Integer optimization and machine learning:
Some recent developments

Karen Aardal
TU Delft

PGMO Days, Paris, December 3, 2019
Part I: Using Machine Learning to enhance elements of Integer Optimization algorithms

Part II: Some relations between Integer (Linear) Optimization and Deep Learning
Background: Integer Optimization

**IP:** \( z_{IP} = \min c^T x \)  
\[ \text{s.t. } A x \geq b \]  
\[ x \in \mathbb{Z}^n \]

**LP-relaxation:**  
\( z_{LP} = \min c^T x \)  
\[ \text{s.t. } A x \geq b \]  
\[ x \geq 0 \]

---

**Branch-and-bound: (B&B)**

\( \bar{z} \): Value of best feasible solution found so far

**Solve LP-relaxation**
If we cannot prune, choose a variable \( x_j = \hat{x}_j \notin \mathbb{Z} \):

\[ x_j \leq \lfloor \hat{x}_j \rfloor \]
\[ x_j \geq \lfloor \hat{x}_j \rfloor + 1 \]
Variable Selection Problem: Choose the non-integer variable to “branch” on in a given B&B-node

LP\((k)\) = LP-relaxation in node \(k\) = LP-relaxation in \(k\) obtained by adding all constraints on the path from the root node to \(k\).

We can prune the tree under node \(k\) if one of the following happens:

- If LP\((k)\) has integral solution. IP\((k)\) has been solved to optimality.
- If LP\((k)\) is infeasible. IP\((k)\) is infeasible.
- If \(z_{LP}(k) \geq \bar{z}\). Prune by bound.

\[ x_j \leq \lfloor \hat{x}_j \rfloor \quad x_j \geq \lceil \hat{x}_j \rceil + 1 \]
We can prune the tree under node $k$ if one of the following happens:

- If $\text{LP}(k)$ has integral solution.
- If $\text{LP}(k)$ is infeasible.
- If $z_{LP}(k) \geq \bar{z}$.

If we cannot prune under node $k$, we need to branch on a non-integer variable, i.e., solve the Branching Variable Selection Problem (VSP).

Depending on how “well” we solve the VSP, the size of the resulting search tree can differ a lot!

In addition to VSP, we also need to solve the Branching Node Selection Problem (NSP), i.e., which of the non-pruned nodes should we investigate next?
How is the VSP solved in academic/commercial solvers?

Four important rules:

• Pseudocost branching: keeps a history of objective gain per unit change in variable value

• Strong branching: computes progress in objective value for each fractional variable, and chooses the best

• Reliability branching: start with strong branching until pseudocost branching becomes “reliable”

• Hybrid branching: combination of several rules, also from the CP/SAT community

**Strong branching** seems to result in the smallest search trees.

Drawback: **Computationally heavy!**
Node Selection Problem:
Choose which of the non-pruned B&B-nodes to explore next.

Two basic strategies:

• Best-first: Choose the node $k$ that has the smallest value $z_{LP}(k)$.

• Depth-first: go deeper and deeper and backtrack only when a node is pruned.
Cutting planes:

**IP:** \[ z_{IP} = \min \mathbf{c}^T \mathbf{x} \]  
\[ \text{s.t. } \mathbf{A} \mathbf{x} \geq \mathbf{b} \]  
\[ \mathbf{x} \in \mathbb{Z}^n \]

**LP-relaxation:** \[ z_{LP} = \min \mathbf{c}^T \mathbf{x} \]  
\[ \text{s.t. } \mathbf{A} \mathbf{x} \geq \mathbf{b} \]  
\[ \mathbf{x} \geq 0 \]
How can we improve the “natural” LP-relaxation?

Add a new constraint that cuts off the current LP-optimum, but none of the integer feasible points.

Cutting planes that do not assume any specific problem structure:

• Gomory mixed-integer cuts
• Split cuts
• Mixed-integer rounding cuts
Theorem: (Gomory 1958) After adding finitely many Gomory cuts, the integer optimum is achieved (under some technical conditions).

All modern commercial/academic B&B solvers for (mixed)-integer optimization problem include:

• Presolve to reduce problem size and improve bounds.
• Heuristics for finding good feasible solutions. (Improves $\bar{z}$)
• Cutting plane generating algorithms. (Improves $z_{LP}(k)$)
• Advanced algorithms for the variable and node selection problems.

QUESTION: Can we use machine learning to “learn” any of these components?
Not easy to improve on what the best solvers already do!

In the first part I will focus on the variable selection problem.

