Solving Mixed Integer Programs Using Neural Networks

Vinod Nair*†1, Sergey Bartunov*1, Felix Gimeno*1, Ingrid von Glehn*1, Pawel Lichocki*2, Ivan Lobov*1, Brendan O'Donoghue*1, Nicolas Sonnerat*1, Christian Tjandraatmadja*2, Pengming Wang*1, Ravichandra Addanki[1], Tharindi Hapuarachchi[1], Thomas Keck[1], James Keeling[1], Pushmeet Kohli[1], Ira Ktena[1], Yujia Li[1], Oriol Vinyals[1], Yori Zwols[1]

[1]DeepMind, [2]Google Research

Mixed Integer Programming (MIP) solvers rely on an array of sophisticated heuristics developed with decades of research to solve large-scale MIP instances encountered in practice. Machine learning offers to automatically construct better heuristics from data by exploiting shared structure among instances in the data. This paper applies learning to the two key sub-tasks of a MIP solver, generating a high-quality joint variable assignment, and bounding the gap in objective value between that assignment and an optimal one. Our approach constructs two corresponding neural network-based components, *Neural Diving* and *Neural Branching*, to use in a base MIP solver such as SCIP. Neural Diving learns a deep neural network to generate multiple partial assignments for its integer variables, and the resulting smaller MIPs for un-assigned variables are solved with SCIP to construct high quality joint assignments. Neural Branching learns a deep neural network to make variable selection decisions in branch-and-bound to bound the objective value gap with a small tree. This is done by imitating a new variant of Full Strong Branching we propose that scales to large instances using GPUs. We evaluate our approach on six diverse real-world datasets, including two Google production datasets and MIPLIB, by training separate neural networks on each. Most instances in all the datasets combined have $10^3 - 10^6$ variables and constraints after presolve, which is significantly larger than previous learning approaches. Comparing solvers with respect to primal-dual gap averaged over a held-out set of instances, the learning-augmented SCIP is 2× to 10× better on all datasets except one on which it is $10^5\times$ better, at large time limits. To the best of our knowledge, ours is the first learning approach to demonstrate such large improvements over SCIP on both large-scale real-world application datasets and MIPLIB.

Key words: deep learning, mixed integer programming, discrete optimization, graph networks

History: First version December 2020.

1. Introduction

Mixed integer programs (MIPs) are a class of NP-hard problems where the goal is to minimize a linear objective subject to linear constraints, with some or all of the variables constrained to be integer-valued (Wolsey 1998, Karp 1972). They have enjoyed widespread adoption in a broad range

*Equal contributors, listed alphabetically by last name after the lead author.

†Corresponding author, email: vinair@google.com.

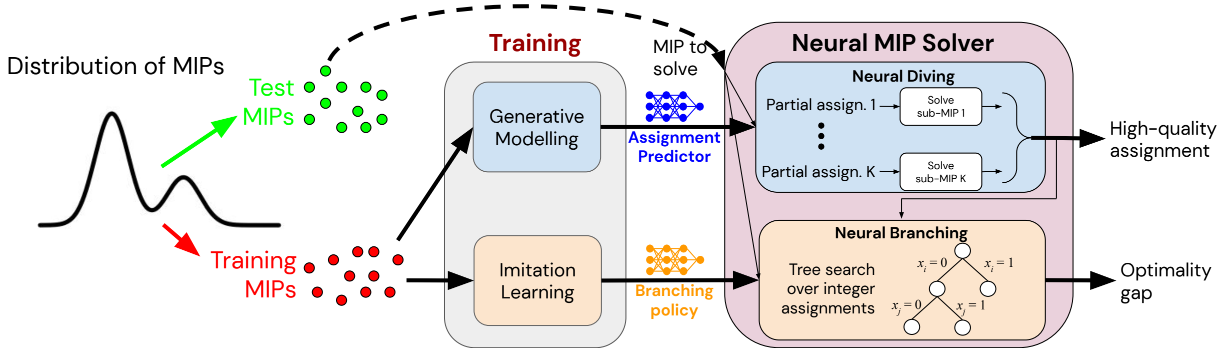


Figure 1 Our approach constructs two neural network-based components to use in a MIP solver, *Neural Diving* and *Neural Branching*, and combine them to produce a *Neural MIP solver* customized to a given MIP dataset.

of applications such as capacity planning, resource allocation, bin packing, etc. (Taha 2014, Jünger et al. 2009, Sierksma and Zwols 2015). Significant research and engineering effort has gone into developing practical solvers, such as SCIP (Gamrath et al. 2020), CPLEX (IBM ILOG CPLEX 2019), Gurobi (Gurobi Optimization 2020), and Xpress (FICO Xpress 2020). These solvers use sophisticated heuristics to control the search process for solving a MIP. A solver's performance on a given application depends crucially on the strength of its heuristics.

In this paper we show that machine learning can be used to automatically construct effective heuristics from a dataset of MIP instances. A compelling use case for this arises often in practice where an application requires solving a large set of instances of the same high-level semantic problem with different problem parameters. Examples of such "homogenous" datasets in this paper are 1) optimizing the choice of power plants on an electric grid to meet demand (O'Neill 2017), where grid topology remains the same while demand, renewable generation, etc. vary across instances, and 2) solving a packing problem in Google's production system where the semantics of "items" and "bins" to be packed remain mostly the same but their sizes fluctuate across instances. Even a "heterogeneous" dataset that combines many semantically different problems, such as MIPLIB 2017 (Gleixner et al. 2019), can have structure across instances that can be used to learn better heuristics, as we shall show. Off-the-shelf MIP solvers cannot automatically construct heuristics to exploit such structure. In challenging applications users may rely on an expert to hand-design such heuristics, or forego potentially large performance improvements. Machine learning offers the possibility of large improvements without needing application-specific expertise.

We demonstrate that machine learning can construct heuristics that significantly outperform the classical ones used in a MIP solver, specifically the state-of-the-art non-commercial solver SCIP 7.0.1 (Gamrath et al. 2020), on a given dataset. Our approach applies learning to the two key subtasks of a MIP solver: a) output an assignment of values to all variables that satisfy the constraints (if such an assignment exists), and b) prove a bound for the gap in objective value between that assignment and an optimal one. They define the main components of our approach, *Neural Diving* and *Neural Branching* (see figure 1).

Neural Diving: This component finds high quality joint variable assignments. It is an instance of a *primal heuristic* (Berthold 2006), a class of search heuristics that have been identified as key to effective MIP solvers (Berthold 2013). We train a deep neural network to produce multiple partial assignments of the integer variables of the input MIP. The remaining unassigned variables define smaller 'sub-MIPs', which are solved using an off-the-shelf MIP solver (e.g., SCIP) to complete the assignments. The sub-MIPs can be solved in parallel if the compute budget allows. The model is trained to give higher probability to feasible assignments that have better objective values, using training examples collected offline with an off-the-shelf solver. It learns on all available feasible assignments instead of only the optimal ones, and does not necessarily require optimal assignments, which can be expensive to collect.

Neural Branching: This component is mainly used to bound the gap between the objective value of the best assignment and an optimal one. MIP solvers use a form of tree search over integer variables called *branch-and-bound* (Land and Doig 1960) (see section 2), which progressively tightens the bound and helps find feasible assignments. The choice of the variable to branch on at a given node is a key factor in determining search efficiency (Achterberg et al. 2005, Glankwamdee and Linderoth 2011, Schubert 2017, Yang et al. 2019). We train a deep neural network policy to imitate choices made by an expert policy. The imitation target is a well-known heuristic called Full Strong Branching (FSB), which has been empirically shown to produce small search trees (Achterberg et al. 2005). While it is often too computationally expensive for practical MIP solving, it can still be used to generate imitation learning data offline as a slow and expensive one-time computation. Once trained, the neural network is able to approximate the expert at test time at a fraction of the computational cost. A CPU-based implementation of FSB can be too expensive on large-scale MIPs even for offline data generation. We develop a variant of FSB using the alternating directions method of multipliers (ADMM) (Boyd et al. 2011) that scales to large-scale MIPs by performing the required computation in a batch manner on GPU.

We evaluate our approach on six datasets. Five of them are from individual real-world applications, including two from Google's production systems. The sixth one is MIPLIB (Gleixner et al. 2019), which is a heterogeneous dataset and a standard benchmark. Most of the combined set of MIPs from all datasets have 10^3–10^6 variables and constraints after presolve (see figure 4 in section 4), which is significantly larger than earlier works (Gasse et al. 2019, Ding et al. 2020). Once Neural Diving and Neural Branching models are trained on a given dataset, they are integrated into SCIP to form a "Neural MIP solver" specifically for that dataset. The baseline is SCIP with its

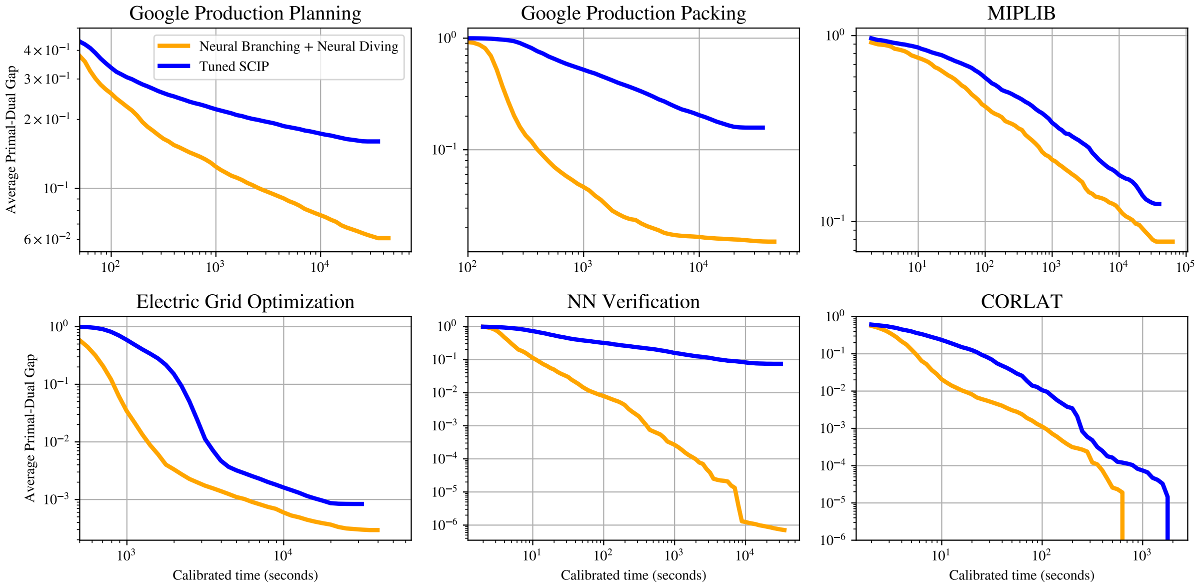


Figure 2 Main result of the paper: our approach, Neural Branching + Neural Diving, significantly outperforms SCIP with respect to the primal-dual gap (Berthold 2006), averaged on held-out instances, as a function of running time (referred to as calibrated time, see section 5) for all six datasets that we have tested. Note the log scale on both axes.

emphasis parameters tuned by grid search on each dataset separately, which we refer to as Tuned SCIP. Comparing the *primal-dual gap* (Berthold 2006) averaged over a held-out set of instances, the Neural MIP solver is about 2× to 10× better than Tuned SCIP on all datasets except the Neural Network Verification dataset, on which it is $10^5\times$ better, at large time limits (see figure 2). To the best of our knowledge, this is the first work to demonstrate such large improvements over SCIP on both large-scale real-world application datasets and MIPLIB using machine learning.

Tuned SCIP is the baseline we compare to since we use SCIP as the base solver for integrating learned heuristics. As a base solver SCIP provides a) extensive access to its internal state for integrating learned models, and b) permissive licensing that enables large-scale evaluation by running a large number of solver instances in parallel. We do not have access to commercial solvers with these features, which makes a fair comparison to them infeasible. We have done a partial comparison of Gurobi versus Neural Diving alone, with Gurobi as its sub-MIP solver, on two datasets (see section 6). Comparing the *primal gap* (Berthold 2006) averaged over a held-out set of instances, Neural Diving with parallel sub-MIP solving reaches 1% average primal gap in 3× and 3.6× less time than Gurobi on the two datasets. We have also applied a modified version of Neural Diving to a subset of 'open' instances in MIPLIB to find three new best known assignments, beating commercial solvers.

Several earlier works have focused on learning primal heuristics (Khalil et al. 2017b, Ding et al. 2020, Hendel 2018, Hottung and Tierney 2019, Xavier et al. 2020, Song et al. 2020, Addanki et al. 2020). Unlike them, Neural Diving poses the problem of predicting variable assignments as a generative modelling problem, which provides a principled way to learn on all available feasible assignments and also to generate partial assignments at test time. Several works have also looked at learning a branching policy (He et al. 2014, Khalil et al. 2016, Alvarez et al. 2017, Balcan et al. 2018, Gasse et al. 2019, Yang et al. 2020, Zarpellon et al. 2020, Gupta et al. 2020). Many of these focus specifically on learning to imitate FSB as we do (Khalil et al. 2016, Alvarez et al. 2017, Gasse et al. 2019, Gupta et al. 2020). Unlike them, Neural Branching uses a more scalable approach to computing the target policy using GPUs, which allows it to generate more imitation data from larger instances in the same time limit than a CPU-based FSB implementation. We also go beyond earlier works that study learning individual heuristics in isolation by combining a learned primal heuristic and a learned branching policy in a solver to achieve significantly better performance on large-scale real-world application datasets and MIPLIB.

1.1. Contributions

1. We propose Neural Diving (section 6), a new learning-based approach to generating high-quality joint variable assignments for a MIP. On homogeneous datasets, Neural Diving achieves an average primal gap of 1% on held out instances 3-10× faster than Tuned SCIP. On one dataset Tuned SCIP does not reach 1% average primal gap within the time limit, while Neural Diving does.

2. We propose Neural Branching which learns a branching policy to use in the branch-and-bound algorithm by imitating a new scalable expert policy based on ADMM (section 7). On two of the datasets used in our evaluation for which FSB is slow due to instance sizes (e.g., with $> 10^5$ variables) or high per-iteration times, the ADMM expert generates 1.4× and 12× more training data in the same running time. The learned policy significantly outperforms SCIP's branching heuristic on four of the six datasets with 2-20× better average *dual gap* (Berthold 2006) on held out instances at large time limits, and has comparable performance on the other two.

3. We combine Neural Diving and Neural Branching (section 8) to attain significantly better running time performance than SCIP across all datasets, as measured by the average primal-dual gap on held out instances.

In addition, we plan to open source two of the datasets, for the applications of Neural Network Verification and Electric Grid Optimization, which we hope will help further research on new learning techniques for MIPs.

2. Integer programming background

In this section we present some basic integer programming concepts and techniques relevant for the paper. A mixed integer linear program has the form

$$\begin{array}{ll} \text{minimize} & c^\top x \\ \text{subject to} & Ax \leq b \\ & l \leq x \leq u \\ & x_i \in \mathbb{Z}, \quad i \in \mathcal{I} \end{array} \tag{1}$$

over variables $x \in \mathbb{R}^n$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, $l \in (\mathbb{R} \cup \{-\infty\})^n$, and $u \in (\mathbb{R} \cup \{\infty\})^n$ are given data, and $\mathcal{I} \subseteq \{1, \dots, n\}$ refers to the index set of integer variables. We allow l and u to take infinite values to indicate that the associated variable has no lower or upper bound respectively, for those indices. A (complete) *assignment* is any point $x \in \mathbb{R}^n$. A partial assignment is when we have fixed some, but not, all of the variable values. A *feasible* assignment is an assignment that satisfies all the constraints in problem (1), and an *optimal* assignment, or a *solution*, is a feasible assignment that also minimizes the objective (Boyd and Vandenberghe 2004, §1.1).

2.1. Linear programming relaxation

If we remove the integer constraints in problem (1) then it becomes a linear program (LP), which is convex and can be solved efficiently (Boyd and Vandenberghe 2004). The optimal value of the relaxed problem is guaranteed to be a lower bound for the original problem, since removing constraints can only expand the feasible set. If the optimal solution to the LP satisfies the integral constraints then it is guaranteed to be optimal for the original problem. We shall refer to any lower bound (in our case found via linear programming) as a *dual bound.*

2.2. Branch-and-bound

A common procedure to solve MIPs is to recursively build a search tree, with partial integer assignments at each node, and use the information gathered at each node to eventually converge on an optimal (or close to optimal) assignment (Land and Doig 1960, Lawler and Wood 1966). At every step we must choose a leaf node from which to 'branch'. At this node we can solve the LP relaxation, where we constrain the ranges of the fixed variables at that node to their assigned values. This gives us a valid lower bound on the true objective value of any further child nodes from this node. If this bound is larger than a known feasible assignment, then we can safely prune this part of the search tree since no optima for the original problem can exist in the subtree from that node. If we decide to expand this node, then we must choose a variable to branch on from the set of unfixed variables at that node. Once a variable is selected, we take a branching step, which adds two child nodes to the current node. One node has the domain of the selected variable constrained to be greater or equal to the ceiling of its LP relaxation value at the parent node. The other node has the selected variable's domain constrained to be less than or equal to the floor of its LP relaxation value. The tree is updated, and the procedure begins again. This algorithm is referred to as *branch-and-bound*. Linear programming is a main workhorse of this procedure, both to derive the dual bounds at every node and to decide the branching variable for some of the more sophisticated branching heuristics. In theory the size of the search tree can be exponential in the input size of the problem, but in many cases the search trees can be small and it is an area of established and active research to come up with node selection and variable selection heuristics that keep the tree as small as possible.

2.3. Primal heuristics

A primal heuristic is a method that attempts to find a *feasible*, but not necessarily optimal, variable assignment (Berthold 2006). Any such feasible assignment provides a guaranteed upper bound on the optimal value of the MIP. Any such bound found at any point during a MIP solve is called a *primal bound*. Primal heuristics can be run independently of branch-and-bound, but they can also be run within a branch-and-bound tree and attempt to find a feasible assignment of the unfixed variables from a given node in the search tree. Better primal heuristics that produce lower primal bounds allow the branch-and-bound procedure to prune more of the tree. Simple rounding, where the fractional variables are rounded to integer values (possibly randomly) is an example of a primal heuristic. Another case is *diving* which attempts to find a feasible solution by exploring the search tree from a given node in a depth-first manner (Berthold 2006, Eckstein and Nediak 2007).

2.4. Primal-dual gap

When running branch-and-bound we keep track of the *global* primal bound (the minimum objective value of any feasible assignment) and the *global* dual bound (the minimum dual bound across all leaves of the branch-and-bound tree). We can combine these to define a sub-optimality gap

$$\text{gap} = \text{global primal bound} - \text{global dual bound}.$$

The gap is always nonnegative by construction, and if it is zero then we have solved the problem, the feasible point that corresponds to the primal bound is optimal and the dual bound is a certificate of optimality. In practice we terminate branch-and-bound when the *relative* gap (*i.e.*, normalized in some way, see §5) is below some application-dependent quantity, and produce the best found primal solution as the approximately optimal solution.

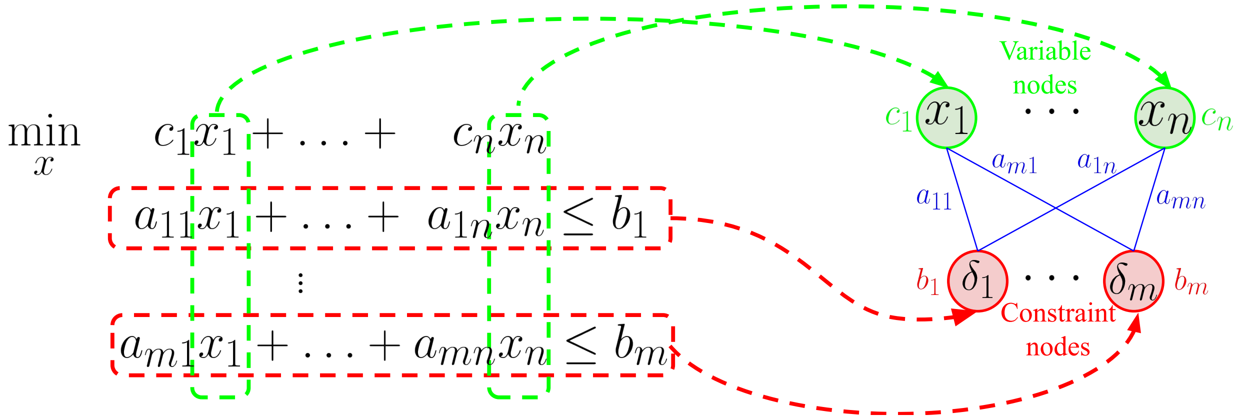

$\min_x$
$c_1x_1 + \ldots + c_nx_n$
$a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1$
$\vdots$
$a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m$
Variable nodes
c_1 x_1 $\cdots$ x_n c_n
a_{11} a_{m1} a_{1n} a_{mn}
b_1 δ_1 $\cdots$ δ_m b_m
Constraint nodes

Figure 3 Bipartite graph representation of a MIP used as the input to a neural network. The set of n variables $\{x_1, \ldots, x_n\}$ and the set of m constraints $\{\delta_1, \ldots, \delta_m\}$ form the two sets of nodes of the bipartite graph. The coefficients are encoded as features of the nodes and edges.

