Learning Fast Optimizers for Contextual Stochastic Integer Programs

Vinod Nair
DeepMind
vinair@google.com

Krishnamurthy Dvijotham
DeepMind
dvij@google.com

Iain Dunning
DeepMind
idunning@google.com

Oriol Vinyals
DeepMind
vinyals@google.com

Abstract

We present a novel reinforcement learning (RL) approach to learning a fast and highly scalable solver for a two-stage stochastic integer program in the large-scale data setting. Mixed integer programming solvers do not scale to large datasets for this problem class. Additionally, they solve each instance independently, without any knowledge transfer across instances. We address these limitations with a learnable local search solver that jointly learns two policies, one to generate an initial solution and another to iteratively improve it with local moves. The policies use contextual features for a problem instance as input, which enables learning across instances and generalization to new ones. We also propose learning a policy to compute a bound on the objective using dual decomposition. Benchmark results show that on test instances our approach rapidly achieves approximately 30% to 2000% better objective value, which a state of the art integer programming solver (SCIP) requires more than an order of magnitude more running time to match. Our approach also achieves better solution quality on seven out of eight benchmark problems than standard baselines such as Tabu Search and Progressive Hedging.

1 INTRODUCTION

Stochastic integer programming (Birge and Louveaux [1997], Shapiro et al. [2009]) arises in various applications that require combinatorial optimization under uncertainty, such as electric grid optimization, finance, and logistics. Many domains involve large-scale data for random variables (e.g., weather, demand, etc.), and require solving a set of related problem instances instead of only one instance (e.g., solve instances periodically, each with different weather and demand forecast distributions).

While powerful solvers exist for deterministic, single-instance mixed integer programs (MIPs), this is not the case for the stochastic, multi-instance setting. Traditional solvers from the optimization literature are not able to scale to high dimensional stochastic MIPs where several thousands of samples are required to represent uncertainty, and do not reuse experience on solving past optimization problems to solve future problems.

In this paper we consider learning an approximate solver with reinforcement learning (RL). We frame optimization as a sequential decision problem where at each step the solver "agent" proposes a solution, and the "environment" evaluates the objective value and any constraint violations to provide reward and observations. Contextual features describing a problem instance are used as an input to the solver so that its actions are adapted to that instance. The parameters of such a *contextual solver* are learned offline on a training set of instances. Once trained, the solver is applied to a new instance from the same distribution. We also learn a *dual policy* that computes a bound on the optimal value to estimate how far the solution computed by the contextual solver is from the global optimum for a given instance.

The advantages of our approach are: a) it can scale to a large set of problem instances, as well as a large set of samples for the random variables in each instance, and b) it can generalize to a new instance to achieve better solution quality and/or time than a solver that treats each instance independently. As the results show, our approach is able to scale to much larger datasets and achieve significant speedups compared to SCIP (Gleixner et al. [2017]), the state-of-the-art open source MIP solver.

Two-stage stochastic integer programs: We focus on an important class of stochastic programs called a two-stage stochastic MIP (Birge and Louveaux [1997]). In the first stage, a *planning decision* is optimized under uncertainty, then the values of all random variables are observed, and in the second stage a *recourse decision* is

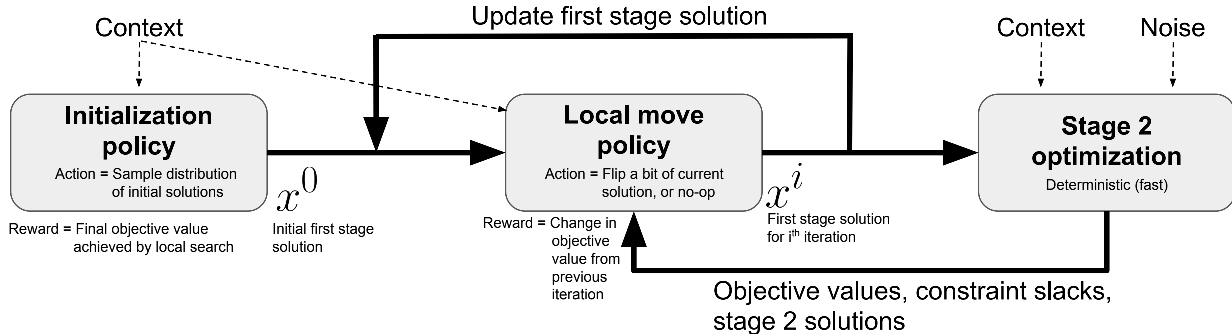


Figure 1: Local search for a two-stage stochastic MIP as an RL problem. The initialization and local move policies are part of the "agent", while the second stage optimization is part of the "environment." Given a problem instance specified by the context, the agent's "action" is to propose a first stage solution at each step of local search. The environment evaluates it on a minibatch of noise samples (denoted by ω in the text) by optimizing the second stage problem for each sample using a MIP solver to compute the reward and observations.

deterministically optimized to adapt the plan in repsonse to observations. The uncertainty is exogenous, so it is independent of the first stage solution. This problem structure arises in many applications, such as electric grid day-ahead planning (Zheng et al. [2015]) and logistics (Laporte et al. [1992]).

