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:
$$\begin{aligned} z_{IP} = \min\ & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.}\ \boldsymbol{A}\boldsymbol{x} & \geq \boldsymbol{b} \\ \boldsymbol{x} & \in \mathbb{Z}^n \end{aligned}$$

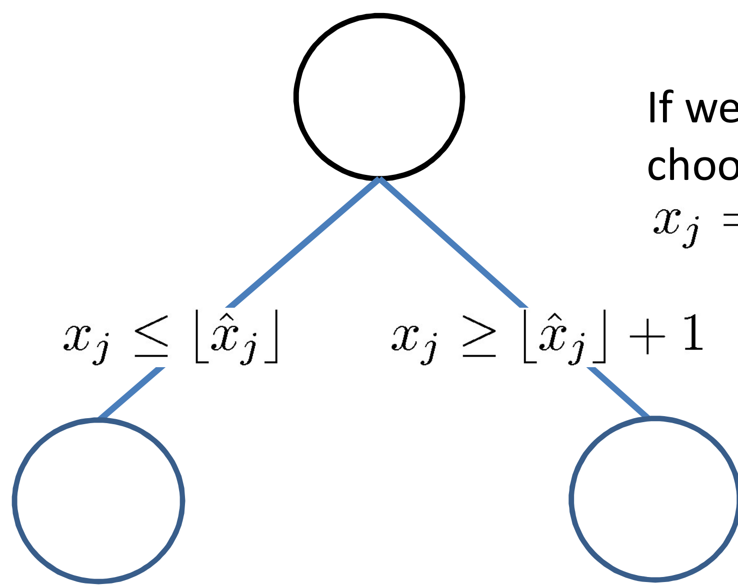
LP-relaxation:
$$\begin{aligned} z_{LP} = \min\ & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.}\ \boldsymbol{A}\boldsymbol{x} & \geq \boldsymbol{b} \\ \boldsymbol{x} & \geq \boldsymbol{0} \end{aligned}$$

Branch-and-bound: (B&B)

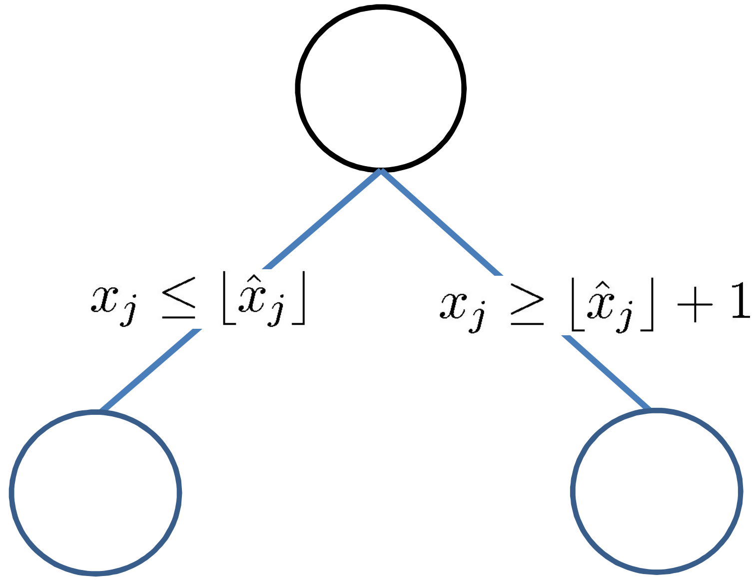
$\bar{z}$: Value of best feasible solution found so far

Solve LP-relaxation

If we cannot prune, choose a variable $x_j = \hat{x}_j \notin \mathbb{Z}$:



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.

We can prune the tree under node *k* if one of the following happens:

- If LP(*k*) has integral solution.
- If 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?

