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *International Conference on Learning Representations* (2015).

[77] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. “Learning to superoptimize programs”. In: *International Conference on Learning Representations* (2017).

[78] Henry S Warren. *Hacker’s Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[79] Torbjörn Granlund and Richard Kenner. “Eliminating branches using a superoptimizer and the GNU C compiler”. In: *ACM SIGPLAN* (1992).

[80] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. “Scaling up superoptimization”. In: *ACM SIGPLAN* (2016).

[81] Wojciech Zaremba, Karol Kurach, and Rob Fergus. “Learning to Discover Efficient Mathematical Identities”. In: *Neural Information Processing Systems* (2014).

[82] Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. "HC-Search: A Learning Framework for Search-based Structured Prediction." In: *JAIR* (2014).

[83] Ke Li and Jitendra Malik. "Learning to Optimize". In: *CoRR* (2016).

[84] Brookes Paige and Frank Wood. "Inference Networks for Sequential Monte Carlo in Graphical Models". In: *International Conference on Machine Learning* (2016).

[85] Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V Gehler. "The informed sampler: A discriminative approach to Bayesian inference in generative computer vision models". In: *CVIU* (2015).

[86] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. "Picture: A probabilistic programming language for scene perception". In: *Conference on Computer Vision and Pattern Recognition* (2015).

[87] Song-Chun Zhu, Rong Zhang, and Zhuowen Tu. "Integrating bottom-up/top-down for object recognition by data driven Markov chain Monte Carlo". In: *Conference on Computer Vision and Pattern Recognition* (2000).

[88] Tim Salimans, Diederik P Kingma, Max Welling, et al. "Markov chain Monte Carlo and variational inference: Bridging the gap". In: *International Conference on Machine Learning* (2015).

[89] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. "Equation of state calculations by fast computing machines". In: *The journal of chemical physics* (1953).

[90] Peter W Glynn. "Likelihood ratio gradient estimation for stochastic systems". In: *Communications of the ACM* (1990).

[91] Michael C. Fu. "Gradient Estimation". In: *Handbooks in Operations Research and Management Science* (2006).

[92] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. "Gradient estimation using stochastic computation graphs". In: *Neural Information Processing Systems* (2015).

[93] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *Neural Information Processing Systems* (2011).

[94] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of Loop-free Programs". In: *PLDI* (2011).

[95] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. "Oracle-guided component-based program synthesis". In: *International Conference on Software Engineering* (2010).

[96] Monique Guignard and Siwhan Kim. "Lagrangean decomposition: A model yielding stronger lagrangean bounds". In: *Mathematical Programming* (1987).

[97] Martin Jaggi. "Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization". In: *International Conference on Machine Learning* (2013).

[98] URL: `http://www.gurobi.com/`.

[99] URL: `https://www.cplex.com`.

[100] URL: `https://www.gnu.org/software/glpk/glpk.html`.

[101] URL: https://developers.google.com/optimization/lp/glop.

[102] URL: https://www.coin-or.org/Clp/.

[103] Neal Parikh and Stephen Boyd. "Block splitting for distributed optimization". In: *Mathematical Programming Computation* (2014).

[104] https://foges.github.io/pogs/.

[105] Jack Edmonds. "Matroids and the Greedy Algorithm". In: *Mathematical Programming* (1971).

[106] Vladimir Kolmogorov and Carsten Rother. "Minimizing Nonsubmodular Functions with Graph Cuts-A Review". In: *IEEE TPAMI* (2007).

[107] Marshall L. Fisher. "The Lagrangian Relaxation Method for Solving Integer Programming Problems". In: *Management Science* (1981).

[108] Monique Guignard and Moshe B. Rosenwein. "An Application of Lagrangean Decomposition to the Resource-Constrained Minimum Weighted Arborescence Problem". In: *Networks* (1990).

[109] Susara A. van den Heever, Ignacio E. Grossmann, Sriram Vasantharajan, and Krisanne Edwards. "A Lagrangean Decomposition Heuristic for the Design and Planning of Offshore Hydrocarbon Field Infrastructures with Complex Economic Objectives". In: *Industrial & Engineering Chemistry Research* (2001).

[110] Endre Boros and Peter L. Hammer. "Pseudo-Boolean optimization". In: *Discrete Applied Mathematics* (2002).

[111] Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. "A Unified View of Piecewise Linear Neural Network Verification". In: *Neural Information Processing Systems* (2018).

[112] Ruediger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *Automated Technology for Verification and Analysis* (2017).

[113] Dimitri P. Bertsekas. *Convex Optimization Theory*. Athena Scientific, 2009.

[114] Simon Lacoste-Julien, Martin Jaggi, Mark Schmidt, and Patrick Pletscher. "On the Global Linear Convergence of Frank-Wolfe Optimization Variants". In: *Neural Information Processing Systems* (2015).

[115] Alex Krizhevsky. "Learning multiple layers of features from tiny images". In: *Technical Report* (2009).

[116] J. Zico Kolter and Eric Wong. "Provable defenses against adversarial examples via the convex outer adversarial polytope". In: *International Conference on Machine Learning* (2017).

[117] Albert-László Barabasi and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* (1999).

[118] Paul Erdös and Alfréd Rényi. "On Random Graphs I". In: *Publicationes Mathematicae* (1959).

[119] *Stoke*. 2016. URL: https://github.com/StanfordPL/stoke.

[120] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. "Learning Combinatorial Optimization Algorithms over Graphs". In: *Neural Information Processing Systems* (2017).

[121] Hanif D. Sherali and Warren P. Adams. “A Hierarchy of Relaxation Between the Continuous and Convex Hull Representations”. In: *SIAM Journal on Discrete Mathematics* (1990).

[122] Jean-Bernard Lasserre. “Optimality conditions and LMI relaxations for 0 1 programs”. In: *Tech report* (2000).

[123] Yair Bartal. “On Approximating Arbitrary Metrices by Tree Metrics”. In: *STOC* (1998).

[124] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. “A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics”. In: *STOC* (2003).

[125] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.