3. MIP Representation and Neural Network Architecture

We describe how a MIP is represented as an input to a neural network, and the architecture we use to learn models for both Neural Diving and Neural Branching. The key deep learning architecture we use is a form of *graph neural network* (Scarselli et al. (2009), see survey by Battaglia et al. (2018)), specifically a *graph convolutional network* (GCN) (Kipf and Welling 2016).

3.1. Representing a MIP as an Input to a Neural Network

We use a bipartite graph representation of a MIP, as done in Gasse et al. (2019). Equation (1) can be used to define a bipartite graph where one set of n nodes in the graph correspond to the n variables being optimized, and the other set of m nodes correspond to the m constraints; see figure 3. The edge is present between a variable node and a constraint node if the corresponding variable appears in that constraint, and so the number of edges correspond to the number of non-zeros in the constraint matrix. The objective coefficients $\{c_1, \ldots, c_n\}$, the constraint bounds $\{b_1, \ldots, b_m\}$, and the non-zero coefficients of the constraint matrix are encoded as scalar "features" that annotate the corresponding variable nodes, constraint nodes, and edges, respectively. The variable type (continuous or integer) can also be encoded as a (categorical) feature of the variable nodes. This defines a lossless representation of the MIP that can be used as an input to a graph neural network. A similar representation is used for MIPs by Ding et al. (2020), and for Boolean Satisfiability instances in an earlier work by Selsam et al. (2019). Both nodes and edges can be annotated by multi-dimensional feature vectors that encode additional information about the MIP that can be useful for learning (e.g., the solution of the LP relaxation as additional variable node features). We use the code provided by Gasse et al. (2019) to compute the same set of features using SCIP.[1]

3.2. Neural Network Architecture

We describe here the common aspects of the network architecture used by both Neural Diving and Neural Branching. Those aspects that differ between the two, e.g., the outputs and the loss functions, are given in their corresponding sections 6 and 7.

Given the bipartite graph representation of a MIP, we use a GCN to learn models for both Neural Diving and Neural Branching. Let the input to the GCN be a graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{A})$ defined by the set of nodes $\mathcal{V}$, the set of edges $\mathcal{E}$, and the graph adjacency matrix $\mathcal{A}$. In the case of MIP bipartite graphs, $\mathcal{V}$ is the union of n variable nodes and m constraint nodes, of size $N := |\mathcal{V}| = n + m$. $\mathcal{A}$ is an $N \times N$ binary matrix with $\mathcal{A}_{ij} = 1$ if nodes indexed by i and j are connected by an edge, 0 otherwise, and $\mathcal{A}_{ii} = 1$ for all i. Each node has a D-dimensional feature vector, denoted by $u_i \in \mathbb{R}^D$ for the i^{th} node. Let $U \in \mathbb{R}^{N \times D}$ be the matrix containing feature vectors of all nodes as rows, *i.e.*, the i^{th} row is u_i. A single-layer GCN learns to compute an H-dimensional continuous vector representation for each node of the input graph, referred to as a *node embedding*. Let $z_i \in \mathbb{R}^H$ be the node embedding computed by the GCN for the i^{th} node, and $Z \in \mathbb{R}^{N \times H}$ be the matrix containing all node embeddings as rows. We define the function computing Z as follows:

$$Z = \mathcal{A} f_\theta(U), \tag{2}$$

where $f_\theta : \mathbb{R}^D \to \mathbb{R}^H$ is a Multi-Layer Perceptron (MLP) (Goodfellow et al. 2016) with learnable parameters $\theta \in \Theta$. (Here we have generalized f_θ from a linear mapping followed by a fixed nonlinearity in the standard GCN by Kipf and Welling (2016) to an MLP.) We overload the notation to allow f_θ to operate on N nodes simultaneously, *i.e.*, $f_\theta(U)$ denotes applying the MLP to each row of U to compute the corresponding row of its output matrix of size $N \times H$. Multiplying by $\mathcal{A}$ combines the MLP outputs of the i^{th} node's neighbors to compute its node embedding. The above definition can be generalized to L layers as follows:

$$Z^{(0)} = U \tag{3}$$

$$Z^{(l+1)} = \mathcal{A} f_{\theta(l)}(Z^{(l)}), \quad l = 0, \ldots, L-1, \tag{4}$$

where $Z^{(l)}$ and $f_{\theta(l)}()$ denote the node embeddings and the MLP, respectively, for the l^{th} layer. The L^{th} layer's node embeddings can be used as input to another MLP that compute the outputs for

[1] For the list of features, see `https://papers.nips.cc/paper/2019/file/d14c2267d848abeb81fd590f371d39bd-Supplemental.zip`

the final prediction task, as shown in sections 6 and 7. For each variable x_d we further denote the corresponding node embedding from the last layer as v_d.

Two key properties of the bipartite graph representation of the MIP and the GCN architecture are: 1) the network output is invariant to permutations of variables and constraints, and 2) the network can be applied to MIPs of different sizes using the same set of parameters. Both of these are important because there may not be any canonical ordering for variables and constraints, and different instances within the same application can have different number of variables and constraints.

3.3. Improvements to the Architecture

We describe changes to the above architecture, as well as from previous work, which gave performance improvements.

1. We modify the MIP bipartite graph's adjacency matrix $\mathcal{A}$ to contain coefficients from the MIP's constraint matrix A, instead of binary values indicating the presence of edges. Specifically, for the i^{th} variable and j^{th} constraint, their two corresponding entries in $\mathcal{A}$ is set to a_{ji} where a_{ji} is the coefficient at row j and column i of A. This results in edges weighted by the entries of A.

2. We extend the node embeddings for layer $l+1$ by concatenating the node embeddings from layer l. Specifically, we now define the embedding for layer $l+1$ to be $\widetilde{Z}^{(l+1)} = (Z^{(l+1)}, \widetilde{Z}^{(l)})$, *i.e.*, the concatenation of the matrices row-wise, with $\widetilde{Z}^{(0)} = Z^0$. This is a form of skip connection (He et al. 2016) and also similar to the jumping knowledge networks architecture (Xu et al. 2018).

3. We apply layer norm (Lei Ba et al. 2016) at the output of each layer, so that $Z^{(l+1)} = \text{LayerNorm}\left(\mathcal{A} f_{\theta(l)}(Z^{(l)})\right)$.

We have explored alternative architectures which use embeddings for both nodes and edges with separate MLPs to compute them. When using such networks with a high-dimensional edge embedding at every layer, their memory usage can become much higher than that of GCNs, which only need the adjacency matrix, and may not fit the GPU memory unless the number of layers is reduced at the cost of accuracy. GCNs are a better fit for our goal of scaling to large-scale MIPs.

4. Datasets

We summarize the details of our datasets in table 1. All datasets except MIPLIB are application-specific, i.e., they contain instances from only a single application. We include MIPLIB in our evaluation even though it contains instances from many applications. This is not the setting in which we expect learning will provide the most benefit. Nevertheless, it is a well-established benchmark for evaluating solvers, and the results on it can provide insight on the performance of learning

Table 1 MIP datasets used for evaluation. See additional details in section 12.6

Name	**Description**
CORLAT	Public dataset related to wildlife management (Conrad et al. 2007, Gomes et al. 2008).
NN Verification	Verifying whether a neural network is robust to input perturbations can be posed as a MIP (Cheng et al. 2017, Tjeng et al. 2019). Each input on which to verify the network gives rise to a different MIP. In this dataset, a convolutional neural network is verified on each image in the MNIST dataset, giving rise to a corresponding dataset of MIPs.
Google Production Packing	A packing optimization problem solved in a Google production system.
Google Production Planning	A planning optimization problem solved in a Google production system.
Electric Grid Optimization	Electric grid operators optimize the choice of power generators to use at different time intervals during a day to meet electricity demand by solving a MIP. This dataset is constructed for one of the largest grid operators in the US, PJM, using publicly available data about generators and demand, and the MIP formulation in (Knueven et al. 2018).
MIPLIB	Heterogeneous dataset containing 'hard' instances of MIPs across many different application areas that is used as a long-standing standard benchmark for MIP solvers (Gleixner et al. 2019). We use instances from both the 2010 and 2017 versions of MIPLIB.

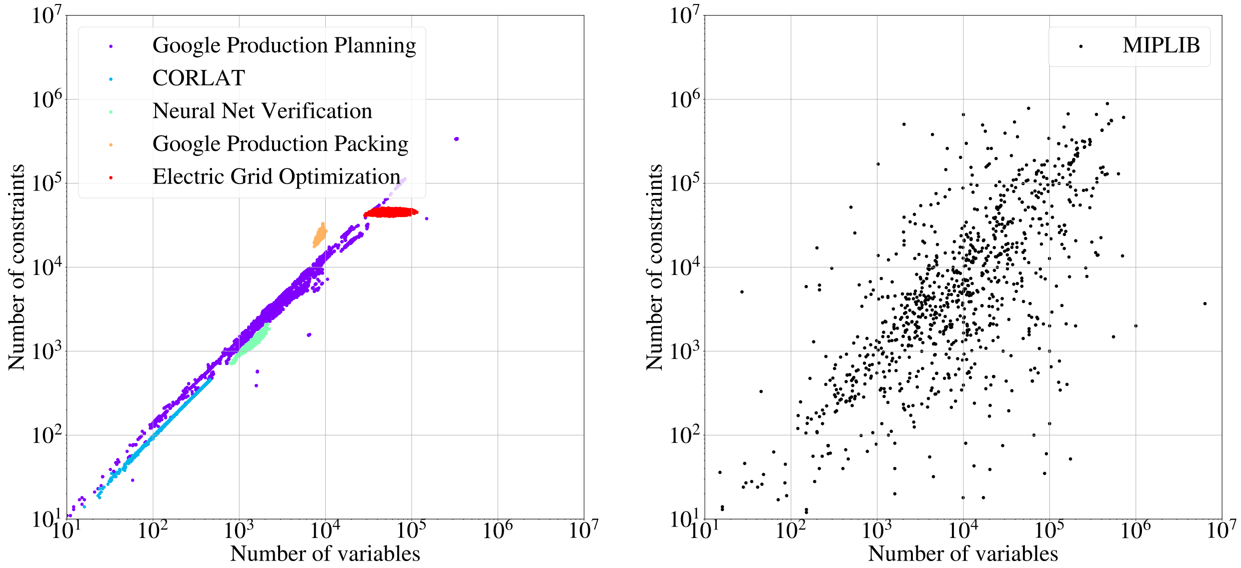


Figure 4 Number of variables and constraints after presolve using SCIP 7.0.1 for the datasets used in our evaluation. MIPLIB is shown separately (right) to reduce clutter.

approaches in an unfavorable setting. The MIP sizes for the various datasets after presolving are shown in figure 4.

For all datasets except MIPLIB, we define the training, validation, and test sets by randomly splitting the instances into disjoint subsets with 70%, 15%, and 15% of the instances, respectively. For MIPLIB, we use instances from the MIPLIB 2017 Benchmark Set as the test set, since that is the set on which solvers are evaluated. The MIPLIB 2017 Collection Set and the MIPLIB 2010 set are used as the training and validation sets, respectively, after removing any overlapping instances with the MIPLIB 2017 Benchmark Set. For each dataset, the training set is used to learn models for that dataset, the validation set is used to tune the learning hyperparameters and SCIP's metaparameters, and the test set is used to report evaluation results. Additional details are given in the Appendix, see section 12.6.

5. Evaluation

We shall evaluate Neural Diving and Neural Branching individually first, and then jointly in the sequel. In all cases we evaluate on a test set of MIPs disjoint from the training set to measure generalization to unseen instances. We present results using *gap plots* and *survival plots*:

Gap plots: Neural Diving is evaluated with respect to the *primal gap* $\gamma_p(t)$ (Berthold 2013) as a function of solving time t. If no feasible assignment is known then the gap is defined to be 1, otherwise denote by $p(t)$ the primal bound at time t and denote by $p^\star$ the best known primal bound (possibly precomputed), the gap is defined as

$$\gamma_p(t) = \begin{cases} 1 & p(t) \cdot p^\star < 0 \\ \frac{p(t) - p^\star}{\max\{|p(t)|, |p^\star|, \epsilon\}} & \text{otherwise.} \end{cases} \tag{5}$$

We use $\epsilon = 10^{-12}$ to avoid division by 0. To evaluate Neural Branching we use a dual gap, which is defined analogously

$$\gamma_d(t) = \begin{cases} 1 & d(t) \cdot p^\star < 0 \\ \frac{p^\star - d(t)}{\max\{|d(t)|, |p^\star|, \epsilon\}} & \text{otherwise,} \end{cases} \tag{6}$$

where $d(t)$ is the global dual bound at time t. When evaluating Neural Diving and Neural Branching jointly, we use the *primal-dual gap*

$$\gamma_{pd}(t) = \begin{cases} 1 & d(t) \cdot p(t) < 0 \\ \frac{p(t) - d(t)}{\max\{|d(t)|, |p(t)|, \epsilon\}} & \text{otherwise.} \end{cases} \tag{7}$$

In each case, we plot the average gap for test MIPs as a function of running time. We pre-compute $p^\star$ for a MIP by running SCIP on it with default parameters and a time limit of 24 hours. Any suboptimality in $p^\star$ will affect all solvers being evaluated and relative comparisons using the gaps are therefore still valid. If during the solve we find a better primal bound then we replace $p^\star$ with the new value and recompute gaps at all previous times.

Survival plots: A survival plot shows the fraction of test set MIPs solved (to the target primal-dual gap) as a function of running time. Applications may specify a target optimality gap higher than 0 as a termination condition. In such a case improving the gap below the target does not improve the solve performance. The survival plot reflects the effect of the target gap, while the gap plots do not.

Calibrated time: The total evaluation workload across all datasets and comparisons requires more than 160,000 MIP solves and nearly a million CPU and GPU hours. To meet the compute requirements, we use a shared, heterogeneous compute cluster. Accurate running time measurement on such a cluster is difficult because the tasks may be scheduled on machines with different hardware, and interference from other unrelated tasks on the same machine increases the variance of solve times. To improve accuracy, for each solve task, we periodically solve a small *calibration MIP* on a different thread from the solve task on the same machine. We use an estimate of the number of calibration MIP solves during the solve task on the same machine to measure time, which is significantly less sensitive to hardware heterogeneity and interference. This quantity is then converted into a time value using the calibration MIP's solve time on a reference machine. Section 12.7 gives the details. Results for four instances from MIPLIB show a 1.5× to 30× reduction in the coefficient of variation of time measurements compared to measuring wall clock time.

Tuned SCIP: The main baseline we compare against is SCIP 7.0.1 with its parameters tuned for each test dataset. SCIP has emphasis "meta-parameters" for presolving, primal heuristics, and cuts, each of which has four possible settings (default, off, aggressive, fast). For each dataset, we use exhaustive search over the $4^3 = 64$ combinations to find the setting that produces the best average primal-dual gap on a subset of 200 validation MIPs in a 3 hour time limit. We call this baseline *Tuned SCIP*.

Performance variability with respect to random seed: To account for the performance variability of MIP solvers (Lodi and Tramontani 2014) with respect to changes to a MIP that leave the problem unchanged, such as permuting rows and columns of the constraint matrix, we vary the random seed parameter used by SCIP. Specifically, we set the SCIP parameter *randomization/permutevars* to *True*, and assign the parameters *randomization/permutationseed* and *randomization/randomseedshift* both to a give seed value. The seed is set to the values $\{1, 2, 3, 4, 5\}$ for each instance. The evaluation results reported for Neural Diving, Neural Branching, and their combination are computed by aggregating over all the instance-seed pairs produced using the test set instances.

6. Neural Diving

In this section we describe our approach to learning a diving-style primal heuristic that produces high quality assignments to MIPs from a given instance distribution. The idea is to train a *generative model* over assignments to a MIP's integer variables from which partial assignments can be sampled. We use SCIP to obtain high-quality assignments (not necessarily optimal) as the target labels for the training set of MIPs. Once trained on this data, the model predicts values for integer variables on unseen instances from the same problem distribution. The uncertainty represented in the model predictions is used to define partial assignments to the original MIP that fix a large fraction of the integer variables. These substantially smaller sub-MIPs can then be solved quickly using SCIP, yielding high quality feasible assignments.

Diving refers to a set of primal heuristics that explore the branch-and-bound tree in a depth-first manner by sequentially fixing integer variables until a leaf node is reached or the assignment is deemed infeasible (Berthold 2006, Eckstein and Nediak 2007). There are a few major differences between regular diving and what we describe here. First, diving can be started from any node, but in this work we focus on diving only from the root node, though in principle it could be performed from other nodes. Secondly, diving typically descends all the way to a leaf node, but in this case we descend only partially and then use a MIP solver to solve the remaining sub-MIP. For this reason one could reasonably call our method a hybrid of diving and neighborhood search (Mladenović and Hansen 1997, Hansen et al. 2010), but for brevity we refer to it simply as Neural Diving. Finally, diving usually proceeds in an iterative manner where decisions are made sequentially and the linear program is re-solved after each decision. The vanilla version of Neural Diving produces assignments that define the sub-MIP entirely in parallel, and only re-solves the linear program once. In section 12.1 we describe an extension where the decisions are made sequentially. Our approach is similar to Relaxation Enforced Neighborhood Search (RENS) (Berthold 2007) in that we fix a subset of variables and solve the resulting sub-MIP, but in our case the variables are assigned values predicted by a learned model rather than based on the linear program solution, and we use multiple partial assignments instead of only one.

6.1. Solution Prediction as Conditional Generative Modelling

Consider an *integer program* (*i.e.*, all variables are integers) with parameters $M = (A, b, c)$ (see equation 1) and a nonempty feasible set over a set of integer variables x. Assuming minimization, we define an *energy function* over x using the objective function:

$$E(x; M) = \begin{cases} c^T x & \text{if } x \text{ is feasible,} \\ \infty & \text{otherwise,} \end{cases} \tag{8}$$

which then defines the conditional distribution

$$p(x|M) = \frac{\exp(-E(x;M))}{Z(M)} \tag{9}$$

where $Z(M)$ is the *partition function* that normalizes the distribution to sum to 1:

$$Z(M) = \sum_{x'} \exp(-E(x';M)). \tag{10}$$

Feasible assignments with better (*i.e.*, lower) objective values have higher probability. Infeasible assignments have zero probability. Note that multiplying c by a constant β changes the distribution, even though the feasible set and the optimal assignment(s) remain the same. (β can be interpreted as the inverse temperature parameter of the distribution.) The distribution can be made invariant to such a re-scaling of the objective by normalizing the energy as $E(x;M)/|E(x^*;M)|$ where x^* is an optimal assignment. We have not used such normalization in this work.

When a MIP contains both integer variables x_I and continuous variables x_C, a given assignment for x_I defines a linear program (LP) on x_C. The energy function $E(x;M)$ can then be computed by assigning x_C to the optimal LP solution ${x^*}_C$ (if feasible) and then setting $E(x;M)$ to the objective value of the resulting complete assignment.

6.1.1. Learning The learning task is to approximate $p(x|M)$ using a generative model $p_\theta(x|M)$ parameterized by θ. The training dataset is $\mathcal{D}_{\text{train}} = \{(X_i, M_i)\}_{i=1}^{N}$, where $\{M_i \sim p(M)\}_{i=1}^{N}$ are N independent and identically distributed (IID) samples from the application-specific MIP distribution $p(M)$ and $X_i = \{x^{i,j}\}_{j=1}^{N_i}$ is a set of unique N_i assignments for the instance M_i. X_i is obtained by running SCIP on M_i and collecting feasible assignments it finds during the solve. While potentially costly, this data collection step needs to be done only once (per application), and outside the training loop. We learn on *all* assignments (explained below), not just the best ones, and do not require that any of the assignments be optimal.