For the node selection problem, see e.g.:


For learning to cut, see e.g.:


Background: Machine Learning

Accessible online, gives a broad introduction to machine learning
Background: Machine Learning

Just published...
Background: Machine Learning

Overview paper:


Papers that introduce how machine learning is used in optimization


**Some learning settings:**

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised learning</td>
<td>Data consists of pairs consisting of a set of features and the correct/optimal outcome that is then used for training.</td>
</tr>
<tr>
<td>Unsupervised learning</td>
<td>Data consist of features, and the process of learning should detect a “pattern” in the features.</td>
</tr>
<tr>
<td>Reinforcement learning</td>
<td>An “agent” interacts with the environment through a MDP. The agent is given a state of the environment and chooses an action that gives a certain reward. The training is done so as to maximize sum of future rewards.</td>
</tr>
<tr>
<td>Deep learning</td>
<td>Input is passed successively through a number of layers in a directed acyclic network. In each vertex of each layer an affine transformation followed by a non-linear operation is applied. The parameters of these transformations/operations are learned by minimizing a “loss” function.</td>
</tr>
</tbody>
</table>
Learning to branch (VSP)

Important: which features should be used in the ML-algorithm?

• There should be strong enough statistical dependencies between the chosen features and the desired output.

• The features should be fast to compute.

• The number of features used should be independent of instance size.

• Features should be invariant to irrelevant changes to instance.

• Features should be invariant under scaling of instance input
We describe results from two papers regarding the VSP:


For more results we refer to the survey papers:


Supervised learning: Train to approximate strong branching.

Learning algorithm: Extremely Randomized Trees (Geurts et al, 2006).

Features (describe variable $j$ in the current node):

Static features:

- objective related: $\text{sign of } c_j$

$|c_j| / \sum_{i: c_i \geq 0} c_i, \quad |c_j| / \sum_{i: c_i < 0} |c_i|$

- constraint related: things that represent the influence of the constraint coefficients of variable $j$

examples: $a_{ij} / |b_i|, \quad |c_j| / a_{ij}$

only the max and the min value over all constraints of each feature is added to the features vector.

Features:

Dynamic features:

Problem related:

• proportion of fixed variables at current solution
• up and down fractionalities of variable \( j \)
• normalized “Driebeck penalties” up and down for variable \( j \) (bound on the increase in objective value)
• normalized sensitivity range of \( c_j \)

Optimization related:

• relative objective increase up and down of variable \( j \)
• number of times variable \( j \) has been branched on normalized by total number of branchings

Computational result:
Tested on 0-1 IPs: Randomly generated and selection of MIPLIB instances.

Instance size: couple of hundreds of variables, about 100 constraints.

Use CPLEX library, turn off cuts, heuristics, presolve, parallelization.

Conclusions:

Random problems: (none were solved within the given limits)

With #B&B-nodes limit:

The closed gap is comparable to Reliability Branching (RB), and slightly worse than Strong Branching (SB).
closed gap $\in [0, 1] = \frac{z_{current} - z_{initial}}{z_{OPT} - z_{LP}}$

If closed gap $\approx 1$, we are closed to verifying optimal solution.

Random problems: (none were solved within the given limits)

- **With #B&B-nodes limit:**

  The closed gap is comparable to Reliability Branching (RB), and slightly worse than Strong Branching (SB).

  Computing time factor 2 worse than RB, but factor 4 better than SB.

- **With time limit:**

  The closed gap is inbetween (SB) and (RB). Using a factor 1-2 less nodes than RB, but a factor 3 more nodes than SB.

MIPLIB problems: (solved within the node/time limit)

With #B&B-nodes limit:

The #B&B-nodes used comparable to Reliability Branching (RB), and a factor of 2 worse than Strong Branching (SB).

Computing time factor 7-8 better than RB and SB.

With time limit:

Time used is comparable to RB and a factor 2 better than SB.

Uses twice as many nodes as RB and SB.

MIPLIB problems: (not solved by at least one method within the node/time limit)

With #B&B-nodes limit:

The closed gap is comparable to Reliability Branching (RB), and slightly worse than Strong Branching (SB).

Number of nodes used comparable to RB and SB.

Computing time factor 6 better than SB and comparable to RB.

With time limit:

Closed gap slightly worse than RB and SB.

Uses factor 2 fewer nodes than RB and a factor 3 more nodes than SB.
Imitation learning: Solve training problems with strong branching.

“Ideally”: model the B&B process as a MDP.

The “state” of the process comprises all relevant info about the current B&B tree.

The “action” to be taken is to choose a variable to branch on.