Local search solver: The contextual solver is structured as a local search algorithm over the space of first stage solutions. It consists of two learnable components: a) an *initialization policy* that takes as input the context vector and outputs an initial solution, and b) a *local move policy* that takes the initial solution and applies a sequence of local moves such that the final solution has high objective value. Both of these policies are learned jointly.

Both during training and testing, evaluation of a first stage solution requires solving several second stage optimization problems to approximate the expected objective as a sample average. Since each such problem is a deterministic MIP, they are well-suited for solving with an off-the-shelf MIP solver, so we use SCIP to solve them. This provides another perspective of our work as a hybrid approach that combines a learned contextual solver with a MIP solver in the learning loop to efficiently optimize stochastic two-stage MIPs.

Contributions: We develop an approach to learn 1) a contextual local search solver with the initialization and local move policies that computes a solution to the optimization for a given problem instance, and 2) a contextual dual policy that computes an upper bound on the optimal value (achievable by *any algorithm*) for a given problem instance. We benchmark our approach on a set of two-stage stochastic MIPs and show that compared to SCIP it achieves approximately 30% to 2000% better objective value for the same running time and at least an order of magnitude faster for the same objective value. It also outperforms baseline algorithms such as Tabu Search and Progressive Hedging on seven out of eight problems.

2 RELATED WORK

Learning solvers: Early examples of learning solvers for combinatorial optimization problems include Zhang and Dietterich [1995], Moll et al. [1999], and Boyan and Moore [2001]. The first two use TD(λ) to learn a value function over the solution space such that applying a local search algorithm with that value function can find a good solution and can generalize to new problem instances. These approaches assume that an initial solution is given (either randomly generated or by another algorithm), whereas our approach learns to generate the initial solution jointly with the local search policy. Boyan and Moore proposed STAGE, an iterative algorithm that alternates between learning a value function over the solution space for one instance and using that function to find a better solution for that instance. This work was primarily focused on the single instance setting.

More recently Vinyals et al. [2015] used supervised learning with pointer networks to approximately solve travelling salesman problems. Subsequent work improved on it by using the same pointer network architecture, but learned with RL (Bello et al. [2016]). Ignoring architectural differences, our initialization policy is similar to Bello et al.'s approach, but we further improve the initial solution using a learned local move policy. As our results show, this is a crucial difference for improving performance. Khalil et al. [2017a] combine graph neural networks with RL to learn a greedy solver for graph combinatorial optimization problems that incrementally extends a partial solution till its complete. Again, our idea of improving the complete solution further with a learned local search policy can be applied here as well.

Additionally, using the dual policy, we can compute a bound on the objective function value even on new problem instances. This allows us to a) assess the quality of the solution produced by the learned solver, and b) to make a fairer comparison to approaches that provide both a solution and a bound since proving the bound can be a significant part of the solver running time.

Learning to improve solvers: Several works have focused on applying learning to improve the decisions made by non-learning solvers, e.g., variable selection and node selection decisions in a branch and bound solver. See Lodi and Zarpellon [2017] for a survey. Khalil et al. [2016] treat variable selection as a ranking problem and learn a ranking function. Khalil et al. [2017b] use learning to predict on which nodes of the branch and bound search tree various primal heuristics will succeed. Such approaches can be used as building blocks for an end-to-end solution.

3 CONTEXTUAL TWO-STAGE STOCHASTIC INTEGER PROGRAM

A contextual two-stage stochastic integer program is defined as follows (for the case of maximization):

$$\max_x f(x; z) + E_{P(\omega)}\left[\max_{y \in Y(x,\omega)} g(y; x, \omega, z)\right] \quad (1)$$

where

- $\omega \in \mathbb{R}^d$ is a random variable representing the uncertainty affecting the second stage objective function and constraints. ω is independent of the decision variables. We are given a set of I.I.D. samples $D_\omega = \{\omega_i \sim P(\omega)\}, i = 1, ..., N_\omega$ with which the expectation in equation 1 is estimated as a sample average.
- $x \in \{0, 1\}^n$ is the first stage decision variable, optimized before observing ω. We assume that every x is feasible.
- $y \in \{0, 1\}^m$ is the second stage decision variable, optimized after observing ω, where $Y(x, \omega) \subseteq \{0, 1\}^m$ is the feasible set defined by second stage constraints. We assume $Y(x, \omega) \neq \emptyset$ for all (x, ω) (i.e., *complete recourse*).
- $z \in \mathbb{R}^c$ is the context feature vector describing an instance of the optimization problem. A set of instances is generated by drawing I.I.D. samples $D_z = \{z_j \sim P(z)\}, j = 1, ..., N_z$.

f and g represent the first and second stage objective functions, respectively. Unlike LP relaxation based approaches to solving MIPs, our approach can be applied to nonlinear f, g, and constraints. However, in this work we focus on the linear case to allow a direct comparison to LP relaxation based approaches:

$$f(x; z) = c_1^T(z)x$$
$$g(y; x, \omega, z) = c_2^T(x, \omega, z)y$$
$$Y(x, \omega) = \{y \in \{0, 1\}^m : B(\omega, z)x + C(\omega, z)y \leq d(\omega, z)\}$$