The model parameters θ are learned by minimizing the following weighted loss function with respect to θ:

$$L(\theta) = -\sum_{i=1}^{N}\sum_{j=1}^{N_i} w_{ij} \log p_\theta(x^{i,j}|M_i), \tag{11}$$

where the weights w_{ij} are used to reduce any bias in sampling the training examples $X_i = \{x^{i,j}\}_{j=1}^{N_i}$ for the instance M_i. Consider the case where X_i contains all possible feasible assignments of the integer variables for M_i without any duplicates (*i.e.*, $x^{i,j} \neq x^{i,k},\ \forall j \neq k$). Then we can use the weights

$$w_{ij} = \frac{\exp(-c_i^T x^{i,j})}{\sum_{k=1}^{N_i} \exp(-c_i^T x^{i,k})}. \tag{12}$$

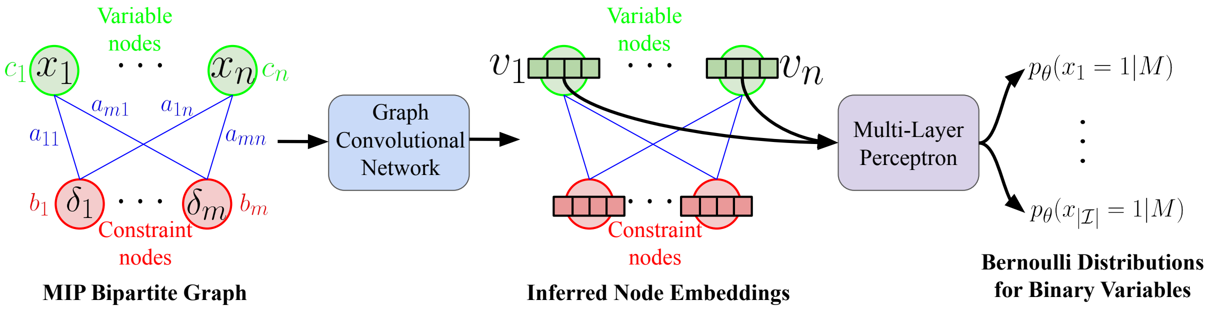


Figure 5 Conditionally independent model (Section 6.1.2) infers node embeddings $\{v_1, \ldots, v_n\}$ for the variable nodes of the bipartite graph representation of an input MIP M using a graph convolutional network and applies a multi-layer perceptron to the node embeddings $\{v_d\}_{d\in\mathcal{I}}$ of the set of binary variables $\mathcal{I}$ in M to compute their Bernoulli distributions. (See Section 6.1.3 for the case of general integers.) These distributions are independent of each other conditioned on the input MIP.

in equation 11 to learn the target distribution $p(x|M_i)$ using X_i. Enumerating all possible assignments to construct X_i is not practical for MIPs of realistic size. Instead we apply an off-the-shelf solver, SCIP in our case, to M_i. We define X_i to include *all* feasible assignments found throughout the solve, with duplicates removed. Using the weights given by equation 12 no longer corresponds to learning the exact target distribution $p(x|M_i)$ as the partition function is approximate. Although approximate, learning still succeeds in producing models with strong empirical performance on the datasets used in our evaluation. Intuitively we expect the approximation to produce good results because it assigns higher weights to better assignments.

6.1.2. Conditionally-independent Model There are several choices for parameterizing the generative model (see, e.g., Bishop (2006), Bengio and Bengio (2000), Kingma and Welling (2014)). Let $\mathcal{I}$ be the set of dimensions of x corresponding to the integer variables. Let x_d denote the d^{th} dimension of x. We use a model of the form:

$$p_\theta(x|M) = \prod_{d\in\mathcal{I}} p_\theta(x_d|M), \tag{13}$$

which predicts a distribution for x_d that is independent of the other dimensions conditioned on M.

For simplicity, we first assume that each x_d is binary with possible values $\{0, 1\}$ (the general integer case is considered in section 6.1.3). We use the Bernoulli distribution for each such variable. The success probability μ_d for the Bernoulli distribution $p_\theta(x_d|M)$ is computed as

$$t_d = \text{MLP}(v_d; \theta), \tag{14}$$

$$\mu_d = p_\theta(x_d = 1|M) = \frac{1}{1 + \exp(-t_d)}, \tag{15}$$

where v_d is the embedding computed by a graph convolutional network for the MIP bipartite graph node corresponding to x_d, as described in section 3.2. Note that the same MLP is used for all variables, see figure 5.

While the conditionally-independent model cannot accurately model a multimodal distribution over assignments, empirically, it still shows strong performance in our experiments. We also investigated more sophisticated models, such as autoregressive models (Bengio and Bengio (2000), see Section 12.1), which provide modest improvements but at a much higher inference cost.

6.1.3. Handling General Integers The model can be generalized to non-binary integer variables with some modifications. The main challenges are: a) the cardinality of an integer variable can vary significantly across instances, and b) it can potentially be very large (e.g., 10^7). This makes simple approaches such as assuming a fixed maximum cardinality highly inefficient and difficult to learn with.

We address these challenges by reframing the prediction task for general integer variables as a sequence of binary prediction tasks, based on the binary representation of the target integer value. For an integer variable z that can be assigned values from a finite set with cardinality $\text{card}(z)$, any target value can be represented as a sequence of $\lceil \log_2(\text{card}(z)) \rceil$ bits. We train our model to predict these bits in sequence, from most significant to least significant bit. Since the maximum cardinality of variables in a test instance becomes known only during inference, and is unknown during training, we introduce a hyperparameter n_b that controls the maximum number of bits predicted for each variable along this bit sequence. We can then train our model to predict the n_b most significant bits of the value for z, given the upper and lower bounds for z. During inference then, if $\lceil \log_2(\text{card}(z)) \rceil$ exceeds n_b, we can still predict the n_b most significant bits of the value for z, and use these predictions to tighten the bounds of z by a factor of at least $2^{n_b - 1}$. Otherwise, if $\lceil \log_2(\text{card}(z)) \rceil \leq n_b$, then we can predict z exactly to a single value. Note that this approach of bit-wise prediction of values can also be seen as predicting branching decisions on variables: Predicting one bit means picking the left or right branch of a binary tree, and the possible range of the integer variable is successively tightened after each prediction.

6.2. Combining Model Predictions with a Classical Solver

Once a model is learned, we can then sample variable assignments $x \sim p_\theta(x|M)$ for a new problem M. While we could try to use these samples directly, they need not be feasible or provide the best objective value. Instead, it is more effective to only fix a (large) subset of the variables to their sampled values, and delegate the solution search for the remaining open variables to a classical solver, in our case, SCIP.

We use the SelectiveNet approach (Geifman and El-Yaniv 2019) to train an additional binary classifier that decides which variables to predict a value for and which to refrain from predicting, and optimizes for "coverage" among variables, defined as the ratio of the number of variables predicted vs not predicted. Specifically, we introduce an additional output $y_d \in \{0,1\}$ for each variable x_d in the input M that determines whether x_d should be assigned ($y_d = 1$) or not. For the conditionally-independent model, we can then train our model by minimizing the following loss function:

$$l_{\text{selective}}(\theta, x, M) = \frac{-\sum_{d\in\mathcal{I}} \log p_\theta(x_d|M) \cdot y_d}{\sum_{d\in\mathcal{I}} y_d} + \lambda\Psi(C - \frac{1}{|\mathcal{I}|}\sum_{d\in\mathcal{I}} y_d), \tag{16}$$

$$L_{\text{selective}}(\theta) = \sum_{i,j} w_{ij} \cdot l_{\text{selective}}(\theta, x^{i,j}, M_i). \tag{17}$$

Here, C is a coverage threshold representing the desired relative frequency of assigned variables, Ψ is a quadratic penalty term, and λ is a hyperparameter controlling the relative importance of achieving a coverage close to the set threshold. We train multiple models simultaneously with different coverage thresholds (typically, values from 0.1 to 0.95, and are tuned on the validation set).

By assigning, or tightening the bounds of a large fraction of the variables we significantly reduce the problem size, and warm-start SCIP to find high quality solutions in much shorter time. The idea here is that by sampling a solution from $p_\theta(x|M)$, we move to a promising region of the solution space; and by removing again some of the assignments we expand the neighbourhood in which we then search for the optimal solution.

This approach also offers practical computational advantages: both the sampling of predictions, and the solution search afterwards are fully parallelizable. We can repeatedly and independently extract many different samples from our model, and each partial assignment of the sample can be independently solved. For a single MIP instance, we can generate many such partial assignments, each of which we can independently solve with SCIP. The feasible solution with the best objective value across all sub-MIPs is then reported as the final solution to the original input MIP. Our approach can hence easily leverage the power of distributed computing even when solving for a single instance, which is not possible with default SCIP (without significant effort (Shinano et al. 2011)). When comparing results, we will show results for both the parallel setting (where we make full use of the parallelism advantage of our approach) and the sequential setting (where we run our approach in a sequential manner, controlling for the total amount of computational resources).

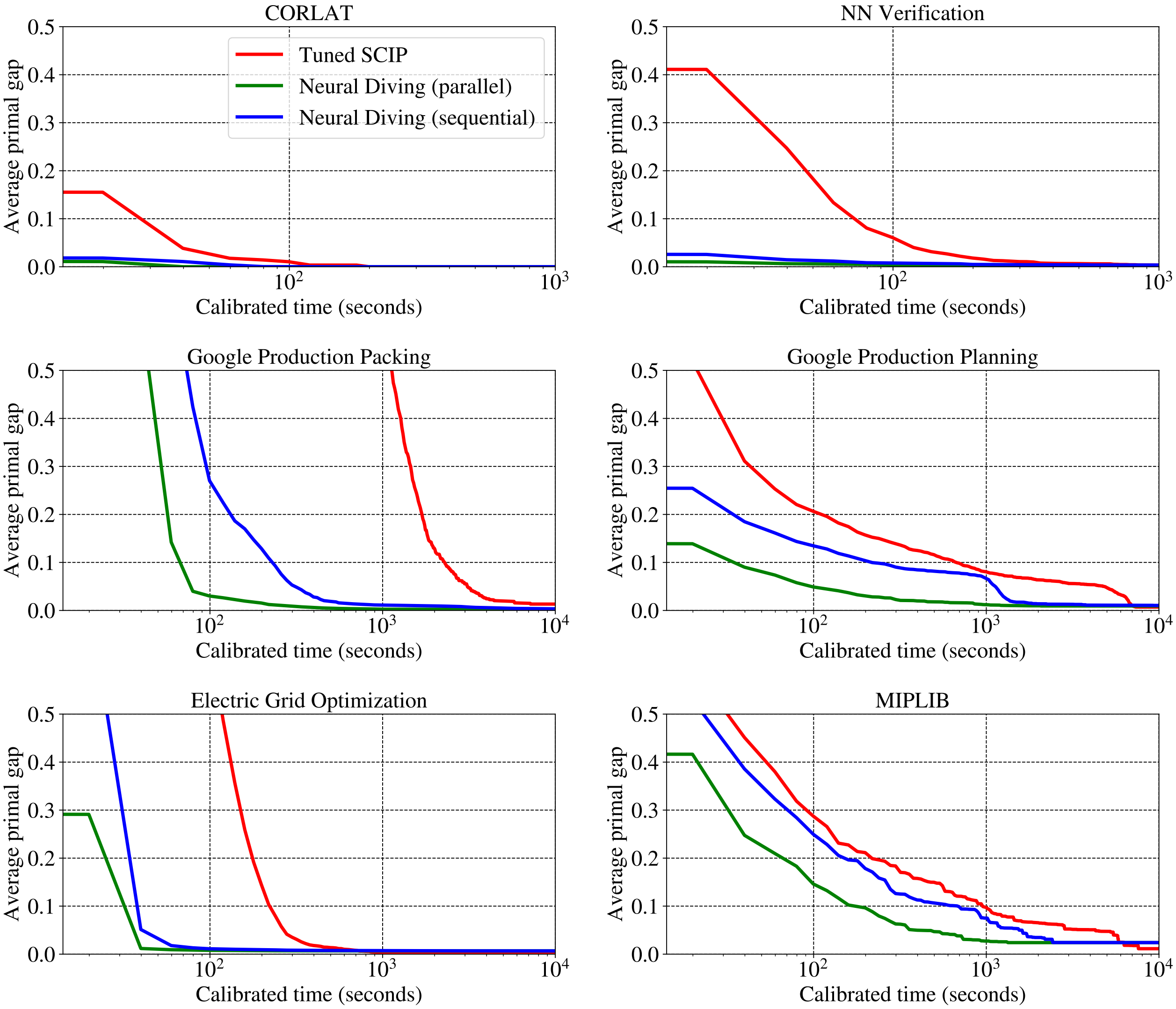


Figure 6 Average primal gap between the primal bound and the best known objective value achieved by various algorithms as a function of running time on the benchmark datasets.

6.2.1. Generating partial assignments As described above, to produce partial assignments to an input MIP, we use the output of our generative models to decide both which values each integer variable should take, as well as which variables to assign: The acceptance head decides which variables are fixed or tightened, while the prediction head provides the variable values. Algorithm 1 describes this procedure.

We can repeat the above procedure many times, by obtaining multiple samples from a model, and also by using multiple models trained with different choices for the coverage threshold for the acceptance decision, to produce a diverse set of sub-MIPs. In experiments with the conditionally independent model we use one sample per model, and generate different sub-MIPs by using models trained with different coverage thresholds for y.

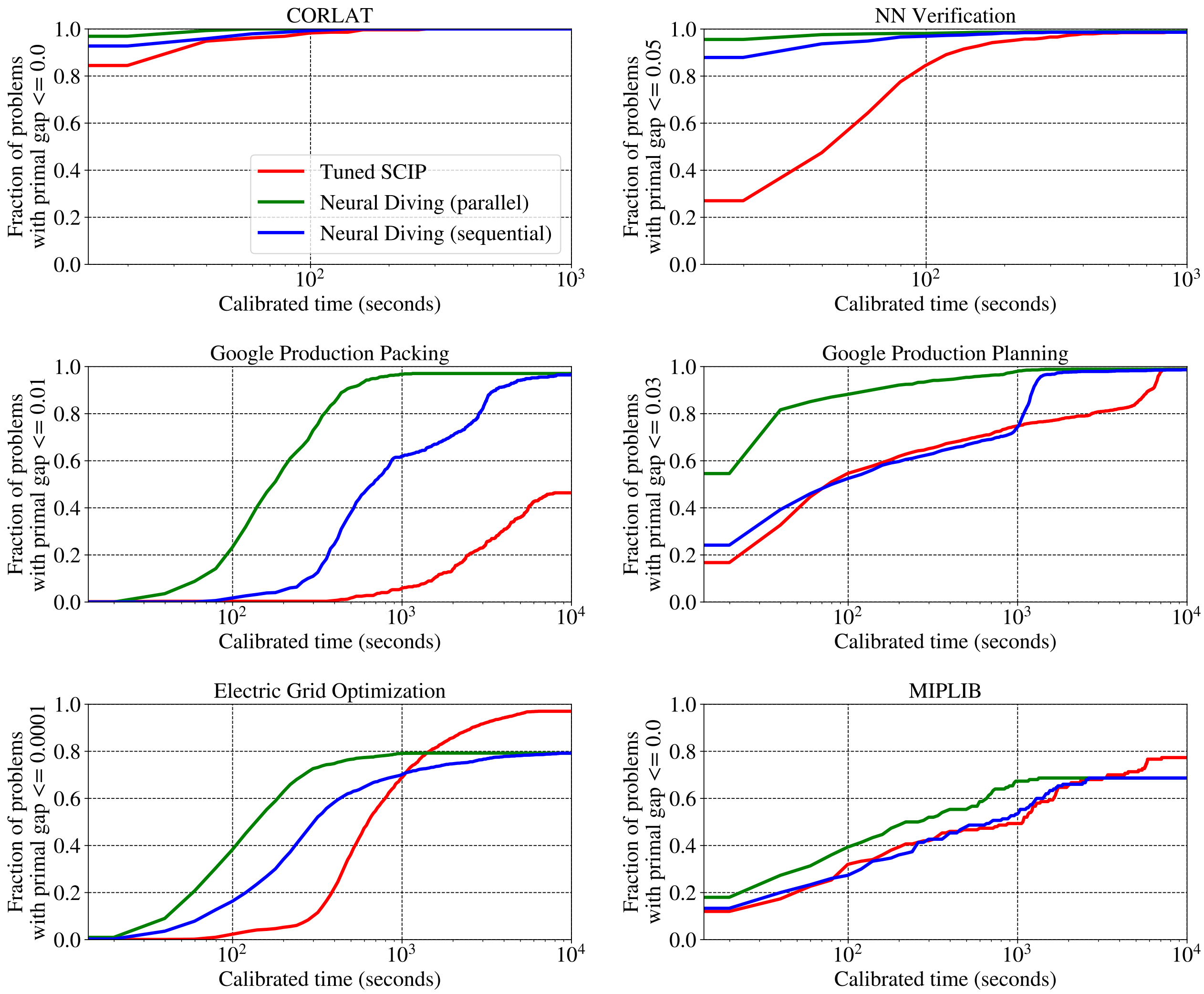


Figure 7 Fraction of instances with primal gap less than or equal to the target gap achieved by various algorithms as a function of running time on the benchmark datasets.

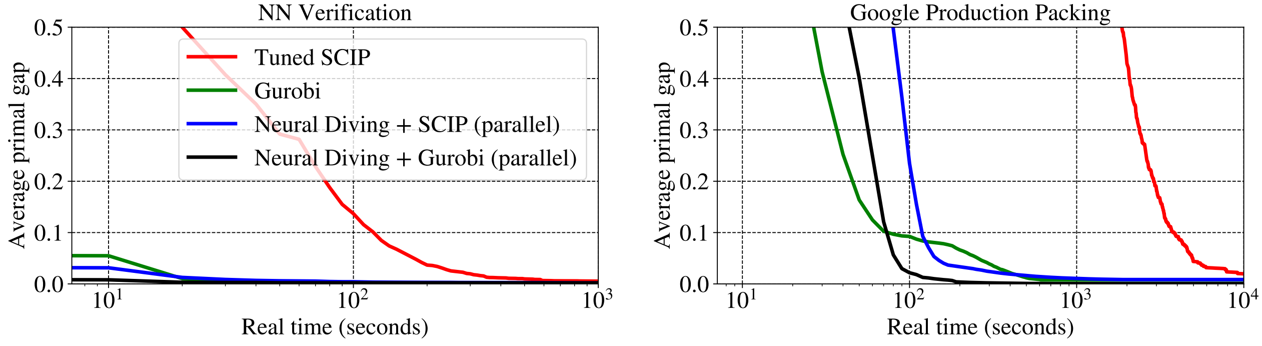


Figure 8 Average primal gap between the primal bound and the best known objective value achieved by various algorithms as a function of running time on the benchmark datasets.

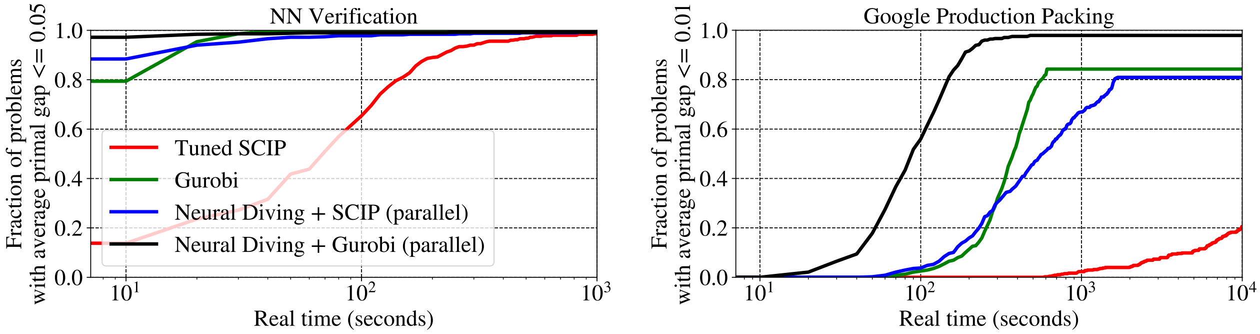


Figure 9 Fraction of instances with primal gap less than or equal to the target gap achieved by various algorithms as a function of running time on the benchmark datasets.

6.2.2. Sequential and parallel sub-MIPs solving The sub-MIPs that we generate in the previous step can be solved fully independently from each other, allowing us to search for solutions for all sub-MIPs in parallel. In this parallel setting, we generate up to 100 sub-MIPs by applying combinations of different models and SCIP random seeds, and distribute each sub-MIP onto their own dedicated machine. Since each sub-MIP in this setting is solved in parallel, we report results in terms of (calibrated) wall clock time, meaning that we take results from the best performing sub-MIP at any given time.