This is computationally too heavy. Therefore, the “state” is encoded as a bipartite graph:

Edge between variable \( j \) and constraint \( i \) if \( a_{ij} \neq 0 \)
Features are associated to both the constraint and variable nodes.

Variable selection policy is parametrized using a **Graph Convolutional Neural Network**

In state $t$

- Input layer
- Bipartite graph with features
- Hidden layers
- Output layer
- Branching policy

Computational result:

They test on three classes of combinatorial optimization problems:

• **Set cover**: 1000 variables, train on 500 constraints, test on 500, 1000, and 2000 rows

• **Combinatorial auction**: train on 100 items, 500 bids (100, 500), test on (100, 500), (200, 1000), (300, 1500)

• **Capacitated facility location**: 100 facilities, train on 100 customers, test on 100, 200, and 400 customers

Use SCIP with time limit 1 hour. Allow for cutting planes in root node. All other SCIP settings are default.
Comparison is made with:

- Three other ML-algorithms
- Default SCIP branching
- Strong branching

Some conclusions:

- Strong branching always gives far fewer nodes, but at substantially higher computational cost.

**Set cover:**

- Small instances: LMART (Hansknecht et al. 2018) is faster, and SCIP uses fewer nodes.
- Medium and large instances: GCNN is faster and uses fewer nodes.
**Combinatorial auction:**

- Small instances: LMART (Hansknecht et al. 2018) is faster, and SCIP uses fewer nodes.
- Medium instances: GCNN is faster, and SCIP uses fewer nodes.
- Large instances: GCNN is faster and uses fewer nodes.

**Capacitated facility location:**

- Small and medium instances: GCNN is faster, and SCIP uses fewer nodes.
- Large instances: SVMRANK (Khalil et al. (2016) is faster, and SCIP uses fewer nodes.
Part II: Some relations between Integer (Linear) Optimization and Deep Learning
A bit more on Deep Learning

Deep Neural network (DNN):

Directed acyclic network with weights on every edge and vertex

input layer  hidden layer 1  hidden layer 2  hidden layer k  output layer
Number of nodes in the input and output layer is related to the problem at hand.

Example: we wish to recognize hand written numbers.
# nodes input layer = # of pixels in the image

# nodes output layer = # of possible numbers in the data set, here 10 (0-9)
What goes on in the hidden layers?

\[ x_j = \varphi(\sum_i w_{ij} x_i + b_j) \]

\( \varphi() : \mathbb{R} \rightarrow \mathbb{R} \) is called an “activation” function

Examples:  
ReLU (Rectified Linear Unit)  
\( \varphi(x) = \max\{0, x\} \)

Sigmoid  
\( \varphi(x) = \frac{e^x}{1 + e^x} \)
Examples:

ReLU (Rectified Linear Unit)  \( \varphi(x) = \max\{0, x\} \)

Sigmoid

\[ \varphi(x) = \frac{e^x}{1 + e^x} \]

ReLU:

Sigmoid:
Some theoretical results for DNNs

**Expressiveness:** What family of functions can one represent using ReLU-DNNs?

**Theorem:** (Arora, Basu, Mianjy, Mukherjee 2018)
Any ReLU-DNN with $n$ inputs implements a continuous piecewise affine function on $\mathbb{R}^n$. Conversely, any continuous piecewise affine function on $\mathbb{R}^n$ can be implemented by some ReLU-DNN. Moreover, at most $\log(n + 1)$ hidden layers are needed.
Size, depth, and width of a DNN

Size of the DNN = $w_1 + w_2 + \cdots + w_k$

Depth of the DNN = $k + 1$

Width of the DNN = $\max\{w_1, w_2, \ldots, w_k\}$
Efficiency: How many layers and vertices do we need to represent functions in the family of continuous piecewise linear functions?

Theorem: (Arora et al, 2018) For every natural number \( N \), there exists a family of \( \mathbb{R} \rightarrow \mathbb{R} \) functions such that for any function \( f \) in the family, we have:

1. \( f \) is in ReLU-DNN(\( N^2 \), \( N^3 \))
2. \( f \) is not in ReLU-DNN(\( N \), \( (1/2)N^n - 1 \))

So, there are “hard” functions which, if represented in a shallower DNN, require a DNN of exponentially larger size.
**Training:**

Given the architecture and data points \((x, y)\), find weights for the best fit function.