Sample Average Approximation: The primary approach in the optimization literature for solving equation 1 is the *Sample Average Approximation* (SAA) method (Ahmed and Shapiro [2002]). It replaces the expectation with a sample average and replicates the second stage decision for each sample:

$$\max_x f(x; z) + \frac{1}{N_\omega}\sum_i^{N_\omega}\left[\max_{y_i \in Y(x,\omega_i)} g(y_i; x, \omega_i, z)\right].$$

This converts the original problem into a deterministic one which can be solved using a deterministic MIP solver (assuming linear f, g, and constraints). The key disadvantage is that the number of second stage decision variables increases by a factor of the number of samples N_ω. As a result, the approach does not scale beyond a moderate number of variables and samples. In practice this limitation is overcome by domain experts carefully selecting a small set of samples that they judge will likely affect the solution. By being more scalable, our learning-based approach makes such expert selection unnecessary.

4 RL APPROACH

We describe our Reinforcement Learning (RL) approach for learning a solver for the optimization problem defined in section 3.

4.1 MAIN IDEA

We formulate our approach as *learning an iterative local search algorithm* over the space of first stage solutions (see figure 1). It has two learnable components which are optimized jointly:

- *Initialization policy:* Given an instance of the optimization problem, generate an initial first stage solution x^0 from which local search starts.
- *Local move policy:* At each iteration of local search, select one of the neighbors of the current first stage solution as the proposal for the next iteration.

The solution x^K after a fixed number of iterations K is the solver's output. The objective function value r^K at

x^K is approximated by:

$$r^K = f(x^K; z) + \frac{1}{N_\omega} \sum_{i=1}^{N_\omega} \left[\max_{y \in Y(x^K, \omega_i)} g(y; x^K, \omega_i, z) \right] \tag{2}$$

where the expectation is approximated as a sample average. For a given x and ω, the second stage optimization over y is deterministic. We assume that an efficient solver is available (e.g., SCIP, or a fast heuristic) so that an accurate approximation of the expectation can be computed quickly. This allows us to leverage existing efficient techniques for deterministic optimization to learn a solver for stochastic optimization.

Limitations: 1) Our approach relies on offline learning on a dataset of instances, and it is unlikely to be useful in a setting where there is only one instance (or few) to be solved. Our early experiments showed that in the single-instance setting (i.e., treat z as a constant), our approach's running times tend to be much higher than baseline algorithms to achieve similar solution quality. 2) The solution found after K iterations is not guaranteed to be optimal. But if learning is successful, it can be expected to be a good solution, and the bound provided by the dual policy indicates how close to optimal it is.

4.2 INITIALIZATION POLICY

The initialization policy is a distribution $P(x|z)$ over the first stage solution x given the context feature vector z. The initial solution is generated by sampling $x^0 \sim P(x|z)$. Reward is the expected objective function value achieved by the solution at the end of local search. We use policy gradients to learn $P(x|z)$.

Architecture: We explored two architectures: 1) a conditional autoregressive generative model, and 2) a simpler, fully connected feedforward neural network with a single hidden layer. Results are better for the former, so we describe it here. Autoregressive generative models provide a flexible neural network parameterization for high dimensional distributions while still allowing tractable exact evaluation of the log probability and its gradient with respect to model parameters. Crucially, they do not make any independence assumptions, thus capturing complicated correlations in the vector x. Here we consider a conditional version

$$P_{\theta_{init}}(x|z) = \prod_i P_{\theta_{init}}(x_i | x_{<i}, z) \tag{3}$$

where x_i is the ith dimension of x, $x_{<i}$ is the set of first $i-1$ dimensions of x, and θ_{init} denotes the model's parameters. Specifically, we use the Neural Autoregressive Density Estimator (NADE) (Larochelle and Murray [2011]). NADE is not specialized for a particular type of data (e.g., images, time series), which makes our approach application agnostic. It uses weight sharing across the conditional distributions in equation 3 to achieve a compact parameterization. For details see Larochelle and Murray [2011].

Training: The loss function for learning θ_{init} is

$$\mathcal{L}(\theta_{init}) = -E_{P_{\theta_{init}}(x^0|z)P(r^K|x^0,z)} \left[r^K \right]. \tag{4}$$

θ_{init} is learned with stochastic gradient descent using the policy gradient method of REINFORCE (Williams [1992]). The gradient is given by:

$$\nabla_{\theta_{init}} \mathcal{L}(\theta_{init}) = - E \left[(r^K - b_w(z)) \nabla_{\theta_{init}} (\log P_{\theta_{init}}(x|z)) \right]$$

where $b_w(z)$ is a learned baseline function parameterized by w, and the expectation is with respect to $P_{\theta_{init}}(x^0|z)P(r^K|x^0, z)$. The baseline $b_w(z)$ is a single hidden layer, fully connected feedforward neural network with ReLU hidden units. Its weights w are learned jointly with the initialization policy by stochastic gradient descent to minimize $(r^K - b_w(z))^2$.