While the parallel setting showcases the advantage that our approach has when distributed compute resources are available, we also set up a comparison that controls for the amount of computational resources used. That is, in addition to the parallel setting, we also compare our approach to SCIP when both are run sequentially on a single machine. In this setting, we are still generating multiple sub-MIPs as before, but process them sequentially in a random order, instead of in parallel.

6.3. Results

We trained a GNN for every dataset on the training set and tuned hyperparameters (number of layers, width of layers, learning rate, exponential decay steps and assignment coverage thresholds) on the validation set. We report results on average primal gap vs our baseline SCIP in Figure 6. Our parallel and sequential runs overall produce better primal bounds in shorter time on all datasets, compared to SCIP. We believe that the strength of our approach is in quickly finding good solutions, but it sometimes fails to find an optimal or near-optimal solution. This can can be seen e.g. on the survival plots in figure 7 where Neural Diving approach wins at shorter time limits, but loses to SCIP at the end on the Electric Grid Optimization and MIPLIB datasets. Note that SCIP's internal definition of gap is different from the one used in this paper, so in all experiments we set the relative gap setting to 0 in order to avoid stopping prematurely.

Algorithm 1: Generating variable assignments and tightenings

```
Input: Learned distributions $p_\theta(x|M)$, $p_\theta(y|M)$, MIP instance $M$
Output: Variable assignment and bound tightenings

assignment $:= \{\}$
tightenings $:= \{\}$
for $x_i \in \text{Variables}(M)$ do
    if $x_i$ is binary variable then
        Sample $p_x$ from $p_\theta(x_i|M)$
        Sample $y$ from $p_\theta(y_i|M)$
        if $y = 1$ then
            Add $(x_i := \text{round}(p_x))$ to assignment
    if $x_i$ is non-binary integer variable then
        $lb :=$ lower bound of $x_i$
        $ub :=$ upper bound of $x_i$
        $b_0, ..., b_k :=$ binary representation of $(ub - lb)$, $b_0$ being most significant bit
        for $j \in \{0, ..., k\}$ do
            Sample $p_x$ from $p_\theta(x_{i,j}|M)$
            Sample $y$ from $p_\theta(y_{i,j}|M)$
            if $y = 1$ then
                if $\text{round}(p_x) = 1$ then
                    $lb := lb + \lceil (ub - lb)/2 \rceil$
                else
                    $ub := lb + \lfloor (ub - lb)/2 \rfloor$
            else
                Add $(lb \le x_i \le ub)$ to tightenings
                break
return assignment, tightenings
```

We include additional results on two datasets (Google Production Packing and NN Verification, see figures 8 and 9) where we combine Neural Diving with the Gurobi solver (Gurobi Optimization 2020): we assign variables in the same way, but use Gurobi instead of SCIP to solve the remaining problem. The reported results are in non-calibrated time as we do not have access to the Gurobi internals and cannot report the calibrated time during solving.

6.3.1. Finding incumbents to open MIPLIB instances In addition to evaluating Neural Diving on the datasets from Table 1, we also applied a modification of our approach on solving *open* instances from the MIPLIB 2017 Collection Set. These instances are classified as open, because no optimal solution is known: Either there is no feasible solution known; or there exists a known

Table 2 Incumbents found for open MIPLIB 2017 instances by Neural Diving. Lower objective value is better.

Instance	Neural Diving objective value	Previous best objective value
milo-v12-6-r1-75-1	**1153756.398**	1153880
neos-1420790	**3121.29**	3121.42
xmas10-2	**-497**	-495

feasible solution (incumbent), but it does not match any known dual bound. In this sense, these instances are the hardest problems that the MIPLIB 2017 Collection Set offers.

Apart from their inherent hardness, these instances also impose an additional challenge to our learning-based approach: The set of instances is almost entirely heterogeneous, in that there are only a handful of related instances available per problem, if at all. Our approach of learning a domain-specific primal heuristic hence needed to be adapted. To tackle this challenge, we built a training set for each attempted open instance which should represent the problem distribution of the target instance. More specifically, for a given target instance, we generated up to 500 similar instances through randomly applying a mix of the following manipulations: (1) Randomly dropping constraints of the target instance, and (2) fixing a random subset of variables to the previous incumbent. To generate each such instance, we applied 10 iterations of each step (1) and (2) on the original target instance.

After creating such a training set for each target instance, we apply our approach as described in earlier sections: We trained a separate GNN for each target instance, using the generated training set, with label solutions found on the training set using SCIP. After training, we used the corresponding GNN to produce up to 10 partial assignments to the target instance (as described in Section 6.2.1), and used Gurobi to solve the sub-MIPs defined by the partial assignments. For three open instances, this approach yielded new incumbents that were not known before. Table 2 shows the improved objective values compared the previous best known solution.

7. Neural Branching

A branch-and-bound procedure has two decisions to make every iteration: which leaf node to expand, and which variable to branch on. In this work we focus on the latter. The quality of the variable selection decision can have a large impact on the number of steps taken by branch-and-bound to solve a MIP (Achterberg et al. 2005, Glankwamdee and Linderoth 2011, Schubert 2017, Yang et al. 2019). We use a deep neural network to learn a variable selection policy by imitating the actions of a node-efficient, but computationally expensive, expert. By distilling the policy of the expensive expert into the neural network we seek to maintain approximately the same decision quality but substantially reduce the time taken to make the decision. The decision at a given tree node is entirely local to that node, so a learned policy only needs to have a representation of the node as input, rather than the entire tree, which makes it more scalable.

7.1. Expert policy

We would like an expert policy that solves MIPs by building small branch-and-bound trees, since such an expert is making good branching decisions. Among the many branching policies proposed in the optimization literature (see, e.g., Achterberg et al. (2005)) none provably achieve the smallest trees, but empirically *full strong branching* (FSB) (Achterberg et al. 2005) tends to use fewer steps than competing approaches. It performs one-step lookahead search by simulating one branching step for *all* candidate variables and picking the one that provides the largest improvement in the dual bound, as determined by the solution to the linear program at the new node. (A generalization to k-step lookahead search can be found in Glankwamdee and Linderoth (2011), Schubert (2017).) This requires solving $2 \times n_{\text{cands}}$ linear programs, where n_{cands} is the number of possible branching candidates at a given step of branch-and-bound. Concretely, denote by $x^\star$ the solution to the LP relaxation of (1) at the current node we are branching from with variable constraint vectors l and u, *i.e.*, $l \leq x^\star \leq u$. Then for each variable candidate i, FSB needs to solve the following two LPs:

$$\begin{array}{llll} \text{minimize} & c^\top x^{\text{up}} & \text{minimize} & c^\top x^{\text{down}} \\ \text{subject to} & Ax^{\text{up}} \leq b & \text{subject to} & Ax^{\text{down}} \leq b \\ & l^{(i)} \leq x^{\text{up}} \leq u & & l \leq x^{\text{down}} \leq u^{(i)} \end{array} \tag{18}$$

over variables $x^{\text{up}} \in \mathbb{R}^n$ and $x^{\text{down}} \in \mathbb{R}^n$, where $l_i^{(i)} = \lceil x_i^\star \rceil$ and $u_i^{(i)} = \lfloor x_i^\star \rfloor$, and other entries of $l^{(i)}$ and $u^{(i)}$ are unchanged from their values at the current node, l and u. It combines the optimal values of these LPs into a score for that candidate, and it uses those scores to decide which variable to branch on.

For practical MIP solving FSB's per-step computational cost is often so high that it can be used only for a few branch-and-bound steps before the running time becomes prohibitively high. SCIP's default variable selection policy, *reliable pseudocost branching* (RPB) (Achterberg et al. 2005), uses FSB for a small number of steps at the start of branch-and-bound and switches to a heuristic with lower per-step cost for the subsequent steps. But an expert policy used as the target for imitation learning can afford a higher per-step cost. The expert is only used to generate training data offline as a one-time expense by running it on a training set of MIPs and recording its inputs and outputs at each step. Even then, on large-scale instances (e.g., with $> 10^5$ variables), offline data generation can be prohibitively slow, which prevents learning approaches from scaling to such instances. Therefore we speed up the expert policy by exploiting GPUs, as explained next.

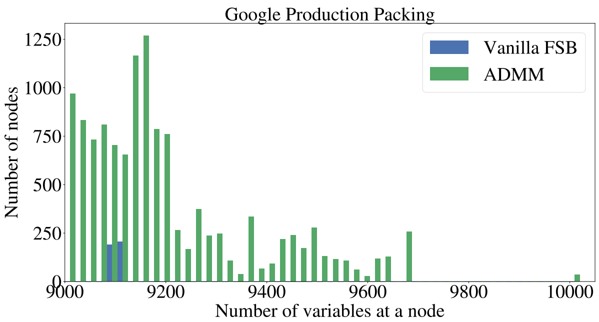


Figure 10 Histogram of the number of imitation learning examples, where each example is a node in a branch-and-bound search tree, generated by the ADMM expert and the Vanilla Full Strong Branching expert on the training sets for Google Production Packing and MIPLIB in the same time limit.

7.1.1. ADMM batch LP solving SCIP by default uses the LP solver SOPLEX (Gamrath et al. 2020) based on the simplex algorithm (Dantzig 1998) for solving the linear programs arising from FSB. Although in principle the simplex algorithm can be parallelized, it is not easy to *batch* the computation together for several LPs that are closely related (and therefore leverage a GPU), and consequently SOPLEX simply solves the LPs sequentially on CPU. In order to scale to a large number of candidate variables at a reasonable cost we wrote our own batch LP solver using the alternating directions method of multipliers (ADMM) (Boyd et al. 2011), based on the Splitting Conic Solver (SCS) algorithm for cone programs (O'Donoghue et al. 2016, O'Donoghue 2020). This solver can simultaneously handle many LPs in a batch that we process on GPU. This can (approximately) solve all the LPs in full strong branching substantially faster than running SOPLEX sequentially. Moreover, since we are using this solver merely to produce training data, we can tune the algorithm parameters with the primary goal of producing smaller search trees, even at the cost of some data generation time. It is the output produced by this ADMM-based policy that we will train a graph neural network to imitate. The graph neural network is able to produce actions that are close enough to the policy of the ADMM expert to produce small search trees, but at a fraction of the computational cost. We provide full details in the Appendix, section 12.5.

To demonstrate the improved scalability of the expert data generation, we compare the ADMM expert to the expert used by Gasse et al. (2019) called Vanilla Full Strong Branching (VFSB) on Google Production Packing and MIPLIB. VFSB is a modified version of SCIP's FSB implementation that is still performed sequentially but disables its 'side effects', *i.e.*, changes to the solver state not directly due to a branching decision itself, but due to information gained during the lookahead search over branching candidates. Figure 10 shows a histogram of the number of variables in each branch-and-bound node generated by the two experts within 60 hours, on nodes with the largest number of variables for both datasets (> 9000 variables for Google Production Packing and $> 10^5$

variables for MIPLIB). The ADMM expert generates 31.7× and 1.6× more data on nodes with the largest number of variables for Google Production Packing and MIPLIB, respectively. Across the full range of the number of variables, the ADMM expert generates 12.1× and 1.4× more data on Google Production Packing and MIPLIB, respectively. The largest node that the VFSB expert manages to generate data from on MIPLIB has 3.4×10^5 variables, while the ADMM expert is able to generate data from nodes with almost 10^6 variables.

7.2. Imitation Learning

Imitation learning is the name given to the broad family of algorithms that seek to learn the policy of an expert, given examples of expert behavior. In most cases, and in this paper, this is formulated as a supervised learning problem. We consider three variants: 1) cloning an expert policy (Pomerleau 1989, Bain and Sammut 1995) 2) distillation with random moves, and 3) DAgger (Ross et al. 2011). Distillation is the simplest of the three and it simply attempts to learn to predict the outputs of the expert at each node. Distillation is not robust to shifts away from the expert's state distribution. This is particularly problematic in the case of branching as mistakes near the root can cause the subsequent nodes visited in the search tree to be very different than those seen during training, since the expert never visited those nodes. Mixing random moves with expert moves is a simple heuristic that can alleviate this problem to some degree. When running the expert we take a random action with probability 10% at each node. We can run the expert many times on each training MIP generating slightly different data each time, thereby increasing the total amount of training data available. In all our experiments we did this five times for each MIP. DAgger is another, more involved, procedure where we train an agent using distillation, run it in a branch-and-bound procedure to make decisions, but also compute the expert outputs at each step to use as targets for learning. This produces one step of DAgger data upon which we can re-train the agent. In theory this procedure can be repeated using the new agent trained on the previous agent DAgger data, but in this manuscript we only consider a single iteration of DAgger.

In our experiments, we treat the choice of the three imitation learning variants as a hyperparameter to tune for each dataset. We generate data and train policies using all three variants for each dataset, and we select the policy that achieves the lowest average dual gap on the validation set instances in a 3 hour time limit. The selected policy is then evaluated on the test set to report results.

7.3. Implementation details

The ADMM expert solves a pair of LPs for each branching candidate (the 'up' and 'down' branches). Let us denote by $\mathrm{OPT}_i^{\mathrm{up}}$ and $\mathrm{OPT}_i^{\mathrm{down}}$ the (estimated) optimal values of the LP for the up and down branches respectively for candidate i, and denote by OPT the objective value of the LP at the node we are branching from. We combine these numbers into a single score for each variable defined as

$$s_i = (\mathrm{OPT}_i^{\mathrm{up}} - \mathrm{OPT} + \epsilon)(\mathrm{OPT}_i^{\mathrm{down}} - \mathrm{OPT} + \epsilon) \tag{19}$$

where $\epsilon = 10^{-4}$ (this is essentially the product rule from Achterberg (2009)). Given a set $\mathcal{C}$ of candidate variables for branching, we convert their scores into a categorical distribution over the candidates using

$$p_i^{\mathrm{expert}} = \frac{s_i}{\sum_{c \in \mathcal{C}} s_c}.$$

On the training set of MIPs we ran the ADMM expert, with random moves or initialized using DAgger as appropriate, and logged the expert scores taken at each node (we did not log the action if it was selected randomly or selected by the agent in the DAgger setting - only the ADMM expert action was ever logged). This provides our training dataset for the imitation learning agent.

The neural network policy for variable selection has the same graph convolutional network architecture described in section 3.2. It conditions on a leaf node of the branch-and-bound tree selected by SCIP to branch on. As the node itself is a MIP, it can be represented using the bipartite graph representation described in section 3.1. Its output is a categorical distribution over the set of candidates $\mathcal{C}$. Let M be the MIP bipartite graph representation of the leaf node, v_c be the embedding computed by a graph convolutional network for the bipartite graph node for the branching candidate variable x_c, as described in section 3.2, and ϕ be the learnable parameters of the policy. The probability $p_\phi(x_c|M)$ of selecting x_c for branching at the node is given by:

$$t_c = \mathrm{MLP}(v_c; \phi), \tag{20}$$

$$p_\phi(x_c|M) = \frac{\exp(-t_c)}{\sum_{c' \in \mathcal{C}} \exp(-t_{c'})}. \tag{21}$$

When training, the neural network receives a batch of node features with the associated expert scores and produces a set of probabilities p^{graphnet} for each node in the batch by passing the activations of the last layer through a softmax. We used batch size 8 sampled uniformly at random from the training dataset. We tried a variety of losses and found that negative cross-entropy loss performed best on average, which is given by

$$L(\phi) = \sum_{c \in \mathcal{C}} p_c^{\mathrm{expert}} \log p_\phi(x_c|M).$$

The loss is minimized using ADAM (Kingma and Ba 2014) with a learning rate of 10^{-4}. Figure 16 in the appendix shows that imitation learning succeeds to accurately approximate the expert policy, both on the training and test sets.

7.4. Results

We evaluate the learned branching policy on the task of optimizing the dual bound. As explained in section 5, for comparison we use the dual gap $\gamma_d(t)$ with respect to a pre-computed best known primal bound $p^\star$ as a function of time. The gap is averaged over all MIPs in the test set of a dataset.

Figure 11 shows the average dual gap curves for Neural Branching and Tuned SCIP. We also compare to SCIP's Full Strong branching policy (run with all SCIP parameters set to default, except the branching policy) as a node-efficient baseline. Neural Branching achieves significantly smaller average dual gap for the same running time on four out of six datasets. (Note the log scale of the y-axis.) MIPLIB and Google Production Planning are the only datasets where Neural Branching is not the best, but it is still comparable to Tuned SCIP. The improvements given by Neural Branching can be substantial – e.g., its average dual gap is about 7× better on CORLAT, 20× better on Neural Network Verification, and 2× better on Electric Grid Optimization, compared to Tuned SCIP at large time limits. On Google Production Packing, Neural Branching is better at intermediate times (e.g., about 1.5× better at 10^3 seconds), with Tuned SCIP catching up eventually.

Figure 12 shows survival plots computed by applying a dataset's target optimality gap to the dual gap of each test set MIP instance. They confirm the conclusions from figure 11 – Neural Branching solves a higher fraction of test instances consistently across all time limits on all datasets except Google Production Planning and MIPLIB.

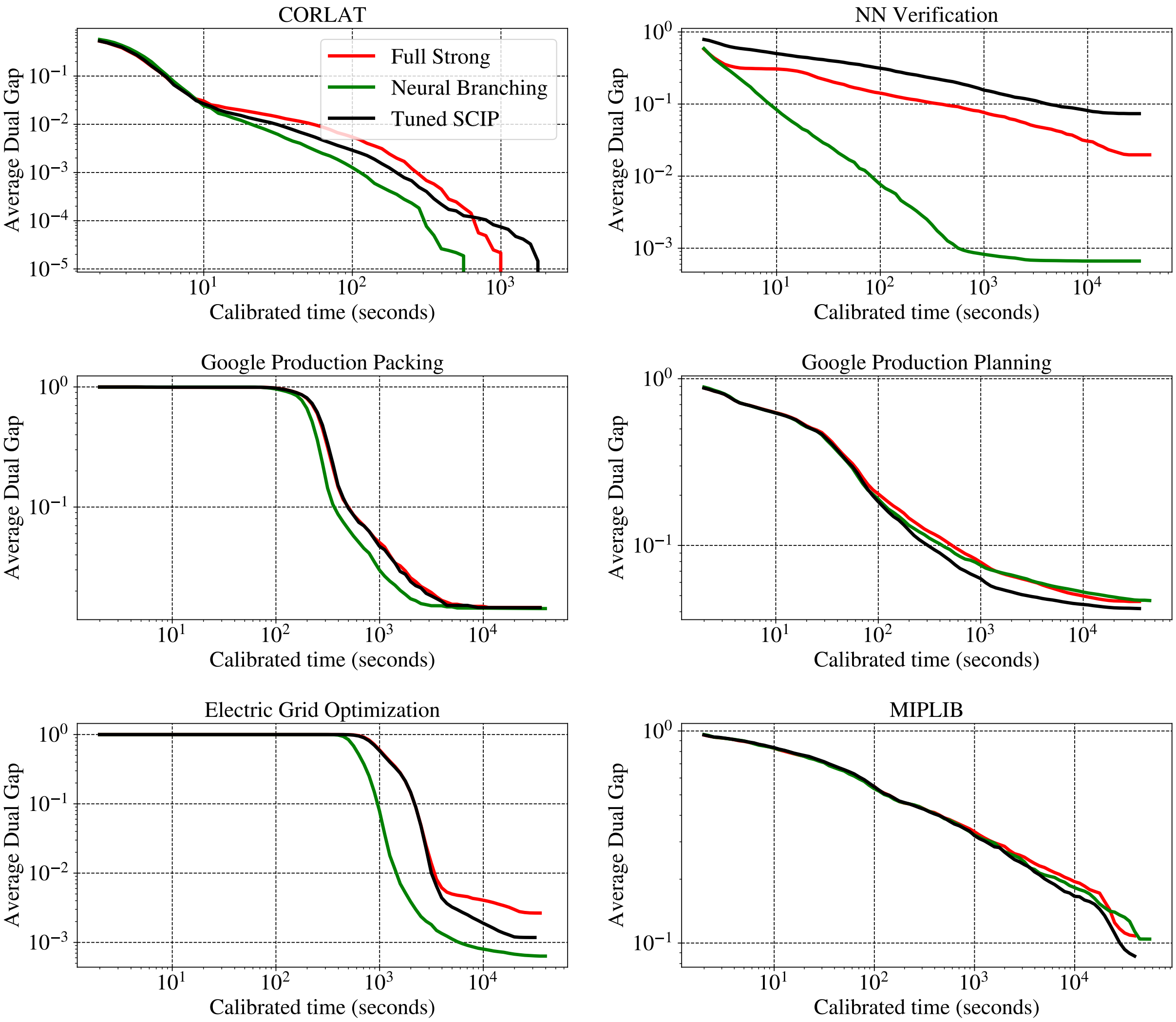


Figure 11 Average dual gap with respect to a pre-computed best known primal bound as a function of running time on the test set of benchmark datasets.