**Theorem:** (Arora et al, 2018) Let \(n\) and \(w\) be natural numbers, and 
\((x^{1}, y^{1}), \ldots, (x^{D}, y^{D})\) a set of \(D\) data points in \(\mathbb{R}^n \times \mathbb{R}\). There exists an algorithm that solves the following training problem to optimality:

\[
\min \{ |F(x^{1}) - y^{1}| + \cdots + |F(x^{D}) - y^{D}| : F \in \text{ReLU-DNN}(1, w) \}.
\]

The running time of the algorithm is \(2^w D^{nw} \text{poly}(D, n, w)\).

Polynomial in data size \(D\) for fixed \(n\) and \(w\).

They convert the training problem, for an arbitrary number of layers, into a linear programming (LP) problem with size(LP) linear in $D$ and exponential in input and parameter space dimensions.

Make use of Bienstock & Muñoz reformulation of non-convex problems to approximate LPs.
Adversarial machine learning and MIP


Model a ReLU-DNN as a MIP:
What goes on in the hidden layers?

\[ x_j = \varphi(\sum_i w_{ij} x_i + b_j) \]

\( \varphi() : \mathbb{R} \rightarrow \mathbb{R} \) is called an “activation” function

ReLU (Rectified Linear Unit) \( \varphi(x) = \max\{0, x\} \)
Adversarial machine learning and MIP


Model a ReLU-DNN as a MIP:

Recall: ReLU (Rectified Linear Unit)\[ x_j = \varphi(\sum_i w_{ij}x_i + b_j) \]
\[ \varphi(x) = \max\{0, x\} \]

\[
\begin{align*}
\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1}x_i^{k-1} + b_j^{k-1} &= x_j^k - z_j^k, \\
x_j^k, z_j^k &\geq 0
\end{align*}
\]

\[
\begin{align*}
k &= 1, \ldots, K \\
j &= 1, \ldots, n_k
\end{align*}
\]

How can we make sure that \(x_j^k\) and \(z_j^k\) do not both take positive values?

Standard MIP modeling technique!
Adversarial machine learning and MIP


\[
\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_{i}^{k-1} + b_{j}^{k-1} = x_{j}^{k} - z_{j}^{k}, \quad k = 1, \ldots, K \\
x_{j}^{k}, z_{j}^{k} \geq 0 \\
\]

How can we make sure that $x_{j}^{k}$ and $z_{j}^{k}$ do not both take positive values?

Standard MIP modeling technique!

Introduce binary variables $y_{j}^{k}$:

- $y_{j}^{k} = 1$ should imply $x_{j}^{k} \leq 0$
- $y_{j}^{k} = 0$ should imply $z_{j}^{k} \leq 0$

\[
\begin{align*}
x_{j}^{k} & \leq M(1 - y_{j}^{k}) \\
z_{j}^{k} & \leq M y_{j}^{k} \\
y_{j}^{k} & \in \{0, 1\}
\end{align*}
\]
MIP model:

\[
\begin{aligned}
\min & \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=0}^{K} \sum_{j=1}^{n_k} \gamma_j^k y_j^k \\
\text{s.t.} & \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - z_j^k, \\
& x_j^k, z_j^k \geq 0 \\
& x_j^k \leq M (1 - y_j^k) \\
& z_j^k \leq M y_j^k \\
& y_j^k \in \{0, 1\}
\end{aligned}
\]

The input \( w_{ij}^{k-1}, c_j^k, \gamma_j^k, b_j^k \) is obtained from training.
Fischetti and Jo use their MIP model to find “adversarial” instances:
How little can we change the input such that the DNN makes a mistake?

Example: The MNIST data set again.
We are given an input vector $\tilde{x}^0$ correctly classified as a “0”. **Example:** We want to produce a similar vector $x^0$ that is wrongly classified as a “5”.

Add constraints for the final layer:

$$x^K_{5+1} \geq 1.2 x^K_{j+1}, \quad j \in \{0, 1, 2, 3, 4, 6, 7, 8, 9\}$$

This means that the activation of the output element corresponding to the “5” should be 20% higher than for any other digit.

Introduce an ad-hoc objective function $\sum_{j=1}^{n_0} d_j$ to minimize the $L_1$-norm distance between $x^0$ and $\tilde{x}^0$.

The new additional variables $d_j$ should satisfy the constraints

$$-d_j \leq x^0_j - \tilde{x}^0_j \leq d_j \quad \text{for } j = 1, \ldots, n_0.$$ 

For the MNIST examples the change in input vector is only attributed to a very few (2-3) pixels!
Other papers discussing MILP models of DNNs:


Open questions

• Can we learn how to branch on hyperplanes rather than on single variables?

• How to use learning in MIP in a parallel environment?

• Can we use learning to classify problems (instances) in terms of “difficulty”? 