4.3 LOCAL MOVE POLICY

We learn a policy that makes K local moves starting from the initial solution x^0 to produce x^K. At an intermediate step k, the local move policy takes the first stage solution x^k and selects one of its neighbors from a Hamming ball of radius 1 as x^{k+1}. This results in $n+1$ possible actions per step: either toggle one of the dimensions of x^k, or a no-op. The policy outputs a categorical distribution $\pi_{\theta_{lm}}(a^k | x^k, s^k, x^0, z)$ over the actions, given state s^k (described below), initial solution x^0, and context vector z, parameterized by θ_{lm}. The reward at each iteration is the change in objective value from the previous iteration. So the policy is being learned to maximize the total expected increase in objective value relative to the initial solution in an episode. We use the Asynchronous Advantage Actor Critic (A3C) algorithm (Mnih et al. [2016]) for learning.

Architecture: $\pi_{\theta_{lm}}(a^k | x^k, s^k, x^0, z)$ is a two hidden layer, fully connected feedforward neural network with ReLU hidden units. The observations o^k at step k consists of:

- *Constraint slacks:* The difference between a second stage constraint's value and its upper/lower bounds obtained for each sample ω_i.
- *Second stage solutions:* The solutions y^i obtained for each sample ω_i.

We define state s^k to be a fixed size time window of J observations $\{o^{k-1}, ..., o^{k-J}\}$. The window size is a hyperparameter tuned based on validation set performance. A3C also learns a value function $V_{\theta_v}(s_k; x_0, z)$ jointly with the policy. Following standard practice, this function shares with the policy all the hidden layers and corresponding parameters, with unshared parameters only in the output layers to compute the value and policy outputs.

Training: The reward at step k of an episode is given by

$$r_{lm}^k = r^{k+1} - r^k,$$

where r^k is defined by equation 2. A3C defines its loss in terms of the *advantage function*:

$$A(a^k, s^k; x^0, z) = \sum_{l=0}^{L-1} \gamma^l r_{lm}^{k+l} + \gamma^L V_{\theta_v}(s_{k+L}; x_0, z) - V_{\theta_v}(s_k; x_0, z),$$

where the first two terms on the RHS give an estimate of the L-step return, and γ is the discount factor. The loss is given by

$$\mathcal{L}(\theta_{lm}, \theta_v) = -E_{\pi_{\theta_{lm}}(a^k|x^k, s^k, x^0, z)} \left[A(a^k, s^k; x^0, z) \right], \tag{5}$$

with the gradient with respect to θ_{lm} given by:

$$\nabla_{\theta_{lm}} \mathcal{L}(\theta_{lm}, \theta_v) = -E \left[A(a^k, s^k; x^0, z) \nabla_{\theta_{init}} (\log \pi_{\theta_{lm}}(a^k|x^k, s^k, x^0, z) \right].$$

The square of the advantage function is an additional loss used to learn the value function parameters θ_v. We also apply entropy regularization to encourage exploration by preventing the policy from becoming deterministic. The relative magnitude of these losses' contribution to the total loss are tuned as hyperparameters based on validation set performance.

We use distributed TensorFlow with N_w parallel workers executing episodes and computing gradients, and one central learner asynchronously applying gradients to update parameters. This setup is replicated for each of the N_h randomly sampled hyperparameter settings for hyperparameter tuning. We use $N_w = 20$ and $N_h = 25$ for each problem in our benchmark.

5 BOUNDS VIA LEARNED DUAL POLICIES

An attractive feature of MIP solvers is that they provide an upper bound on the optimal value of a maximization problem, quantifying the objective value gap between a solution and the global optimum. However, as explained in section 3, directly applying a MIP solver to a two-stage stochastic integer program with a large set of samples is computationally infeasible. We use *dual decomposition* (Carøe and Schultz [1999]) to develop an alternative approach.

The main idea is that if we are allowed to choose a different $x = x_\omega$ for each sample ω, the optimization problem (1) decomposes into independent subproblems for each sample that can be solved in parallel. This constitutes a relaxation of the optimization problem, and hence its optimal value is an upper bound on the original problem. Solving the relaxation only requires a deterministic MIP problem, which can be solved efficiently or bounded using a further LP relaxation.

This bound can be tightened by adding Lagrangian multipliers that encourage the scenario subproblems to agree on a single $x = x_\omega$. Since we are interested in large-scale stochastic MIPs and in out-of-sample tests, it is not appropriate to compute an independent dual variable for each scenario ω. Instead, we train a neural network that, given features that depend on ω (the sample) and z (the context), predict the dual variables for the problem. Given any such policy, we can obtain a bound on the value of the 2-stage stochastic MIP and further, we can evaluate this bound out of sample (on samples that were unavailable during training) to estimate the "generalization" of the bound.