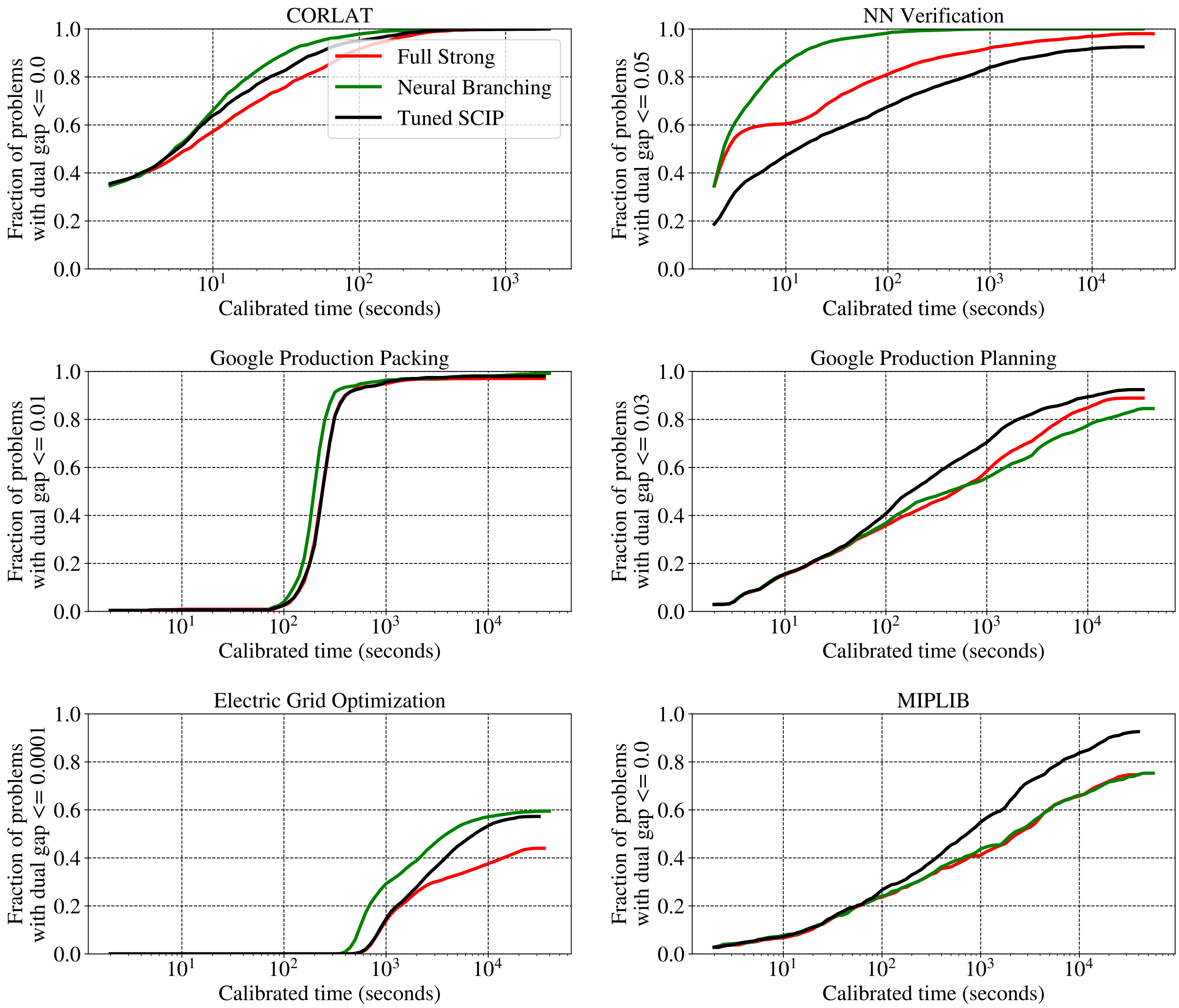


Figure 12 Survival plots computed by applying a dataset's target optimality gap to the dual gap of each test set MIP instance.

8. Joint Evaluation

We now combine Neural Branching and Neural Diving into a single solver. As we shall see, this results in significant speedups over Tuned SCIP. We use the learned heuristics by integrating them into SCIP via the interface provided by PySCIPOpt (Maher et al. 2016). We consider all four possible ways of combining Neural Branching and Neural Diving with SCIP: 1) *Tuned SCIP* alone, 2) *Neural Branching + Neural Diving (Sequential)* uses both neural heuristics, 3) *Neural Branching* uses only the learned branching policy, and 4) *Tuned SCIP + Neural Diving (Sequential)* uses only the sequential version of Neural Diving. When Neural Diving (Sequential) is used as a primal heuristic, we disable all built-in primal heuristics of SCIP.

To make a comparison based on Neural Diving fairer in terms of compute resources, we only consider its sequential version here. For a single MIP solve, Neural Diving (Sequential) and Neural

Branching each use a CPU core and a GPU. So *Neural Branching + Neural Diving (Sequential)* uses two cores and two GPUs. SCIP uses only one core and cannot use GPUs. Note that this means Tuned SCIP uses one less core than any of the neural solvers. The best strategy for using multiple cores with SCIP remains open. A simple approach to use K cores is to solve K copies of the instance with K random seeds to exploit seed-dependent performance variability. We defer such a comparison to future work.

For each expert we solve each MIP in the test dataset five times, with five different seeds for SCIP. We present the primal-dual gap $\gamma_{pd}(t)$ (see section 5) averaged over each dataset's test MIPs as a function of calibrated time. We also present survival plots that show the fraction of test MIPs solved within the required dataset-specific optimality gap as a function of calibrated time.

8.1. Results

Figure 13 shows the average primal-dual gap curves as a function of running time. In all cases a neural MIP solver significantly outperforms Tuned SCIP. (Note the log scale of the y-axis.) The average primal-dual gap is about six orders of magnitude better than Tuned SCIP on Neural Network Verification at large time limits. It is about 10× better on Google Production Packing, 2.5× better on Google Production Planning and Electric Grid Optimization, and 2× better on MIPLIB. Neural Branching + Neural Diving (Sequential) performs consistently well, being the best or near-best across all datasets. Tuned SCIP + Neural Diving (Sequential) is the best or near-best on all datasets except Neural Network Verification and Electric Grid Optimization. Optimizing the dual bound is the main challenge on those two datasets, which explains the strong performance of Neural Branching on them (and confirmed by the dual gap results in section 7.4).

Figure 14 shows the survival plots as a function of running time. They further confirm the observations from figure 13, with a neural solver solving a higher fraction of test set problems at a given time limit than Tuned SCIP. On Google Production Packing the number of problems solved more than doubles compared to Tuned SCIP at the largest time limit we tried. On MIPLIB, about 23% more problems are solved than Tuned SCIP at the largest time limit. Again, Neural Branching + Neural Diving (Sequential) is the most consistent neural solver by being the best or near-best on all datasets.

Together, the results show that learning can be used to significantly improve a strong solver such as SCIP. The improvements are seen across different problem scales and applications. Surprisingly, learning improves performance even on MIPLIB, a heterogeneous dataset combining many applications that a priori would not be considered similar enough to have sufficient shared structure for learning to exploit. That the improvements are observed across diverse settings confirms the broad usefulness of learning.

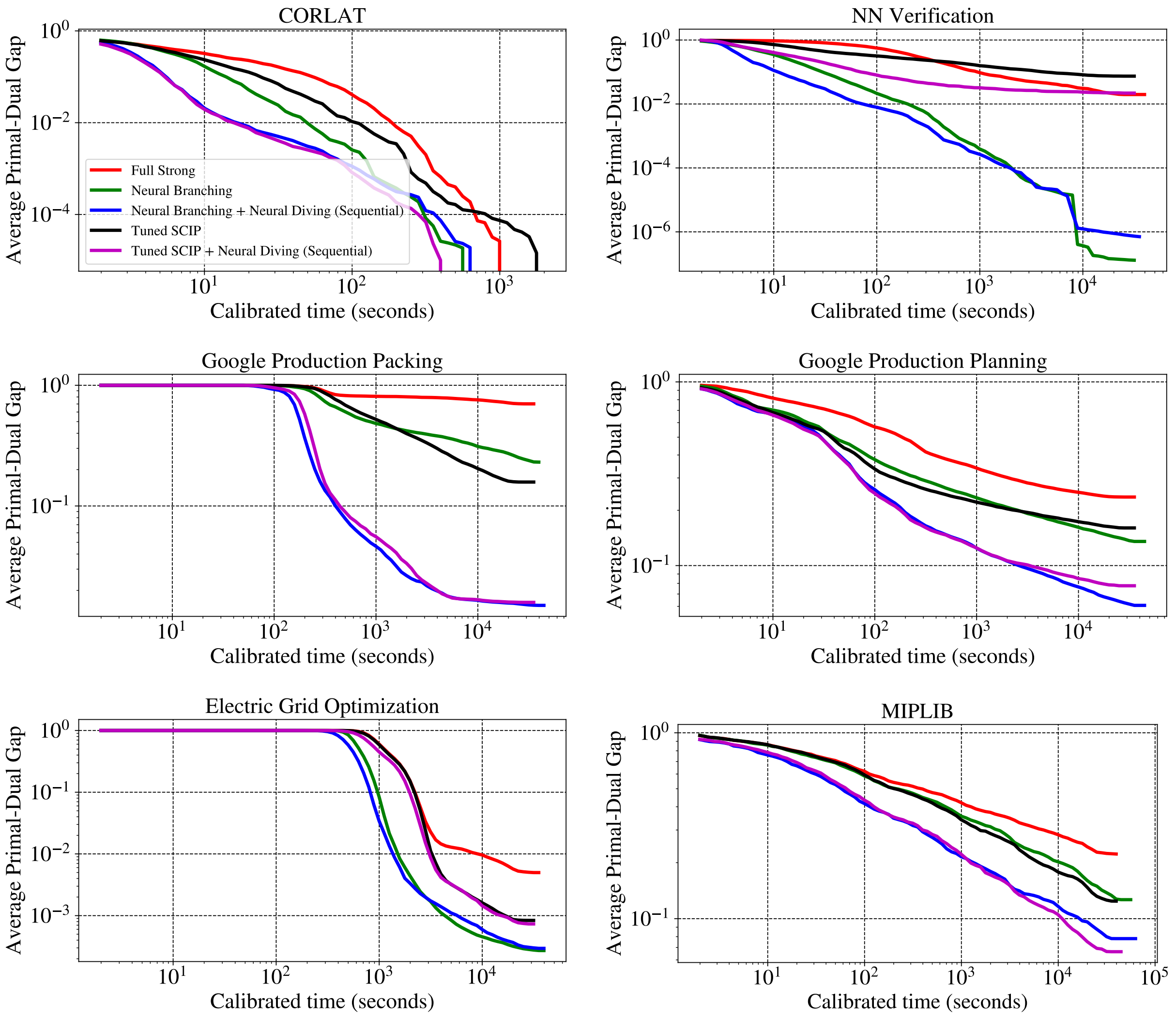


Figure 13 Primal-dual gap achieved by various combinations of primal heuristics and variable selection policies as a function of running time on the benchmark datasets. The gap is computed for each MIP in the test set of each dataset and then aggregated using the average.

A further comparison with respect to the *Penalized Average Runtime* metric can be found in Appendix section 12.4. The overall conclusions remain the same.

9. Related Work

Learning and MIP Primal Heuristics: Learning has previously been used to switch among an ensemble of existing primal heuristics during a branch-and-bound run. Khalil et al. (2017b) learn a binary classifier offline to predict whether applying a primal heuristic at a branch-and-bound node will succeed in finding an incumbent. Hendel (2018) formulate a multi-armed bandit approach to learn a switching policy online. Neural Diving constructs a new primal heuristic specifically for a

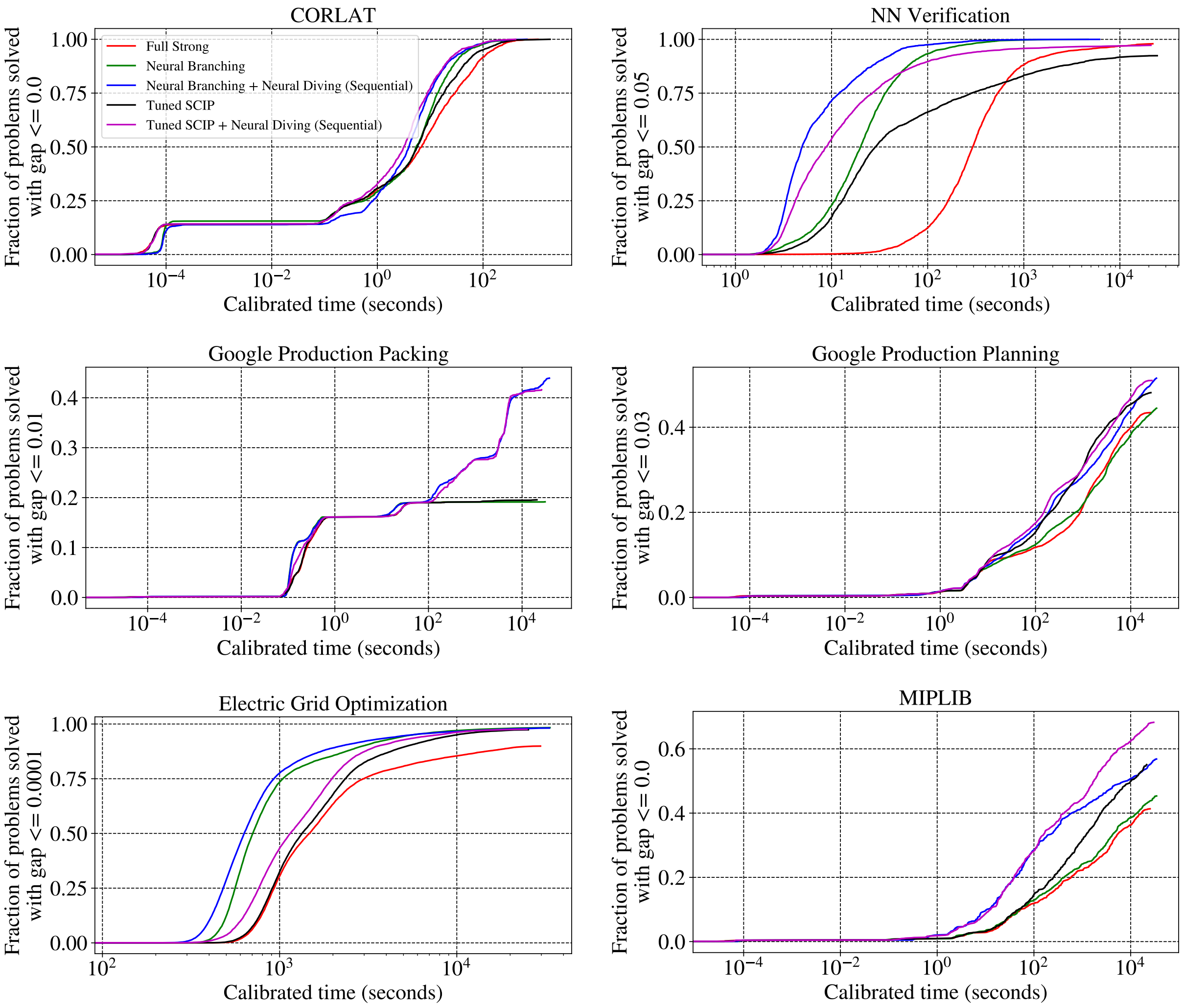


Figure 14 Survival plots with respect to calibrated running time for various combinations of primal heuristics and variable selection policies as a function of running time on the benchmark datasets.

given application. This can be particularly powerful if none of the existing heuristics are well-suited for the application.

Several works consider learning neighborhood search policies (Addanki et al. 2020, Hottung and Tierney 2019, Song et al. 2020). Given a feasible point, a learned policy is used to modify the assignment for a subset of variables such that the resulting feasible assignment has a lower objective. In these works the learned policy is used to either select the variables to be modified, or to assign new values to an already selected subset of variables. They require a feasible assignment as input, while Neural Diving does not.

Learning to predict variable values in a MIP and combining the predictions with a solver has been studied before. Ding et al. (2020) trains a model that takes a MIP as input and predicts values for a subset of its binary variables. This is then used to define an additional constraint that any solution be within a pre-specified Hamming distance from the predicted values, which can significantly reduce the search space and speed up a MIP solver. Training labels are generated by iteratively improving a feasible assignment and selecting only the subset of binary variables whose values are stable across iterations as prediction targets. Results are presented on several synthetic datasets. Xavier et al. (2020) learns to warm start a MIP solver specifically for the electric grid *unit commitment* problem. They use a k-Nearest Neighbor binary classifier to predict values for a subset of binary variables in the MIP. Assuming a fixed grid topology and a fixed set of power plants allows them to use a MIP representation with constant dimensionality and a pre-defined distance metric to identify neighbors. Our work differs significantly from both of these: 1) the generative modelling formulation provides a principled approach to capturing the uncertainty in variable values and exploiting it to generate several sub-problems to find high-quality feasible assignments, 2) it handles general MIPs (varying sizes, non-binary integers), and 3) we present results on diverse real-world applications.

Learning and MIP Branching Heuristics: Several works have used imitation learning to train a branching policy to imitate (a variant of) Full Strong Branching. Khalil et al. (2016) learns to imitate Full Strong Branching for a single instance, using data collected as it is being solved. They apply a learning-to-rank formulation and handcrafted features. Similarly, Alvarez et al. (2017) learns a regression model to predict the scores computed by Full Strong Branching for candidate variables, again using handcrafted features. Neither of these works report current state-of-the-art results for the learned policies. Neural Branching builds on the work by Gasse et al. (2019), which trains a Graph Convolutional Network to imitate decisions by Full Strong Branching. The main differences are 1) our use of the ADMM-based expert to scale up Full Strong Branching to large instances, and 2) extensive evaluation on real-world applications and significant performance improvements over SCIP.

More recently, Zarpellon et al. (2020) show that using a representation of the branch-and-bound tree (rather than that of just a single node) can improve the generalization of a learned branching policy. Gupta et al. (2020) obtain computational improvements similar to Gasse et al. (2019) in the setting of CPU-restricted machines by using an expensive graph network only at the root node of the branch-and-bound tree and a cheaper MLP elsewhere.

Other Learning Approaches for MIP Solvers: Learning has been used to automatically tune parameters for MIP solvers. *Algorithm Configuration* (Ansótegui et al. 2009, Hutter et al. 2009, 2011, Ansótegui et al. 2015) aims to find parameter values that improve aggregate performance on a dataset of instances. For example, Hutter et al. (2011) proposes Sequential Model-based Algorithm Configuration (SMAC), which iterates over three main steps: 1) evaluate the performance of selected solver parameter values on a set of instances, 2) learn a model to predict the performance of given parameter values from the evaluations done so far, and 3) use the model learned so far to select new candidate parameter values to evaluate. Results on CPLEX show that SMAC finds parameter values that are better than the default and the values found by CPLEX's tuning tool. *Algorithm Selection* (Kotthoff 2016) aims to improve solver parameter values for a specific instance. For example, Hutter et al. (2014) shows that it is possible to accurately predict the running time of MIP solvers as a function of the solver parameters and the instance being solved. Such a predictor can be used to improve the solver parameters for a specific instance. Note that learning techniques for Algorithm Configuration and Algorithm Selection are complementary to and can be combined with the learning of application-specific heuristics.

Learning for Combinatorial Optimization: Our work is an instance of the broader topic of learning to solve combinatorial optimization problems. Some of the earliest works in this area are Zhang and Dietterich (1995), Moll et al. (1999), Boyan and Moore (1997). More recently, deep learning has been applied to the Travelling Salesman Problem (Vinyals et al. 2015, Bello et al. 2016), Vehicle Routing (Kool et al. 2019, Nazari et al. 2018), Boolean Satisfiability (Selsam et al. 2019), and general graph-structured combinatorial optimization problems (Khalil et al. 2017a, Li et al. 2018). A survey of the topic is available by Bengio et al. (2018).

10. Conclusion

This work has demonstrated the long-held promise of machine learning to significantly improve MIP solver performance on both large-scale real-world application datasets and MIPLIB. We believe even bigger improvements are possible with further advances in models and algorithms.

Some promising future directions are:

- Learning to cut: Better selection and generation of cuts using learning is another potential source of performance improvements. Tang et al. (2020) provide evidence for the usefulness of this direction.