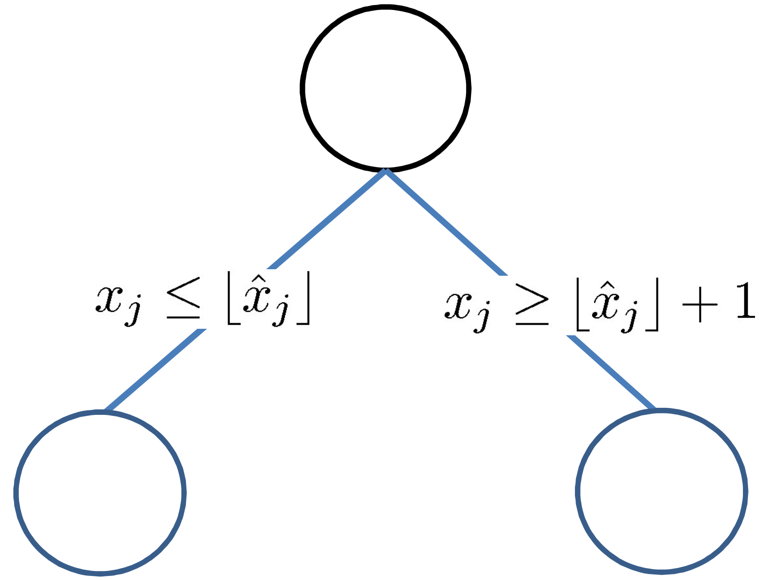
Four important rules:

- 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!

<u>Node Selection Problem</u>: Choose which of the non-pruned B&B-nodes to explore next.



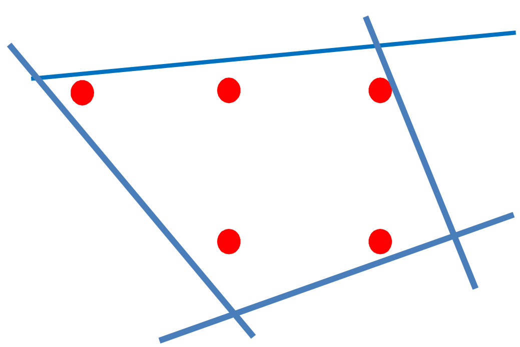
Two basic strategies:

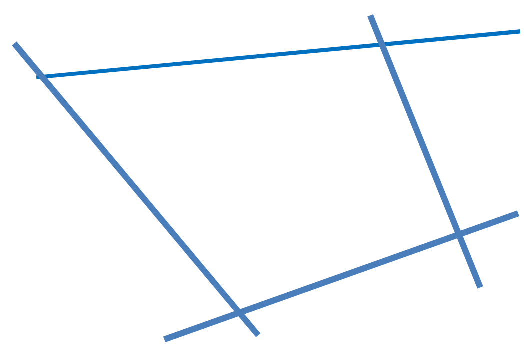
- Best-first: Choose the node *k* that has the smallest value $z_{LP}(k)$.
- Depth-first: go deeper and deeper and backtrack only when a node is pruned.

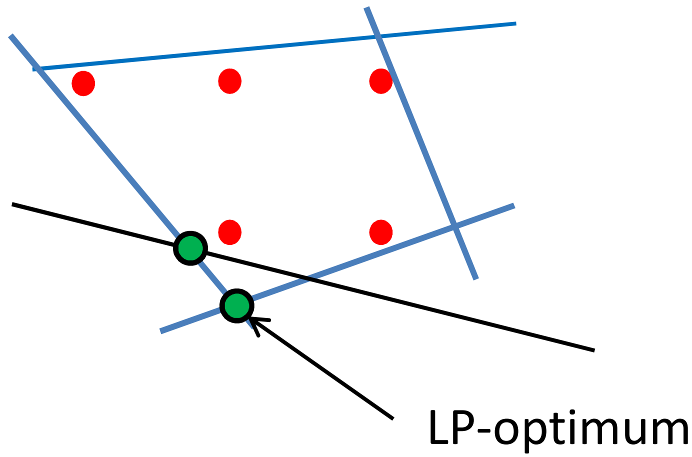
Cutting planes:

IP: $$z_{IP} = \min \boldsymbol{c}^T \boldsymbol{x}$$
$$\begin{aligned} \text{s.t.}\ \ \boldsymbol{A}\boldsymbol{x} &\geq \boldsymbol{b} \\ \boldsymbol{x} &\in \mathbb{Z}^n \end{aligned}$$

LP-relaxation: $$z_{LP} = \min \boldsymbol{c}^T \boldsymbol{x}$$
$$\begin{aligned} \text{s.t.}\ \ \boldsymbol{A}\boldsymbol{x} &\geq \boldsymbol{b} \\ \boldsymbol{x} &\geq \boldsymbol{0} \end{aligned}$$