5.1 DUAL DECOMPOSITION OF 2-STAGE STOCHASTIC MIPs

Consider equation 1 in the linear case:

$$\max_{x \in \{0,1\}^n} c_1(z)^T x + \frac{1}{N_\omega} \sum_{\omega \in D_\omega} \left[\max_{y \in Y(x,\omega,z)} c_2(\omega_i, z)^T y \right]. \tag{6}$$

We now introduce independent copies x_ω for each sample ω with constraints to enforce equality of x_ω:

$$\max_{x, \{x_\omega\}} \frac{1}{N_\omega} \sum_{\omega \in D_\omega} \left[\max_{y \in Y(x_\omega,\omega,z)} c_1(z)^T x_\omega + c_2(\omega, z)^T y \right], \quad \text{s.t } x_\omega = x \quad \forall \omega \in \Omega.$$

We drop the constraints and add a Lagrangian term to obtain a relaxation:

$$\max_{x, \{x_\omega\}} \frac{1}{N_\omega} \sum_{\omega \in D_\omega} \left[\max_{\substack{y \in \\ Y(x_\omega,\omega,z)}} c_1^T x_\omega + c_2^T y + \lambda_\omega^T (x - x_\omega) \right], \tag{7}$$

where λ_ω for each ω is a dual variables, and we dropped the dependence of c_1, c_2 on ω, z for brevity.

From equation 7 we can derive the following *dual problem* to optimize the upper bound (see supplementary material for details):

$$\min_{\{\lambda_\omega\}} \frac{1}{N_\omega} \sum_{\omega \in D_\omega} h(\omega, z, \lambda_\omega) + \mathbf{1}^T \max\left(\frac{1}{N_\omega} \sum_{\omega \in D_\omega} \lambda_\omega, 0\right), \tag{8}$$

where

$$h(\omega, z, \lambda_\omega) = \max_{y \in Y(x_\omega, \omega, z)} c_1(z)^T x_\omega + c_2(\omega, z)^T y - \lambda_\omega^T x_\omega.$$

We use (8) to estimate a bound.

5.2 LEARNING DUAL POLICIES

There are three issues with applying the above dual decomposition approach to large scale contextual stochastic MIPs:

- *Context ignored*: If we directly apply dual decomposition, even after solving the problem for thousands of context vectors z, we would start from scratch for a new context. This is wasteful particularly if the solutions for the dual variables are simple functions of z that can be learned.
- *Generalization*: The problem (8) only implies a bound for the set of scenarios in the training set D_ω. However, this does not say anything about how the bound generalizes to unseen samples.
- *Computation*: The problem (8) is expensive to solve if the number of scenarios D_ω is large, since one has to solve an optimization problem (to compute h) for each scenario ω simply to evaluate the objective and compute gradients with respect to λ_ω.

We address all three issues by learning a *dual policy* $\lambda_\omega = \lambda_{\theta_d}(\omega, z)$ that maps from the context z and sample features ω to the dual variables λ_ω, parameterized by θ_d. We train the policy to minimize the dual objective averaged over samples D_ω and context sample set D_z:

$$\min_\theta \frac{1}{N_z} \sum_{z \in D_z} \frac{1}{N_\omega} \sum_{\omega \in D_\omega} h\left(\omega, z, \lambda_{\theta_d}(\omega, z)\right) \tag{9a}$$

$$+ \mathbf{1}^T \max\left(\frac{1}{N_\omega} \sum_{\omega \in D_\omega} \lambda_{\theta_d}(\omega, z), 0\right). \tag{9b}$$

A major bottleneck in solving this problem is the need to compute h (which itself involves solving a deterministic MIP or an LP relaxation of it). Thus, it is desirable to have an algorithm that does not compute h for each z, ω at each iteration.

In our experiments we sample z^s, ω^s but also get predictions of the neural network for each $\omega \in D_\omega$ to compute $\sigma = \sum_{\omega \in D_\omega} \lambda_\omega(\omega, z^s; \theta)$. Then, an unbiased estimate of a (sub)-gradient of the objective is given by

$$g^s = \frac{\partial h}{\partial \lambda_\omega}(\omega^s, z^s) \frac{\partial \lambda_\omega(\omega^s, z^s; \theta)}{\partial \theta} + \text{sign}(\sigma) \frac{\partial \lambda_\omega(\omega^s, z^s; \theta)}{\partial \theta} \tag{10}$$

This only requires one backprop through the network and N_ω forward passes and computation of h for a single z, ω.

Architecture: We use a simple feedforward architecture with a single hidden layer in all our experiments. The architecture has fully connected layers with ReLU activations. We add a skip connection from the context vector z to the hidden and output layer and a skip connection from ω to the output layer. We tune the size of the hidden layer and learning rate as hyperparameters (picked so that the dual bound on a validation set is minimized).

6 EVALUATION

6.1 BENCHMARK

There is a dearth of standardized, large-scale benchmarks for stochastic programming problems. As a first step, we construct a benchmark using stochastic versions of two standard optimization problems, knapsack and facility location, with large-scale data. We plan to add more problems in the future.