- Warm-starting models: The strong performance of learned models on MIPLIB suggests that it is possible to learn heuristics that work well across diverse MIPs. This can be used to overcome the 'cold-start' problem in applications where the amount of training data available early on in an application's life cycle may be too small to train good models. We can start by using models trained on heterogeneous datasets and use them as a bridge to more specialized models as more data is collected for the application.

- Reinforcement learning: Performance achieved using distillation or behavioral cloning is capped by the best expert available, while reinforcement learning (RL) can potentially exceed it. Efficient exploration, long range credit assignment, and computational scalability of learning are key challenges in applying RL to large-scale MIPs. Addressing them can lead to bigger performance improvements.

11. Acknowledgments

The authors would like to thank

- Alek Andreev, Matko Bosnjak, Elena Buchatskaya, Andrew Cowie, Vali Irimia, Vladimir Macko, Sona Mokra, Rahul Palamuttam, Alvaro Sanchez, and Kevin Waugh for their help in implementing various neural network models,
- David Applegate for advice on Neural Diving and providing the Google Production Packing dataset,
- Dave Helstroom for help with the Google Production Planning dataset,
- Jakob Bauer, Michael King, Andrea Michi, Richard Tanburn, and Sims Witherspoon for providing the Electric Grid Optimization dataset,
- Dj Dvijotham and Sven Gowal for providing the Neural Network Verification dataset, and
- Theophane Weber for proofreading the paper and suggesting improvements.

References

Achterberg T (2009) SCIP: solving constraint integer programs. *Mathematical Programming Computation* 1(1):1–41.

Achterberg T, Koch T, Martin A (2005) Branching rules revisited. *Operations Research Letters* 33(1):42 – 54.

Achterberg T, Koch T, Martin A (2006) Miplib 2003. *Operations Research Letters* 34(4):361–372.

Addanki R, Nair V, Alizadeh M (2020) Neural large neighborhood search. *Learning Meets Combinatorial Algorithms NeurIPS Workshop*.

Alvarez A, Louveaux Q, Wehenkel L (2017) A machine learning-based approximation of strong branching. *INFORMS Journal on Computing* 29:185–195, URL http://dx.doi.org/10.1287/ijoc.2016.0723.

Ansótegui C, Malitsky Y, Samulowitz H, Sellmann M, Tierney K (2015) Model-based genetic algorithms for algorithm configuration. *Proceedings of the 24th International Conference on Artificial Intelligence*, 733–739, IJCAI'15 (AAAI Press), ISBN 9781577357384.

Ansótegui C, Sellmann M, Tierney K (2009) A gender-based genetic algorithm for the automatic configuration of algorithms. Gent IP, ed., *Principles and Practice of Constraint Programming - CP 2009*, 142–157 (Berlin, Heidelberg: Springer Berlin Heidelberg).

Bain M, Sammut C (1995) A framework for behavioural cloning. *Machine Intelligence 15*, 103–129.

Balcan MF, Dick T, Sandholm T, Vitercik E (2018) Learning to branch. volume 80 of *Proceedings of Machine Learning Research*, 344–353 (Stockholmsmässan, Stockholm Sweden: PMLR), URL http://proceedings.mlr.press/v80/balcan18a.html.

Battaglia PW, Hamrick JB, Bapst V, Sanchez-Gonzalez A, Zambaldi VF, Malinowski M, Tacchetti A, Raposo D, Santoro A, Faulkner R, Gülçehre Ç, Song F, Ballard AJ, Gilmer J, Dahl GE, Vaswani A, Allen K, Nash C, Langston V, Dyer C, Heess N, Wierstra D, Kohli P, Botvinick M, Vinyals O, Li Y, Pascanu R (2018) Relational inductive biases, deep learning, and graph networks. *CoRR* URL http://arxiv.org/abs/1806.01261.

Bello I, Pham H, Le QV, Norouzi M, Bengio S (2016) Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* .

Bengio Y, Bengio S (2000) Modeling high-dimensional discrete data with multi-layer neural networks. Solla S, Leen T, Müller K, eds., *Advances in Neural Information Processing Systems*, volume 12, 400–406 (MIT Press), URL https://proceedings.neurips.cc/paper/1999/file/e6384711491713d29bc63fc5eeb5ba4f-Paper.pdf.

Bengio Y, Lodi A, Prouvost A (2018) Machine learning for combinatorial optimization: a methodological tour d'horizon.

Berthold T (2006) Primal heuristics for mixed integer programs. URL https://opus4.kobv.de/opus4-zib/files/1029/Berthold_Primal_Heuristics_For_Mixed_Integer_Programs.pdf.

Berthold T (2007) Rens-relaxation enforced neighborhood search .

Berthold T (2013) Measuring the impact of primal heuristics. *Operations Research Letters* 41(6):611–614.

Bishop CM (2006) *Pattern Recognition and Machine Learning (Information Science and Statistics)* (Berlin, Heidelberg: Springer-Verlag), ISBN 0387310738, URL `https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf`.

Boyan JA, Moore AW (1997) Using prediction to improve combinatorial optimization search. *In Proc. of 6th Int'l Workshop on Artificial Intelligence and Statistics.*

Boyd S, Parikh N, Chu E, Peleato B, Eckstein J (2011) Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning* 3(1):1–122.

Boyd S, Vandenberghe L (2004) *Convex optimization* (Cambridge university press).

Cheng CH, Nührenberg G, Ruess H (2017) Maximum resilience of artificial neural networks. D'Souza D, Narayan Kumar K, eds., *Automated Technology for Verification and Analysis*, 251–268 (Springer International Publishing).

Conrad J, Gomes CP, van Hoeve WJ, Sabharwal A, Suter J (2007) Connections in networks: Hardness of feasibility versus optimality. Van Hentenryck P, Wolsey L, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 16–28 (Berlin, Heidelberg: Springer Berlin Heidelberg).

Dantzig GB (1998) *Linear programming and extensions*, volume 48 (Princeton university press).

Diamond S, Boyd S (2016) Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research* 17(1):2909–2913.

Ding J, Zhang C, Shen L, Li S, Wang B, Xu Y, Song L (2020) Accelerating primal solution findings for mixed integer programs based on solution prediction. *AAAI*, URL `https://www.aaai.org/Papers/AAAI/2020GB/AAAI-DingJ.6745.pdf`.

Domahidi A, Chu E, Boyd S (2013) ECOS: An SOCP solver for embedded systems. *European Control Conference (ECC)*, 3071–3076.

Eckstein J, Bertsekas DP (1992) On the Douglas—Rachford splitting method and the proximal point algorithm for maximal monotone operators. *Mathematical Programming* 55(1-3):293–318.

Eckstein J, Nediak M (2007) Pivot, cut, and dive: A heuristic for 0-1 mixed integer programming. *Journal of Heuristics* 13(5):471–503.

FICO Xpress (2020) FICO Xpress optimization suite. URL `https://www.fico.com/en/products/fico-xpress-optimization`.

Gamrath G, Anderson D, Bestuzheva K, Chen WK, Eifler L, Gasse M, Gemander P, Gleixner A, Gottwald L, Halbig K, et al. (2020) The SCIP optimization suite 7.0 .

Gasse M, Chételat D, Ferroni N, Charlin L, Lodi A (2019) Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 15554–15566.

Geifman Y, El-Yaniv R (2019) Selectivenet: A deep neural network with an integrated reject option. *arXiv preprint arXiv:1901.09192* .

Glankwamdee W, Linderoth JT (2011) Lookahead branching for mixed integer programming. *ICS 2011.*

Gleixner A, Hendel G, Gamrath G, Achterberg T, Bastubbe M, Berthold T, Christophel PM, Jarck K, Koch T, Linderoth J, Lübbecke M, Mittelmann HD, Ozyurt D, Ralphs TK, Salvagnin D, Shinano Y (2019) MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, URL http://www.optimization-online.org/DB_FILE/2019/07/7285.html.

Gomes CP, van Hoeve WJ, Sabharwal A (2008) Connections in networks: A hybrid approach. Perron L, Trick MA, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 303–307 (Berlin, Heidelberg: Springer Berlin Heidelberg).

Goodfellow I, Bengio Y, Courville A (2016) *Deep Learning* (MIT Press), http://www.deeplearningbook.org.

Gupta P, Gasse M, Khalil E, Mudigonda P, Lodi A, Bengio Y (2020) Hybrid models for learning to branch. *Advances in neural information processing systems* 33.

Gurobi Optimization L (2020) Gurobi optimizer reference manual. URL http://www.gurobi.com.

Hansen P, Mladenović N, Pérez JAM (2010) Variable neighbourhood search: Methods and applications. *Annals of Operations Research* 175(1):367–407.

He H, Daume III H, Eisner JM (2014) Learning to search in branch and bound algorithms. Ghahramani Z, Welling M, Cortes C, Lawrence ND, Weinberger KQ, eds., *Advances in Neural Information Processing Systems 27*, 3293–3301 (Curran Associates, Inc.), URL http://papers.nips.cc/paper/5495-learning-to-search-in-branch-and-bound-algorithms.pdf.

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Hendel G (2018) Adaptive large neighborhood search for mixed integer programming. *Mathematical Programming Computation* Under review.

Hochreiter S, Schmidhuber J (1997) Long short-term memory. *Neural computation* 9(8):1735–1780.

Hottung A, Tierney K (2019) Neural large neighborhood search for the capacitated vehicle routing problem. *arXiv preprint arXiv:1911.09539* .

Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, 507–523, LION'05 (Springer-Verlag), ISBN 9783642255656.

Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009) Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.* 36(1):267–306, ISSN 1076-9757.

Hutter F, Xu L, Hoos HH, Leyton-Brown K (2014) Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* 206:79 – 111, ISSN 0004-3702.

IBM ILOG CPLEX (2019) V12.10: User's manual for CPLEX. URL `https://www.ibm.com/analytics/cplex-optimizer`.

Jünger M, Liebling TM, Naddef D, Nemhauser GL, Pulleyblank WR, Reinelt G, Rinaldi G, Wolsey LA (2009) *50 Years of integer programming 1958-2008: From the early years to the state-of-the-art* (Springer Science & Business Media).

Karp RM (1972) Reducibility among combinatorial problems. *Complexity of computer computations*, 85–103 (Springer).

Khalil E, Dai H, Zhang Y, Dilkina B, Song L (2017a) Learning combinatorial optimization algorithms over graphs. *Advances in Neural Information Processing Systems*, 6348–6358.

Khalil E, Le Bodic P, Song L, Nemhauser G, Dilkina B (2016) Learning to branch in mixed integer programming. Schuurmans D, Wellman M, eds., *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 724–731 (United States of America: Association for the Advancement of Artificial Intelligence (AAAI)), URL `http://www.aaai.org/Conferences/AAAI/aaai16.php`, aAAI Conference on Artificial Intelligence 2016, AAAI 16 ; Conference date: 12-02-2016 Through 17-02-2016.

Khalil EB, Dilkina B, Nemhauser GL, Ahmed S, Shao Y (2017b) Learning to run heuristics in tree search. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 659–666, URL `http://dx.doi.org/10.24963/ijcai.2017/92`.

Kingma D, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kingma DP, Welling M (2014) Auto-encoding variational bayes. Bengio Y, LeCun Y, eds., *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, URL `http://arxiv.org/abs/1312.6114`.

Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* .

Knueven B, Ostrowski J, Watson JP (2018) On mixed integer programming formulations for the unit commitment problem. *Optimization Online Repository* 2018, URL `http://www.optimization-online.org/DB_FILE/2018/11/6930.pdf`.

Kool W, van Hoof H, Welling M (2019) Attention, learn to solve routing problems! *International Conference on Learning Representations*, URL `https://openreview.net/forum?id=ByxBFsRqYm`.

Kotthoff L (2016) *Algorithm Selection for Combinatorial Search Problems: A Survey*, 149–190 (Cham: Springer International Publishing).

Land A, Doig A (1960) An automatic method of solving discrete programming problems. *Econometrica* 28(3):497–520.

Lawler EL, Wood DE (1966) Branch-and-bound methods: A survey. *Operations research* 14(4):699–719.

Lei Ba J, Kiros JR, Hinton GE (2016) Layer Normalization. *arXiv e-prints* arXiv:1607.06450.

Li Z, Chen Q, Koltun V (2018) Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in Neural Information Processing Systems*, 539–548.

Lions PL, Mercier B (1979) Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis* 16(6):964–979.

Lodi A, Tramontani A (2014) *Performance Variability in Mixed-Integer Programming*, chapter Chapter 1, 1–12.

Maher S, Miltenberger M, Pedroso JP, Rehfeldt D, Schwarz R, Serrano F (2016) PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. *Mathematical Software – ICMS 2016*, 301–307 (Springer International Publishing), URL `http://dx.doi.org/10.1007/978-3-319-42432-3_37`.

Mladenović N, Hansen P (1997) Variable neighborhood search. *Computers & operations research* 24(11):1097–1100.

Moll R, Barto AG, Perkins TJ, Sutton RS (1999) Learning instance-independent value functions to enhance local search. Kearns MJ, Solla SA, Cohn DA, eds., *Advances in Neural Information Processing Systems 11*, 1017–1023 (MIT Press), URL `http://papers.nips.cc/paper/1573-learning-instance-independent-value-functions-to-enhance-local-search.pdf`.

Nazari M, Oroojlooy A, Snyder L, Takac M (2018) Reinforcement learning for solving the vehicle routing problem. Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R, eds., *Advances in Neural Information Processing Systems*, volume 31, 9839–9849 (Curran Associates, Inc.), URL `https://proceedings.neurips.cc/paper/2018/file/9fb4651c05b2ed70fba5afe0b039a550-Paper.pdf`.

O'Donoghue B (2020) Operator splitting for a homogeneous embedding of the monotone linear complementarity problem. *arXiv preprint arXiv:2004.02177* .

O'Donoghue B, Chu E, Parikh N, Boyd S (2016) Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications* 169(3):1042–1068, URL `http://stanford.edu/~boyd/papers/scs.html`.

O'Neill R (2017) Computational issues in iso market models .

Pomerleau DA (1989) Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 305–313.

Ross S, Gordon G, Bagnell D (2011) A reduction of imitation learning and structured prediction to no-regret online learning. *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 627–635.

Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2009) The graph neural network model. *IEEE Transactions on Neural Networks* 20(1):61–80.

Schubert C (2017) Multi-level lookahead branching. Technical report, Institute of Mathematics, TU Berlin.

Selsam D, Lamm M, Bünz B, Liang P, de Moura L, Dill DL (2019) Learning a SAT solver from single-bit supervision. *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, URL `https://openreview.net/forum?id=HJMC_iA5tm`.

Shinano Y, Achterberg T, Berthold T, Heinz S, Koch T (2011) Parascip: a parallel extension of scip. *Competence in High Performance Computing 2010*, 135–148 (Springer).

Sierksma G, Zwols Y (2015) *Linear and integer optimization: Theory and practice* (CRC Press).

Song J, Lanka R, Yue Y, Dilkina B (2020) A general large neighborhood search framework for solving integer programs. *arXiv preprint arXiv:2004.00422* .

Taha HA (2014) *Integer programming: theory, applications, and computations* (Academic Press).

Tang Y, Agrawal S, Faenza Y (2020) Reinforcement learning for integer programming: Learning to cut. III HD, Singh A, eds., *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, 9367–9376 (Virtual: PMLR), URL `http://proceedings.mlr.press/v119/tang20a.html`.

Tjeng V, Xiao KY, Tedrake R (2019) Evaluating robustness of neural networks with mixed integer programming. *International Conference on Learning Representations*, URL `https://openreview.net/forum?id=HyGIdiRqtm`.

Vinyals O, Fortunato M, Jaitly N (2015) Pointer networks. Cortes C, Lawrence N, Lee D, Sugiyama M, Garnett R, eds., *Advances in Neural Information Processing Systems*, volume 28, 2692–2700 (Curran Associates, Inc.), URL `https://proceedings.neurips.cc/paper/2015/file/29921001f2f04bd3baee84a12e98098f-Paper.pdf`.

Wolsey L (1998) *Integer Programming.* Wiley Series in Discrete Mathematics and Optimization (Wiley), ISBN 9780471283669, URL `https://books.google.co.uk/books?id=x7RvQgAACAAJ`.

Xavier ÁS, Qiu F, Ahmed S (2020) Learning to solve large-scale security-constrained unit commitment problems. *INFORMS Journal on Computing* .

Xu K, Li C, Tian Y, Sonobe T, Kawarabayashi Ki, Jegelka S (2018) Representation learning on graphs with jumping knowledge networks. *International Conference on Machine Learning.*

Yang Y, Boland N, Dilkina B, Savelsbergh M (2020) Learning generalized strong branching for set covering, set packing, and 0-1 knapsack problems. Technical report, URL `http://www.optimization-online.org/DB_HTML/2020/02/7626.html`.

Yang Y, Boland N, Savelsbergh M (2019) Multi-variable branching: A case study with 0-1 knapsack problems. Technical report, URL `http://www.optimization-online.org/DB_HTML/2019/10/7431.html`.

Zarpellon G, Jo J, Lodi A, Bengio Y (2020) Parameterizing branch-and-bound search trees to learn branching policies. *arXiv preprint arXiv:2002.05120* .

Zhang W, Dietterich TG (1995) A reinforcement learning approach to job-shop scheduling. *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, 1114–1120, IJCAI'95 (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.), ISBN 1558603638.

12. Appendix

In the appendix we provide supplementary information which can be grouped into four strands:

1. An autoregressive model approach for Neural Diving, which produces promising initial results, but requires too much further investigation to warrant its inclusion in the main text.
2. Further data shedding light onto the previously described Neural Brancher work from a slightly different perspective, namely analysis results with respect to the number of nodes in the branch-and-bound tree, and also PAR-k metrics.
3. Additional details on the ADMM batch LP solver.
4. Extended description of the datasets we used throughout the paper and of the concept of calibrated time we used to mitigate the problems arising from trying to measure solving times on large clusters of heterogeneous machines.

12.1. Autoregressive Models for Neural Diving

Fairly simple conditionally-independent generative models may provide enough capacity for certain problems, especially when optimal solution is simple to predict. However, when the training set does not contain a dense region of high-quality solutions, such a model might ultimately fail to adequately express uncertainty and thus either learn to predict an overly narrow sub-region of the solution space or output spread probability mass over many infeasible solutions.

One well-proven way of overcoming limitations of conditionally-independent models is to model the desired distribution in auto-regressive way. Such a model would predict variable values sequentially based on some pre-defined ordering. We tried a number of simple extensions of the basic conditionally-independent model such as LSTM Hochreiter and Schmidhuber (1997) on top of the pre-computed variable node embedding, but found this kind of auto-regressive dependencies not expressive enough to provide a clear advantage over the conditionally-independent model.

Another direction we explored was to only provide auto-regressive dependencies through incrementally solving the underlying LP problem as we sample and assign variable values. Thus, we first compute the variable node embeddings $\{v_d\}_{d=1}^{D}$ (see section 3.2 for details). Then defining $\tilde{x}(x_{<d})$ to be an LP-solution of M with the first $d-1$ variables assigned to particular values, we predict the value of x_d in the following way:

$$p_\theta(x|M) = \prod_{d=1}^{D} p(x_d|x_{d-1},\ldots,x_1,M) = \prod_{d=1}^{D} \text{Bernoulli}(x_d|\mu_d), \tag{22}$$

$$\mu_d = \text{MLP}(v_d, \tilde{x}_d(x_{<d});\theta). \tag{23}$$

In practice, we do not re-compute the LP-solution after assigning each of the variables and only do so after every K ($K = 100$ in the experiment below) steps by warm-starting the LP solver with the last state of the previous solution process.

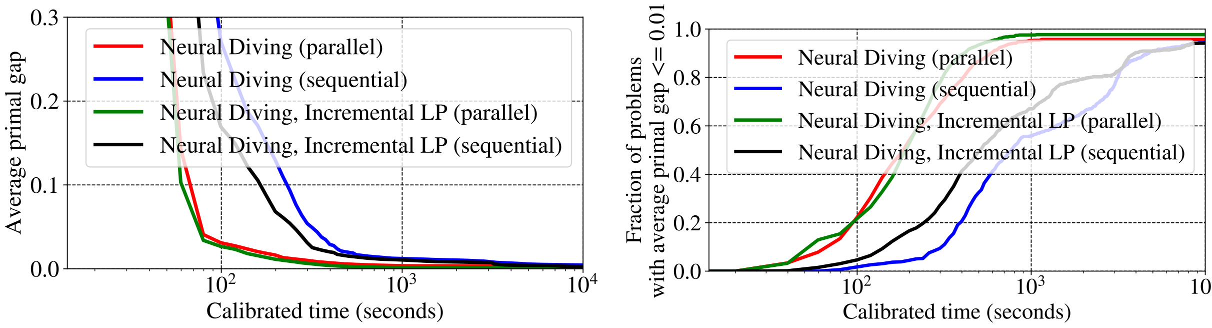


Figure 15 Comparison between the standard conditionally-independent GNN and its autoregressive version with incremental LP solver on Google Production Packing dataset.