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
- ⋮

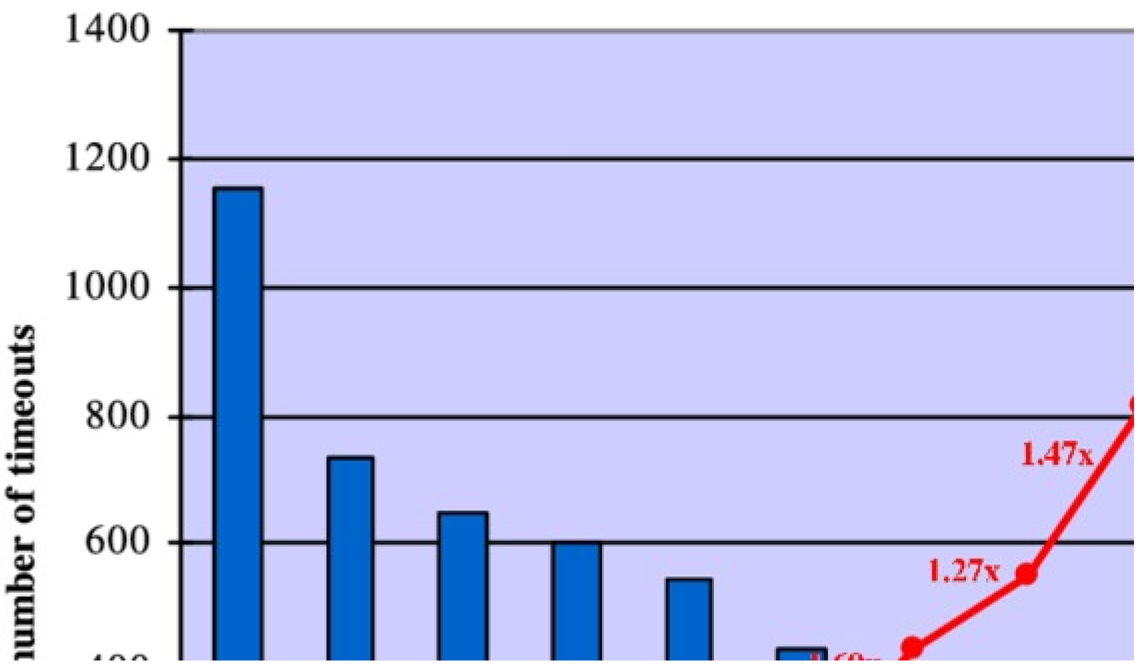
Theorem: (Gomory 1958) After adding finitely many Gomory cuts, the integer optimum is achieved (under some technical conditions).

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!



From: Achterberg & Wunderling: *Mixed Integer Programming: Analyzing 12 Years of Progress*.
In: M. Jünger, G. Reinelt (eds.) Facets of Combinatorial Optimization: Festschrift for Martin Grötschel. Springer, Berlin, pp 449-481, 2013.

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

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

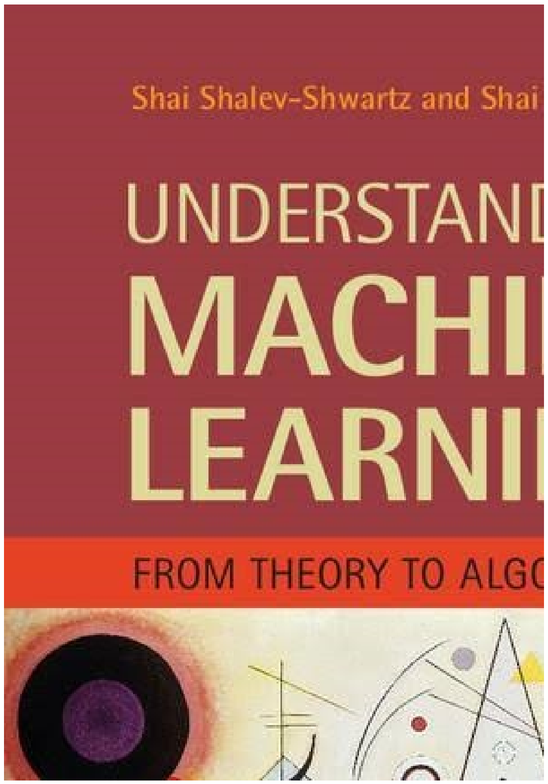
H. He, H. Daumé III, J. Eisner (2014). Learning to search in branch-and-bound algorithms. In: Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, K.Q. Weinberger (eds.) Advances in Neural Information Processing Systems 27, pp 3293-3301.