Knapsack (KS): The problem is to place a set of items with various sizes and values into a knapsack of specified capacity so as to maximize the total knapsack value without exceeding capacity. For a given problem instance the item sizes are stochastic. In the first stage, the items have to be selected without knowing their precise sizes. After the sizes are revealed, the second stage decision is to remove items selected in the first stage to satisfy the capacity limit, but removal incurs a penalty. Context specifies the values of the items, knapsack capacity, and removal penalty.

Facility Location (FL): Adapted from Ntaimo and Sen [2005], the problem is to place facilities at a set of locations with varying costs such that profit is maximized. Customer demand and revenue at the locations are stochastic. There is a linear penalty for not meeting customer demand. In the first stage, locations have to be selected without knowing precise demand and revenue. After demand and revenue are revealed, the second stage

decision is to assign customers to various facilities. Context specifies the location costs, facility capacity, and the linear penalty factor on unsatisfied demand.

Problem size: We define "problem size" to be the number of first stage binary decision varibles. We use the following sizes: $\{25, 50, 100, 200\}$. The number of second stage binary decision variables for KS is the same as in the first stage, while for FL, it is the problem size multiplied by the number of customers (set to 5 in our experiments). In the SAA method the deterministic reformulation multiplies the number of second stage variables by the number of samples. So for problem size 200, even with only 1000 samples, the number of variables already exceeds 2×10^5.

Data generation: We generate samples for the context vectors and for the random variables within an instance using a Mixture of Factor Analyzers (Ghahramani and Hinton [1997]). The resulting distributions are structured with dependencies among variables, and the mixture components make them more complex than a Gaussian.

6.2 BASELINE SOLVERS

SCIP is an open source MIP solver that uses LP relaxation-based branch and bound algorithm. We apply SCIP using the SAA method.

Tabu Search (Glover [1986]) is a local search algorithm that makes moves in the space of the first stage decision variable and maintains a tabu list of recently visited solutions to avoid cycles. See supplementary material for more details.

Progressive Hedging ([Watson and Woodruff, 2011]) is based on the dual decomposition approach described in section 5.1. It iteratively fixes the relaxation introduced in the dual decomposition by adding penalty terms to promote consistency between x_ω and x. See supplementary material for more details.

SCIP, Tabu Search, and Progressive Hedging do not support any transfer across problem instances as they are originally formulated. We implement a simple way to adapt Tabu Search and Progressive Hedging to the contextual setting. At training time, these optimizers solve as many training instances as possible within the given training budget. At test time, a test instance's context vector is used for nearest neighbor selection of a training instance. The solution for the test instance is initialized to the final solution of the nearest training instance by L1 distance.

6.3 EVALUATION PROTOCOL

Solvers that support contextual optimization are given the same training budget of 500 CPU cores for 12 hours. GPUs are not used to allow fair comparison to methods that cannot use them. The training budget is used for hyperparameter tuning and parameter learning. For Tabu Search and Progressive Hedging, instead of parameter learning, a set of training instances are solved for nearest neighbor initialization at test time.

We use 1000 contexts and 10^5 samples for the random variables in a problem for training, and 100 contexts and 1000 random variable samples each for validation and testing. The context distribution is independent of the distribution over random variables in a problem, so we can sample each separately. The number of variable samples is limited to 1000 only because SCIP becomes too slow and uses excessive memory for some problem instances with more samples. RL approaches can easily scale to much larger number of samples. The solver performance on the validation set is used to select the hyperparameters.

At test time we provide one CPU core per instance for all solvers. Tabu Search and Progressive Hedging both make the same number of second stage solver calls (which dominates the running time) as the local search solver so that all three have (approximately) the same computational budget. SCIP's computation cannot be easily characterized by the number of second stage solver calls. Instead we set a maximum time limit and a 5% optimality gap, and allow SCIP to run until whichever condition is satisfied first.

7 EXPERIMENTS

We present the following results: a) a comparison of the local search solver to SCIP in terms of test solution objective value vs. solver running time, b) a comparison to the objective value achieved by the baselines, and c) results for estimating a bound using the dual policy.

7.1 OBJECTIVE VALUE VS. RUNNING TIME

We run SCIP on 100 test problem instances with different running time limits, from 1s to 10^4s. We then compare its objective value to that achieved by the local search solver on the same instances. The running time of the local search solver is fixed to be the time needed to execute one episode per instance and evaluate the solution at the end of the episode.

Figure 2 summarizes the results. Each plot shows the percent difference in SCIP's objective value with respect to that of the local search solver as a function of the running time ratio of SCIP to the local search solver. The key observation is that SCIP gives significantly worse objective value than the local search solver unless it is given much more running time. For knapsack problems