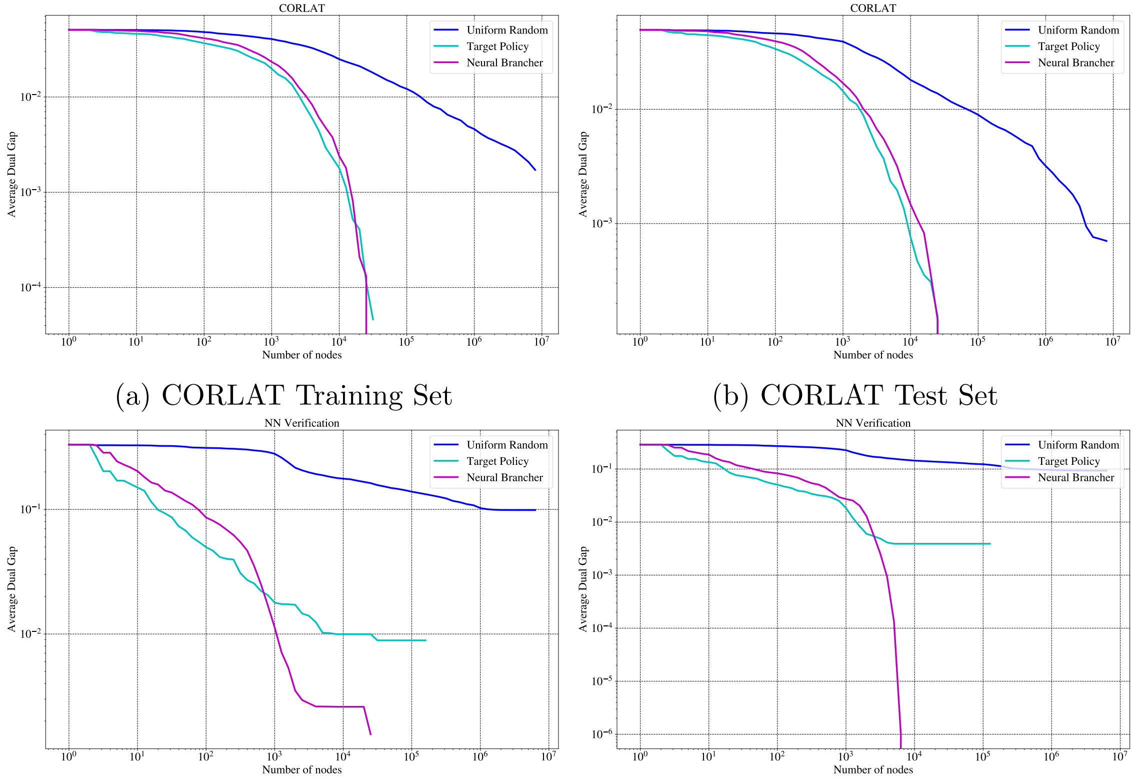
Figure 15 contains a comparison between the conditionally-independent and the autoregressive models on Google Production Packing dataset. One can see, that information encoded in incremenetal LP solutions does bring an improvement in terms of the average primal gap and the number of solutions solved to optimality. However, most of the gains brought by higher-quality assignments are outweighed by increased sampling time since sampling can no longer be done in parallel for each of the variables. These preliminary results indicate that there is potential in further architectural advances and leveraging more advanced features, which we leave for future work.

12.1.1. Ordering of variables In many domains, there is often a reasonably natural way to order variables in an autoregressive model. In the case of audio and other temporal sequences, time defines an order. In the case of images, raster scan ordering of pixels is commonly used.

In the case of integer programs, we need a way to order the binary variables. This ordering should be invariant to row and column permutations of the integer program since such changes do not affect the integer program. Intuitively we want a ”canonical” ordering that places variables that are ”similar” across instances at similar positions in the ordering so that autoregressive models can learn effectively. Unfortunately, there is no universal canonical ordering of variables that works well for all MIPs. Instead, we list several possible ordering methods as an additional hyperparameter, and pick the one that works best during hyperparameter sweeps. The orderings that we consider are:

1. **Input order**: We can proceed through the variables in the order as they are given in the MIP definition. Often, the ordering in which the variables are written down in the input already places ”similar” variables close to each other.

2. **Coefficient order**: We can sort variables by the coefficient by which they contribute to the objective. In some use cases, the objective coefficient of a variable encodes a semantic meaning of importance or priority, and we can hope to use this to e.g. process the most important variables first.



(a) CORLAT Training Set

(b) CORLAT Test Set

(c) Neural Network Verification Training Set

(d) Neural Network Verification Test Set

Figure 16 Comparison of the average dual gap achieved by the target (expert) policy and the learned policy on training and test sets for CORLAT and Neural Network Verification. We also include the uniform random policy as a baseline.

3. **Fractionality order**: We can order variables by their fractionality in the LP solutions. The fractionality of a variable is also often used to determine variable order in diving heuristics in the literature (see, e.g., section 3.1 of Berthold (2006)). Empirically we found this ordering to perform the best with autoregressive models.

12.2. Imitation Accuracy of the Learned Branching Policy

How well does the learned branching policy approximate the expert policy that it is trained to imitate? Figure 16 compares the average dual gap achieved by the expert policy and the learned policy on both the training and test sets for CORLAT and Neural Network Verification. We also include a policy that takes actions uniformly randomly as a baseline. The results show that imitation learning succeeds in accurately approximating the expert policy both on training and test sets. The learned policy is much closer to the expert than it is to the uniform random policy. The similarity in performance between training and test sets show that the learned policy generalizes well to unseen instances.

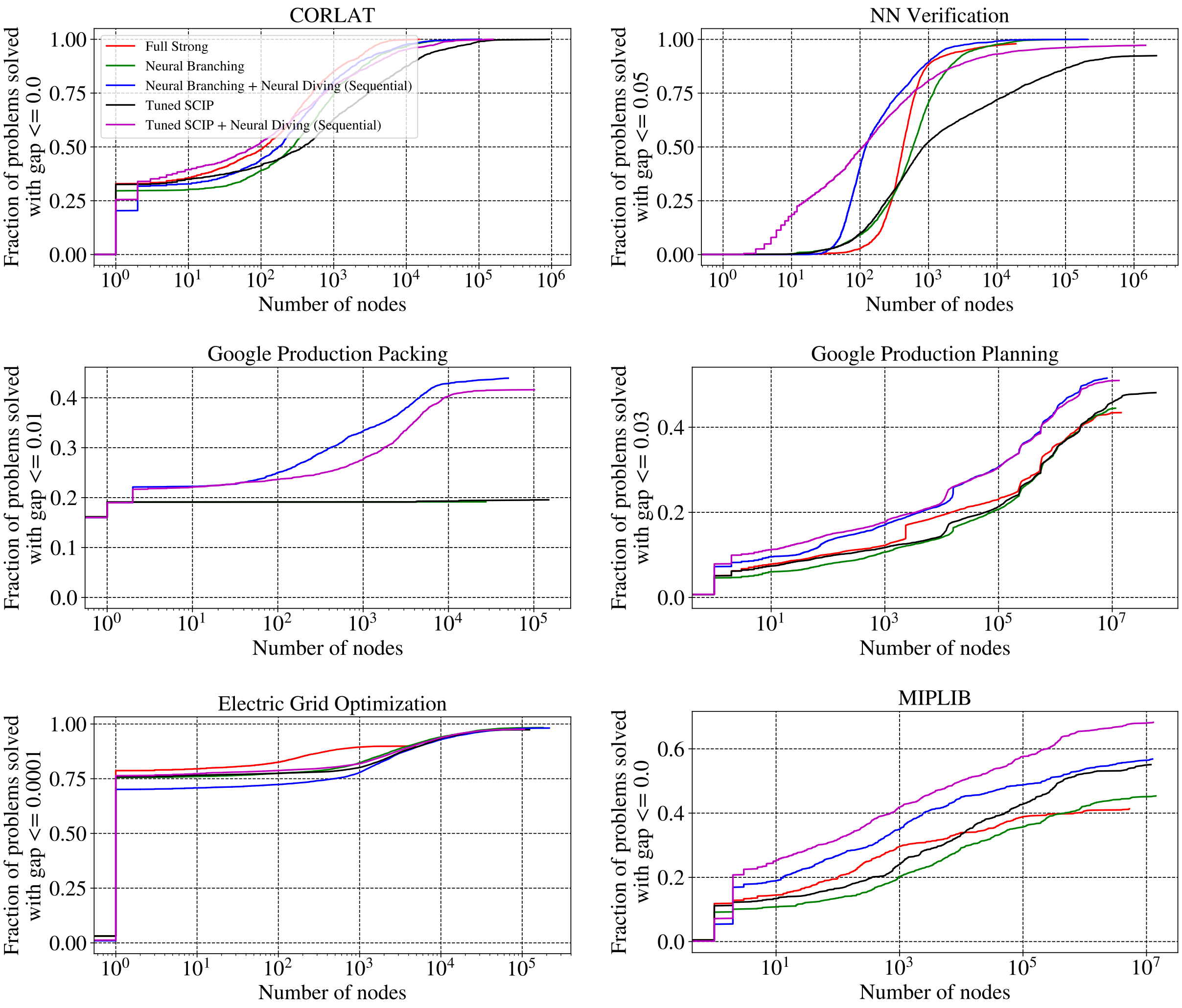


Figure 17 Survival plots with respect to number of nodes for various combinations of primal heuristics and variable selection policies as a function of running time on the benchmark datasets.

12.3. Evaluation Results with Respect to Number of Nodes

Sections 7.4 and 8 present evaluation results for Neural Branching and the combined Neural Branching + Neural Diving solver with respect to calibrated running time. Here we present the corresponding results with respect to number of nodes. They are shown in Figure 18. While running time is often the practically relevant resource metric, evaluating with respect to number of nodes can provide additional insights by removing the effect of different computational costs and hardware for the solvers being compared.

12.4. PAR-10 Results

In Table 3 we present PAR-10 results, PAR-10 refers to Penalized Average Running time with a penalty of 10, which averages over all instances the time taken to achieve the target optimality gap

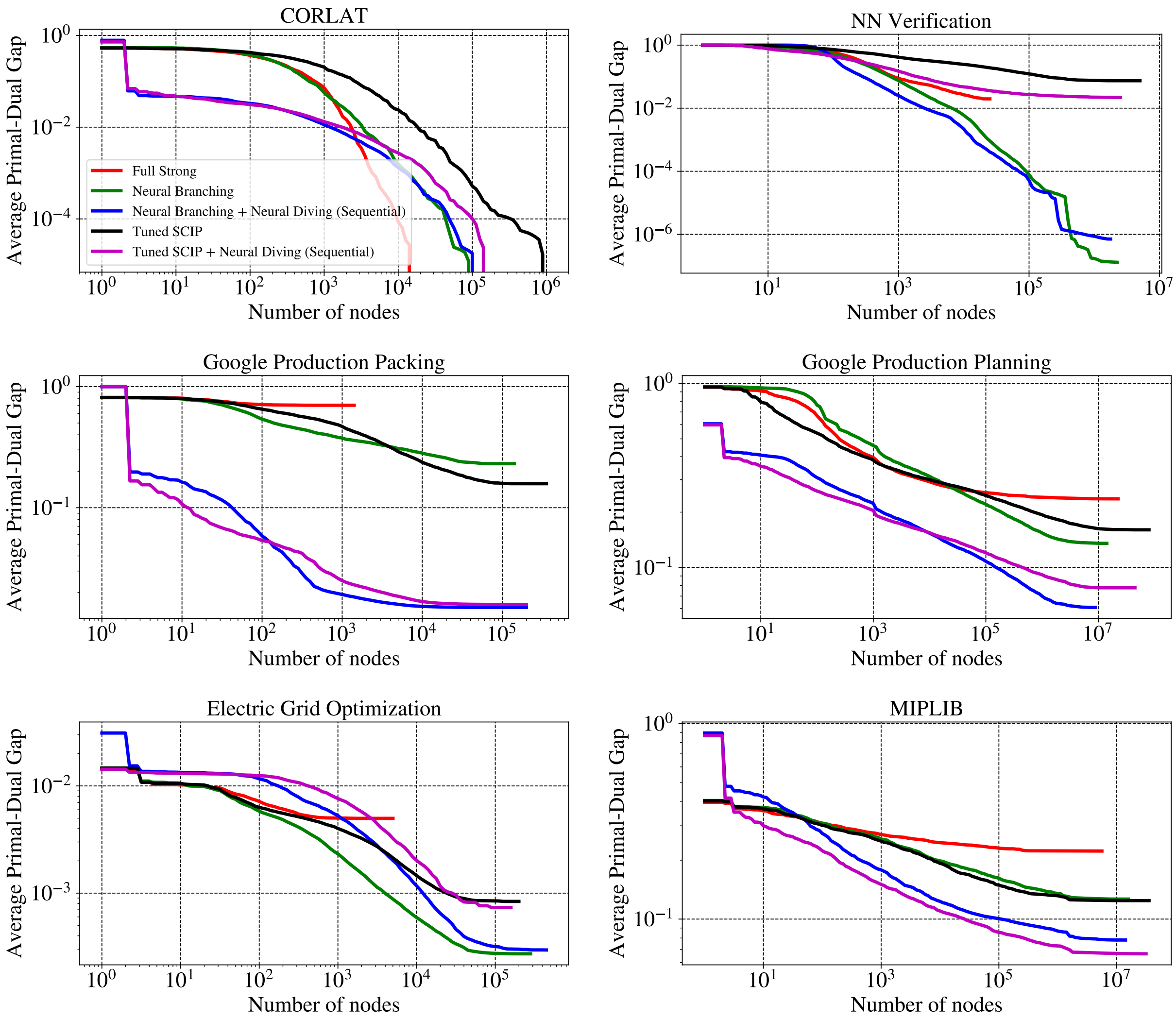


Figure 18 Average primal-dual gap achieved on the test set by various solvers as a function of the number of nodes.

or penalty multiplier times the time limit. In our setting we use calibrated time, and a penalty of 10 and a time limit of 10000.

As can seen in the Table 3 the winner for each dataset is an expert that contains a learned component, supporting our claim that learning is beneficial, the most frequent winner is Neural Branching with Neural Diving (Sequential).

Compared to Tuned SCIP, the winner per dataset has a PAR-10 result that is at least 28% smaller (except for Google Production Planning) than the Tuned SCIP one, with a special mention for the dataset of NN Verification, where the winner's PAR-10 is more than 700 times better than Tuned SCIP's PAR-10.

Solver	CORLAT	NN Verification	Google Production Packing	Electric Grid Optimization	Google Production Planning	MIPLIB
Full Strong	16.22	2375.76	80899.39	11223.37	57157.97	59332.45
Tuned SCIP	12.08	7812.81	80462.69	3697.81	52354.82	45685.67
Neural Branching	132.94	21.72	80856.83	**2311.02**	56459.14	55640.03
Tuned SCIP + Neural Diving (Sequential)	**5.29**	2813.33	58758.45	3279.35	49642.67	**32572.18**
Neural Branching + Neural Diving (Sequential)	6.01	**10.20**	**56738.14**	2450.80	**49524.11**	43989.33
The PAR-10 of the **Winner** divided by Tuned SCIP's	0.4379	0.0013	0.7051	0.6249	0.9459	0.71296

Table 3 PAR-10 results with respect to calibrated time. Lower is better. In **bold** is the best one for each dataset.

12.5. ADMM Batch LP Solver

In this section we give more details about the ADMM batch LP solver we use to generate the expert training data that the neural Branching is trained on. Consider the following composite convex optimization problem

$$\begin{aligned} &\text{minimize} \quad f(x)+g(z) \\ &\text{subject to } x=z \end{aligned}$$

over variables $x \in \mathbb{R}^n$, $z \in \mathbb{R}^n$ and where $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$, $g : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ are closed, proper, convex functions. The *Alternating direction method of multipliers* (ADMM) (Boyd et al. 2011, Lions and Mercier 1979, Eckstein and Bertsekas 1992) applied to this problem is the following procedure for any $\rho > 0$:

$$\begin{aligned} x^{k+1} &= \arg\min_x \left(f(x) + (\rho/2)\|x - z^k - \lambda^k\|_2^2\right) \\ z^{k+1} &= \arg\min_z \left(g(z) + (\rho/2)\|x^{k+1} - z - \lambda^k\|_2^2\right) \\ \lambda^{k+1} &= \lambda^k + x^{k+1} - z^{k+1}. \end{aligned}$$

Under benign conditions this procedure produces a sequence of iterates that converge to optimal objective values, *i.e.*, $f(x^k)+g(z^k) \to f(x^\star)+g(z^\star)$, and $\|x^k - z^k\| \to 0$, where $x^\star$ and $z^\star$ are optimal primal points (Boyd et al. 2011, §3.2). We can write the LP relaxation of MIP (1) in a form more amenable to ADMM:

$$\begin{aligned} &\text{minimize} \quad c^T x + \mathcal{I}_{Ax=y}(x,y) + \mathcal{I}_{[b_l,b_u]}(\tilde{y}) + \mathcal{I}_{[l,u]}(\tilde{x}) \\ &\text{subject to } x = \tilde{x}, \quad y = \tilde{y}, \end{aligned}$$

over variables $x, \tilde{x} \in \mathbb{R}^n$, $y, \tilde{y} \in \mathbb{R}^m$, and where $\mathcal{I}_\mathcal{X}$ denotes the convex indicator function of set $\mathcal{X}$, *i.e.*,

$$\mathcal{I}_\mathcal{X}(z) = \begin{cases} 0 & z \in \mathcal{X} \\ \infty & \text{otherwise.} \end{cases}$$

When ADMM is applied to this problem it yields the following algorithm with iterates $x, \tilde{x} \in \mathbb{R}^n$, $y, \tilde{y} \in \mathbb{R}^m$, $\lambda \in \mathbb{R}^{m+n}$:

$$\begin{aligned}
(x^{k+1}, y^{k+1}) &= \arg\min_{Ax=y} \left\{ c^T x + \rho/2 \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \tilde{x}^k \\ \tilde{y}^k \end{bmatrix} - \begin{bmatrix} \lambda_x^k \\ \lambda_y^k \end{bmatrix} \right\|_2^2 \right\} \\
\tilde{x}^{k+1} &= \Pi_{[l,u]}(x^{k+1} - \lambda_x^k) \\
\tilde{y}^{k+1} &= \Pi_{[b_l,b_u]}(y^{k+1} - \lambda_y^k) \\
\begin{bmatrix} \lambda_x^{k+1} \\ \lambda_y^{k+1} \end{bmatrix} &= \begin{bmatrix} \lambda_x^k \\ \lambda_y^k \end{bmatrix} + \begin{bmatrix} x^{k+1} \\ y^{k+1} \end{bmatrix} - \begin{bmatrix} \tilde{x}^{k+1} \\ \tilde{y}^{k+1} \end{bmatrix},
\end{aligned}$$

where $\Pi_{[l,u]}(z)$ denotes the Euclidean projection of z onto the interval $[l,u]$ elementwise. It turns out that only first step is computationally challenging since it involves solving a (potentially large) set of linear equations. Concretely, for some right-hand side $r \in \mathbb{R}^{m+n}$ we want to solve

$$(x^{k+1}, y^{k+1}) = \underset{Ax=y}{\arg\min} \left\{ c^T x + \rho/2 \left\| \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} r_x^k \\ r_y^k \end{bmatrix} \right\|_2^2 \right\}.$$

The solution to the which is given by

$$\begin{bmatrix} x^{k+1} \\ y^{k+1} \end{bmatrix} = \begin{bmatrix} I & A^T \\ A & -I \end{bmatrix}^{-1} \begin{bmatrix} r_x^k - c/\rho \\ r_y^k \end{bmatrix} + \begin{bmatrix} 0 \\ r_y^k \end{bmatrix}.$$

Note that the matrix to be inverted is the same *for all iterations*. Not only that, but the matrix does not depend on which variable we are branching on; it is the same for all LPs we need to solve at any given node. This means we can batch and solve all the LPs (or however many will fit into the GPU memory) together on the GPU simultaneously. This yields significant hardware benefits for both direct and indirect approaches to solving the linear system. Although GPUs are designed to provide very fast dense matrix-matrix routines the sparse matrix operations make less efficient uses of the GPU stream parallelism. However, by batching sparse matrix-vector operations into matrix-matrix, then it is easier to keep the GPU busy and make better use of the parallelism. We shall see that this yields very large speedups in practice.