For learning to cut, see e.g.:

Y. Tang, S. Agrawal, Y. Faenza (2019). Reinforcement learning for integer programming: learning to cut. arXiv:1906.04859v1 [cs.LG] 11 Jun 2019.

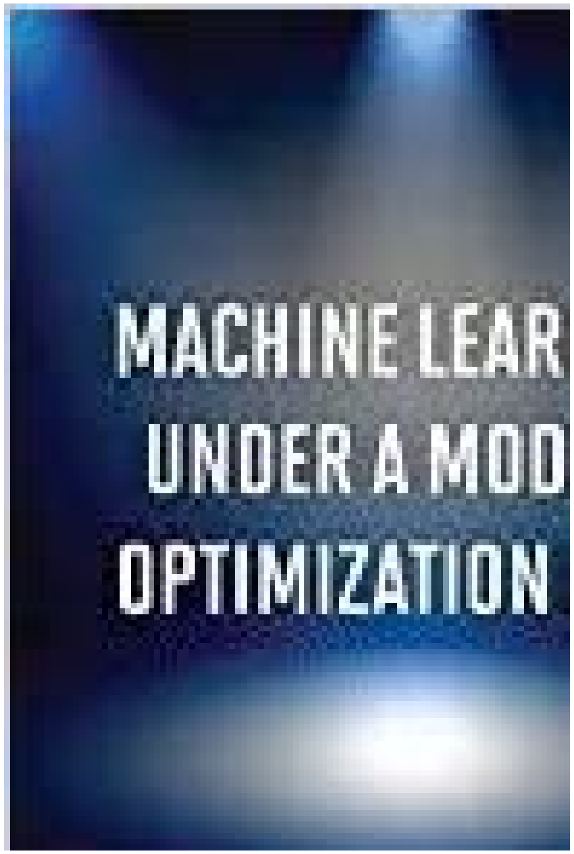
R. Baltean-Lugojan, R. Misener, P. Bonami, A. Tramontani (2018). Strong sparse cut selection via trained neural nets for quadratic semidefinite outer-approximation. Tech report, Imperial College.

Background: Machine Learning



Accessible online, gives a broad introduction to machine learning

Background: Machine Learning



Just published…

Background: Machine Learning

Overview paper:

- P. Domingos (2012). A few useful things to know about machine learning. Commun. ACM 55(10):78-87.

Papers that introduce how machine learning is used in optimization

- A. Lodi and G. Zarapellon (2017). On learning and branching: a survey. TOP 25:207-236.

- Y. Bengio, A. Lodi, A. Prouvost (2018). Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv:1811.06128v1 [cs.LG] 15 Nov 2018. Submitted.

Some learning settings:

• Supervised learning	Data consists of pairs consisting of a set of features and the correct/optimal outcome that is then used for training.
• Unsupervised learning	Data consist of features, and the process of learning should detect a “pattern” in the features.
• Reinforcement learning	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.
• Deep learning	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.

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:

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, A. Lodi (2019). Exact combinatorial optimization with graph convolutional neural networks. arXiv:1906.01692v2 [cs.LG] 7 Jun 2019.

For more results we refer to the survey papers:

- A. Lodi and G. Zarapellon (2017). On learning and branching: a survey. TOP 25:207-236.

- Y. Bengio, A. Lodi, A. Prouvost (2018). Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv:1811.06128v1 [cs.LG] 15 Nov 2018. Submitted.

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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: sign of c_j

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

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.

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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 libary, turn off cuts, heuristics, presolve, parallelization.

Conclusions:

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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 $\in [0,1] = \dfrac{z_{LP}^{\text{current}} - z_{LP}^{\text{initial}}}{z_{OPT} - z_{LP}^{\text{initial}}}$

If closed gap ≈ 1, we are closed to verifying optimal solution.

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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.

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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.

A. Marcos Alvarez, Q. Louveaux, L. Wehenkel (2017). A machine learning-based approximation of strong branching. INFORMS J on Comp. 29(1): 185-195.

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.

M. Gasse, D. Chételat, N. Ferroni, L. Charlin, A. Lodi (2019). Exact combinatorial optimization with graph convolutional neural networks. arXiv:1906.01692v2 [cs.LG] 7 Jun 2019.