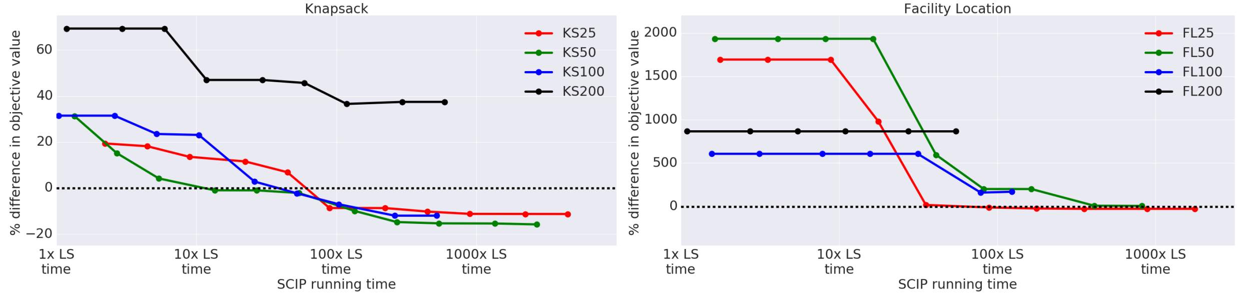


Figure 2: SCIP running time vs. objective value comparison between local search solver and SCIP on test problem instances for the benchmark problems (KS = Knapsack, FL = Facility Location) of sizes $\{25, 50, 100, 200\}$. x-axis is SCIP's average running time as a factor of the local search (LS) solver's average running time. y-axis is the percent improvement in average objective value given by local search over SCIP. Positive values mean local search is better.

of different sizes, SCIP performs 30 to 70% worse when given approximately the same running time. For problem sizes 25, 50, and 100, SCIP needs more than an order of magnitude more time to catch up with the local search solver's solution quality. For problem size 200, running SCIP even 1000 times longer still results in more than 35% worse objective value. A similar result is seen in the facility location problem as well, where the objective value gaps are even bigger in favor of the local search solver. For problem sizes 100 and 200, it appears that SCIP needs more than 10^4 seconds to improve the objective value significantly, and therefore is not able to catch up with the local search solver's objective value in that time.

These results show the speedup benefit of learning a contextual solver. Since the solver has already seen at training time problem instances from the same distribution, generalization allows it to quickly find good solutions on unseen problem instances at test time.

7.2 OBJECTIVE VALUE COMPARISON

We compare the objective value given by the local search solver to that of Tabu Search, Progressive Hedging, and SCIP. We set the running time of SCIP to be ten times that of the local search solver to make it a stronger baseline. We also evaluate the initialization policy and the local move policy independently to get more insight into the performance of the local search solver. For the former we train only the initialization policy to generate a solution directly, without local search. For the latter we train only a local move policy that is initialized by selecting a solution uniformly randomly.

Table 1 summarizes the results. For each (solver, problem, size) triplet the mean of the objective values over a set of 100 test instances is shown. In all but one case the learned local search solver computes the best solution among all the algorithms tried, and in the remaining case it is within 5% of the best.

Both Tabu Search and Progressive Hedging perform poorly on the larger problem sizes (100, 200), especially for Facility Location. As we have already seen in the previous section, SCIP requires significantly more time to achieve comparable objective values.

A comparison among the different variants of the learned solver shows that the best results are given by learning both the initialization and local move policies jointly. Learning a local move policy to start from a random solution performs the worst among the variants, especially for larger problem sizes (KS200, FL100 and FL200). Learning an initialization policy alone performs better, but uniformly worse than learning both jointly.

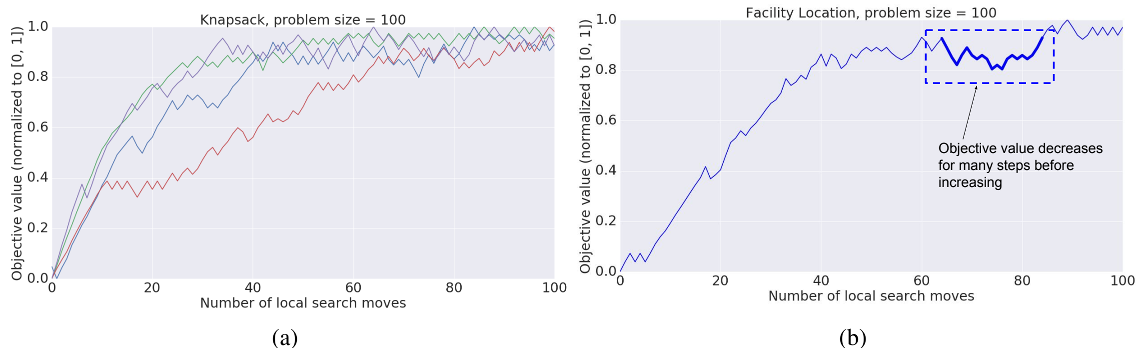
The dual policy bounds computed by the approach from section 5.2 also show that on most problems, our learning based approach not only learns to produce good solutions but is also able to prove that these solutions are within 20% of the optimum (5 of 8 problem instances).

7.3 INSIGHTS INTO SOLVER BEHAVIOR

Figure 3(a) shows examples of how the objective value varies as a function of solver steps (here for KS100, other problems show similar behavior). The objective value typically improves rapidly for roughly the first one-third of an episode, with smaller gains afterwards. Unlike a greedy algorithm, the solver is not constrained to always improve the objective value. Figure 3(b) shows an example. The highlighted part of an episode shows that the objective value can decrease by a large amount ($> 10\%$) for a significant duration ($> 10\%$) of the episode before it improves again. Approximately 7% of test instances across problem types and sizes show non-greedy behavior. The ability to learn a non-greedy policy explains how the local search solver can outperform greedy methods like Tabu Search.