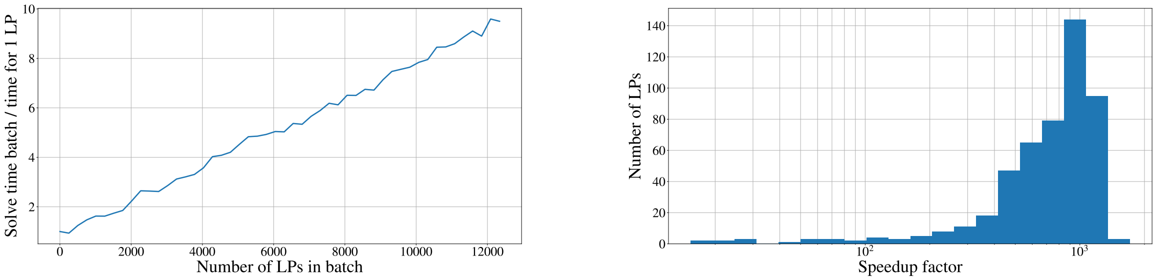


Figure 19 Left: Solve time vs number of LPs in a batch for the air05 MIP. Right: Histogram of speedup factors for all MIPs in MIPLIB at root nodes.

12.5.1. GPU speedups Here we present some speedup results for our batching strategy on a single GPU. We shall consider all possible LPs for all branching candidates at the root nodes of all the MIPs in the MIPLIB dataset. In all cases we ran the ADMM solver for 100 iterations and used the LP solution of the root node as a warm-start with which to initialize the ADMM solver. First, as a concrete example, consider the air05 binary MIP which at the root node (after presolve and any reductions that happens at the root node) has 6.1k variables, 426 constraints. The matrix that defines the problem is 98% sparse with 43.4k non-zeros. This high sparsity is not amenable to acceleration on GPU, however, when batched the solver can make much better use of the GPU parallelism. In Figure 19 we show the time required to approximately solve the LPs vs batch-size on GPU, normalized by time for (approximately) solving a single solver using the same solver. We show that we can compute all 12.2k branch scores in roughly 10 times the cost of a single solve, which is approximately a 1200× speedup from batching relative to solving all the LPs sequentially. As a point of comparison, for the same set of LPs ECOS (Domahidi et al. 2013) takes about 4.5 seconds to solve each one (which is approximately the same as SOPLEX, the built-in LP solver in SCIP, which takes 3 seconds to solve the LP at the root) so to solve all $12k$ LPs would require more than 15 hours. This is far too long for use in an industrial MIP solver and significantly larger than the ADMM batch LP solver, which finds the (approximate) solutions for all the LPs in 5.4 seconds. Next we show that this pattern holds true for the entire MIPLIB dataset. In Figure 19 we show the histogram of this speedup across the root nodes of the entire MIPLIB dataset. If there are n LPs to be solved at the root node, then we define the speedup factor to be

$$\text{speedup} = \frac{n \times (\text{time to solve 1 LP})}{\text{time to solve } n \text{ LPs in batch}}.$$

We remove the effect of the fixed costs (*e.g.*, transferring the data to the GPU) by subtracting the time taken to run zero iterations, which is non-zero, from the denominator and numerator (if we didn't remove this time it would artificially inflate the speedup factors we would observe).

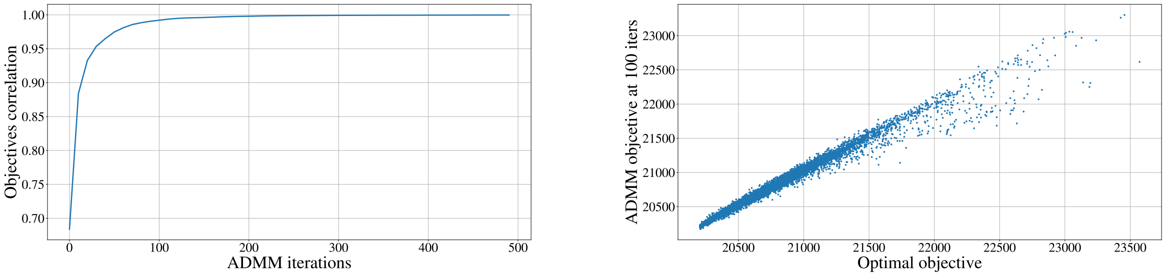


Figure 20 Correlation of ADMM objective and true objectives at root of air05 MIP.

This figure shows that the speedup is significant across the board, averaging approximately 800×, almost three orders of magnitude. Typically larger MIPs yield larger speedups, which also explains the sudden drop off at around 1000× speedup, since the largest MIPs eventually become too large for the GPU memory to handle.

12.5.2. Accuracy Warm-starting from the solution at the previous node means not many iterations are required for reasonable accuracy, which is a typical property of ADMM. This justifies our use of a relatively small number of ADMM iterations to 'solve' the problem. In Figure 20 we plot two quantities at the root node of the air05 MIP, at which we need to solve approximately $12k$ LPs. On the left hand side we plot the correlation of the $12k$ *true* objective values with the objective values returned by ADMM after 100 iterations. The true objective values were found by solving the same LPs using the high-accuracy ECOS interior point solver (Domahidi et al. 2013) with parser-solver CVXPY (Diamond and Boyd 2016). As we can see it converges towards one as the number of iterations increases, but even for a handful of iterations the approximate objective values are well correlated with the true objective values. Since the branch and bound decisions are made using the objective values only (see (19)) this high correlation indicates that we have a strong signal for the branch and bound decision with relatively little computational effort, and typically only low accuracy LP objective values are required for use in FSB (Achterberg et al. 2005). On the right we show the scatterplot of the true objective values and the objective values returned by ADMM after 100 iterations, and we can see the high correlation visually.

12.6. Details of Datasets

The target optimality gap used for each dataset in our evaluation as SCIP's stopping criterion for a MIP solve is given in table 4. In the case of Neural Network Verification, the actual criterion used in the application is to stop when the objective value becomes negative, but this is not expressible as a constant target gap across all instances. In order to treat all datasets consistently, we have selected a gap of 0.05.

Table 4 Target optimality gaps for the datasets in our evaluation used as the stopping criterion for SCIP.

Dataset	Target Optimality Gap
CORLAT	0
Neural Network Verification	0.05
Google Production Packing	0.01
Google Production Planning	0.03
Electric Grid Optimization	0.0001
MIPLIB	0

Table 5 Median of the number of constraints and variables (per type) for the different datasets before / *after* presolving using SCIP 7.0.1.

Dataset	Constraints	Variables	Binary	Integer (non-binary)	Continuous
CORLAT	470/*454*	466/*458*	100/*97*	0/*0*	366/*361*
NN Verification	6531/*1407*	7142/*1629*	171/*170*	0/*0*	6972/*1455*
Google Production Packing	36905/*24495*	10046/*8919*	3773/*3437*	0/*0*	6231/*5421*
Google Production Planning	11910/*478*	9884/*404*	833/*119*	462/*119*	8337/*136*
Electric Grid Optimization	61851/*45834*	60720/*58389*	42240/*42147*	0/*0*	18768/*16623*
MIPLIB	7706/*4388*	11090/*9629*	4450/*3816*	0/*0*	218/*96*

Figures 21 and 22 show the MIP sizes for the datasets used in our evaluation with and without presolving using SCIP 7.0.1. Note that, among the application-specific datasets, only Google Production Planning contains non-binary integer variables. It is also the most heterogeneous (along with MIPLIB) in terms of instance sizes. Table 5 summarizes the characteristics of those datasets before and after presolving with the SCIP 7.0.1 solver. The number of constraints and variables ranges different orders of magnitude across the datasets. It is worth noting that during presolving some instances might be deemed infeasible and, hence, dropped from the dataset.

We additionally report some key network statistics for the bipartite graphs that represent the resulting MIPs of the presolved instances. These include: a) the average degree across all nodes, b) the density of the bipartite graph, i.e. the number of present edges over the number of all possible edges between the nodes, c) the maximum degree centrality for the variables nodes, and d) the diameter of the graph, i.e. the longest among all shortest paths between any pair of nodes. The normalized histograms for the graph characteristics are presented in Figure 23.

As mentioned in the main paper, the above datasets were randomly split into training, validation and test sets with sizes 70%, 15% and 15%, respectively. This split is handled differently for MIPLIB as it constitutes an externally defined 'Benchmark Set'[1]. This set contains 240 MIPs on

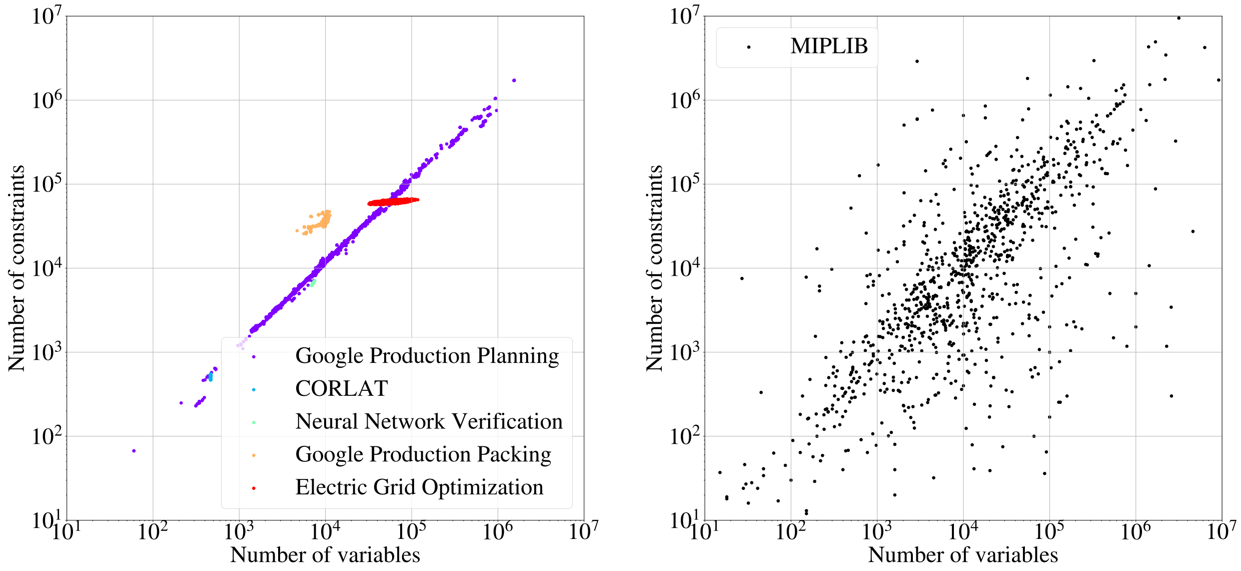


Figure 21 Original number of variables and constraints (before presolving) for the datasets used in our evaluation. Application-specific datasets are shown on the left, and MIPLIB is shown on the right.

which benchmarking results are reported for commercial and open source solvers and, therefore, constitutes our test set. The remaining instances in MIPLIB2017 and MIPLIB2010 correspond to our training and validation sets, respectively. Specifically, we use as the training set those instances from the MIPLIB2017 'Collection Set'[2] which do not belong to the 'Benchmark Set'. Similarly, we use as validation set the instances from MIPLIB2010 which do not belong to MIPLIB2017. This process yields 825 training, 155 validation and 240 test non-overlapping MIPs. The CORLAT dataset is open-source and can be accessed at `https://bitbucket.org/mlindauer/aclib2/src/master/`, while the MIPLIB dataset (with the aforementioned tags) can be found at `https://miplib.zib.de/download.html`.

12.7. Details of Calibrated Time

As explained in section 5, we use *calibrated time* to accurately measure running time of evaluation solve tasks on a shared, heterogeneous compute cluster. For each solve task, we periodically solve a small *calibration MIP* on a different thread as the solve task on the same machine. We define the speed of the machine to be

$$\text{Speed} = \frac{1}{\text{Wall clock time to solve calibration MIP}}. \tag{24}$$

For each periodic measurement of calibrated time, we estimate the speed K times and use the average. K is set to be the number of samples needed to estimate mean speed with 95% confidence,

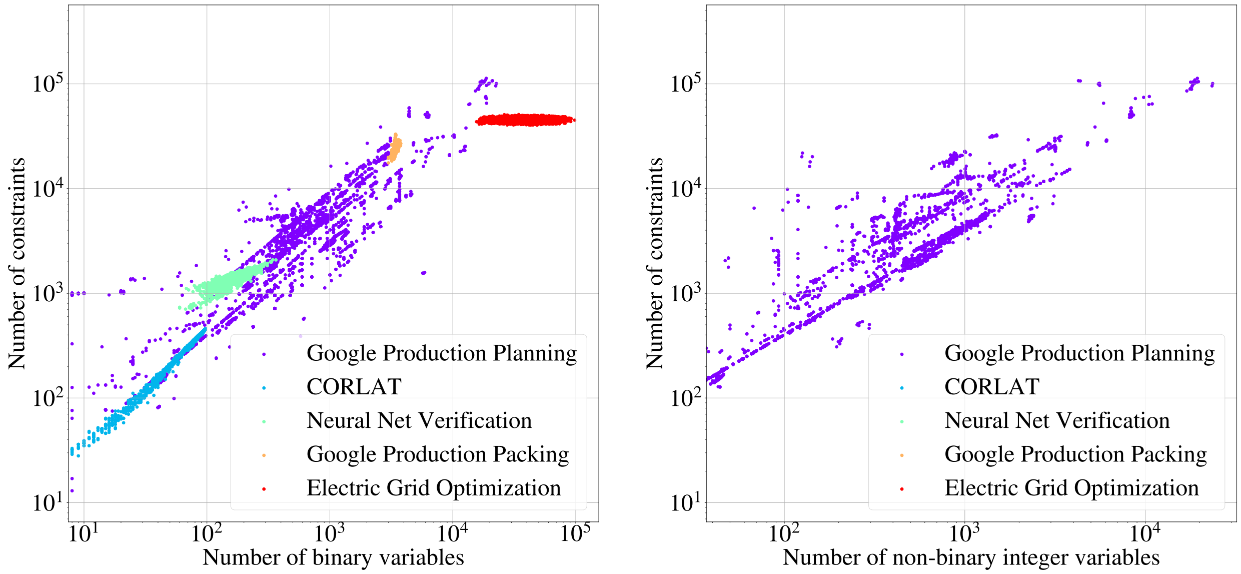


Figure 22 Number of binary variables (left) and non-binary integer variables (right) versus number of constraints after presolving using SCIP 7.0.1 for the application-specific datasets used in our evaluation.

with a minimum of 3 samples and a maximum of 30. The elapsed calibrated time $\Delta t_{\text{calibrated}}$ since the last measurement is

$$\Delta t_{\text{calibrated}} = \text{Speed} \times \Delta t_{\text{wallclock}}, \tag{25}$$

where $\Delta t_{\text{wallclock}}$ is the elapsed wallclock time since the last measurement. We use the MIP named *vpm2* from MIPLIB2003 Achterberg et al. (2006) as the calibration MIP.

Note that the above definition of calibrated time does not have a time unit. Instead it is (in effect) a count of the calibrated MIP solves during the evaluation solve task. To give it a unit of seconds, one can choose a reference machine with respect to which we want to report evaluation solve times, accurately measure the calibration MIP's solve time on it (without other tasks interfering), and multiply the calibrated time in equation 25 by the reference machine's estimated calibration MIP solve time. The resulting quantity has a unit of seconds. It can be interpreted as the time the evaluation solve task would have taken if it ran on the reference machine. We select Intel Xeon 3.50GHz CPU with 32GB RAM as the reference machine. We estimate the mean solving time of the calibration MIP *vpm2* for SCIP 7.0.1 on a single core to be 1.989 seconds (estimated from 1000 samples, collected after a warm up solve which is discarded). All the calibrated time results in the paper are expressed with respect to the reference machine in seconds.

To illustrate the benefit of calibrated time, table 6 presents results for four instances from MIPLIB 2017 (Gleixner et al. 2019) comparing 1) wallclock time measurement on a shared cluster machine, 2) calibrated time measurement on a shared cluster machine, and 3) wallclock time

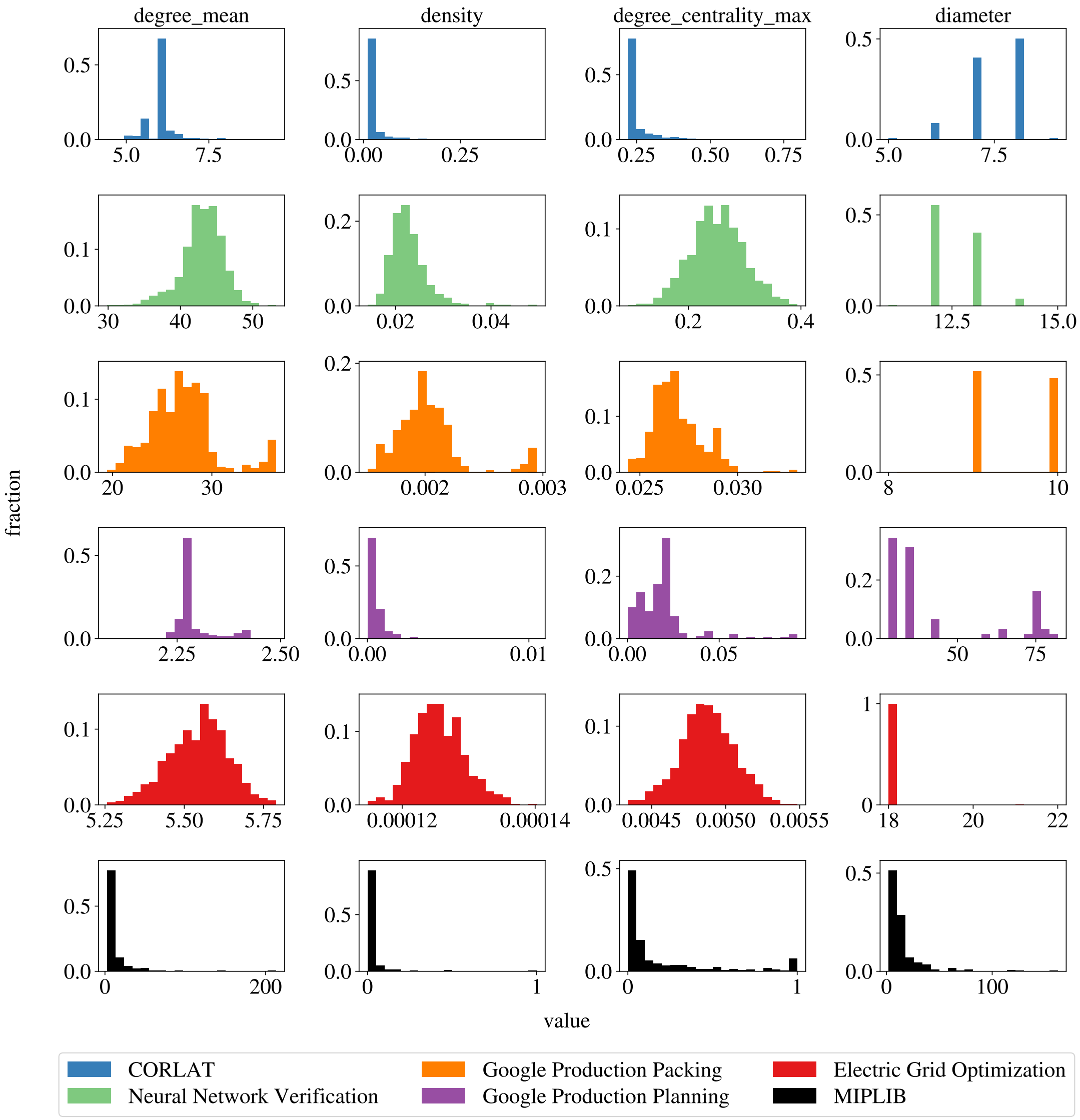


Figure 23 Network statistics for the presolved datasets.

measurement on the reference machine. We solve a MIP to optimality using SCIP 7.0.1 1000 times for each of these three settings. We then compute the *coefficient of variation* as the sample standard deviation divided by the sample mean for the 1000 measurements. Calibrated time reduces the coefficient of variation by about 30×, 8.3×, 7×, and 1.5×, on *air05*, *n5-3*, *swath1*, and *dano3_3*, respectively, compared to wallclock time measurements, and brings it closer to that of the reference machine.

Table 6 Coefficient of variation for different time measurements on MIPs from MIPLIB 2017.

MIP	Shared cluster wallclock time coeff. of var.	Shared cluster calibrated time coeff. of var.	Reference machine wallclock time coeff. of var.
air05	0.979	0.032	0.006
dano3_3	0.224	0.144	0.006
n5-3	0.318	0.038	0.007
swath1	0.623	0.087	0.010