Solver	KS 25	KS 50	KS 100	KS 200	FL 25	FL 50	FL 100	FL 200
SCIP @10x more runtime	0.5985	0.1681	2.731	1.1273	0.0	0.0	-7.0531	-13.8540
Tabu Search	1.3150	**1.4859**	-3.0144	-0.0851	0.9781	0.3089	0.0	0.0
Progressive Hedging	1.4666	1.2584	2.9565	1.1983	0.9301	0.0003	0.0014	0.0
RL-based solvers								
Init. policy only	1.2410	1.2977	0.1304	0.9275	0.8213	0.8148	1.1395	1.3337
Local move policy only	1.1427	1.0631	1.2481	0.1846	1.1058	0.7403	-7.3248	-36.8738
Init. policy + local move policy	**1.4807**	1.4286	**3.2920**	**1.3133**	**1.1352**	**1.2416**	**1.2729**	**1.8197**
Bounds from dual policy								
Dual policy	1.5885	2.4486	3.4289	3.8304	1.2753	1.2903	1.6489	2.4702

Table 1: Comparison of average test set objective values achieved by various solvers across benchmark problems and sizes. KS = Knapsack, FL = Facility Location. The last row shows the upper bounds computed by the dual policy.



(a) (b)

Figure 3: Examples of local search behavior on test instances. (a) Objective value as a function of the number of local search moves for 100-d Knapsack for several test instances. (Other problems show similar behavior.) Objective values are normalized to [0,1] for display purposes. (b) Example of non-greedy behavior (highlighted) shown by the local solver. Approximately 7% of test instances show this type of behavior.

Non-greediness can also potentially make the solution worse than the initial solution. However, only 0.21% of test instances show a decrease $> 1\%$ in objective value relative to the initial solution, while 89.8% show an increase $> 10\%$. It appears that on some instances the initial solution is already very good, and local search introduces small changes which reduces the objective value slightly.

8 CONCLUSIONS

We presented a learned contextual local search solver that jointly learns both an initialization policy and a local move policy. On benchmark problems of two-stage stochastic knapsack and facility location of different sizes, it achieves approximately 30% to 2000% better objective value for the same running time and at least an order of magnitude faster for the same objective value. It also outperforms baseline algorithms such as Tabu Search and Progressive Hedging on seven out of eight problems. Joint learning is key as learning only one of the policies in isolation results in worse performance. The dual policy is able to compute bounds showing that the local search solver's objective value is within 20% of the optimum for 5 out of 8 problem instances.

Next directions include incorporating a more powerful search procedure such as MCTS into the agent as a policy improvement operator (Silver et al. [2017]).

References

S. Ahmed and A. Shapiro. The sample average approximation method for stochastic programs with integer recourse. *SIAM Journal of Optimization*, 12:479–502,

2002.

I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer-Verlag, 1997.

J. Boyan and A. W. Moore. Learning evaluation functions to improve optimization by local search. *JMLR*, 1: 77–112, Sept. 2001.

C. C. Carøe and R. Schultz. Dual decomposition in stochastic integer programming. *Operations Research Letters*, 24(1-2):37–45, 1999.

Z. Ghahramani and G. E. Hinton. The em algorithm for mixtures of factor analyzers. Technical report, University of Toronto, 1997.

A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig. The scip optimization suite 5.0. Technical report, 2017.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5): 533–549, 1986.

E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6348–6358. 2017a.

E. B. Khalil, P. L. Bodic, L. Song, G. Nemhauser, and B. Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 724–731, 2016.

E. B. Khalil, B. Dilkina, G. L. Nemhauser, S. Ahmed, and Y. Shao. Learning to run heuristics in tree search. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 659–666, 2017b.

G. Laporte, F. Louveaux, and H. Mercure. The vehicle routing problem with stochastic travel times. *Transportation Science*, 26(3):161–170, 1992.

H. Larochelle and I. Murray. The neural autoregressive distribution estimator. In *The Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, volume 15 of *JMLR: W&CP*, pages 29–37, 2011.

A. Lodi and G. Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1928–1937, 2016.

R. Moll, A. G. Barto, T. J. Perkins, and R. S. Sutton. Learning instance-independent value functions to enhance local search. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 1017–1023. MIT Press, 1999.

L. Ntaimo and S. Sen. The million-variable march for stochastic combinatorial optimization. 32:385–400, 07 2005.

A. Shapiro, D. Dentcheva, and A. Ruszczyski. *Lectures on Stochastic Programming*. Society for Industrial and Applied Mathematics, 2009.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550, 2017.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. 2015.

J.-P. Watson and D. L. Woodruff. Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science*, 8(4):355–370, Nov 2011.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, 1992.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1114–1120, 1995.

Q. P. Zheng, J. Wang, and A. L. Liu. Stochastic optimization for unit commitment - a review. *IEEE Transactions on Power Systems*, 30(4):1913–1924, 2015.