Imitation learning: Solve training problems with strong branching.

"Ideally": model the B&B process as a MDP.

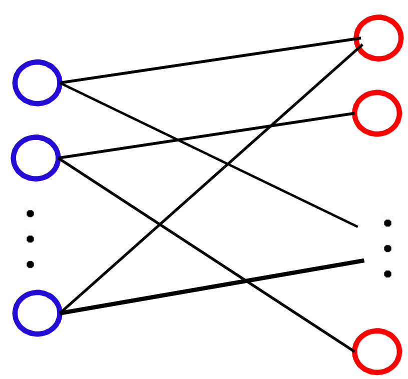
The "state" of the process comprises all relevant info about the current B&B tree.

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 $a_{ij} \neq 0$

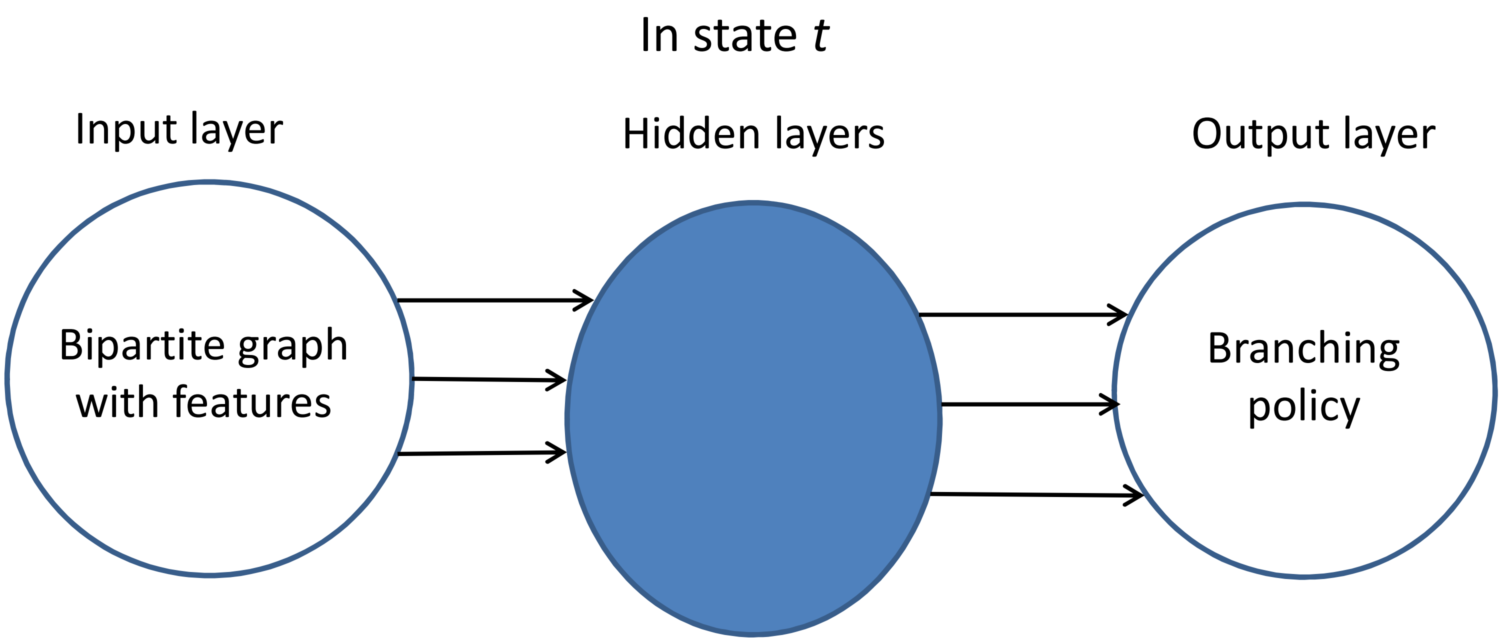
"constraint nodes"



"variable nodes"

Features are associated to both the constraint and variable nodes.

Variable selection policy is parametrized using a
Graph Convolutional Neural Network



M. Gasse, D. Chételat, N. Ferroni, L. Charlin, A. Lodi (2019). Exact combinatorial optimization with graph convolutional neural networks. arXiv:1906.01692v2 [cs.LG] 7 Jun 2019.

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:

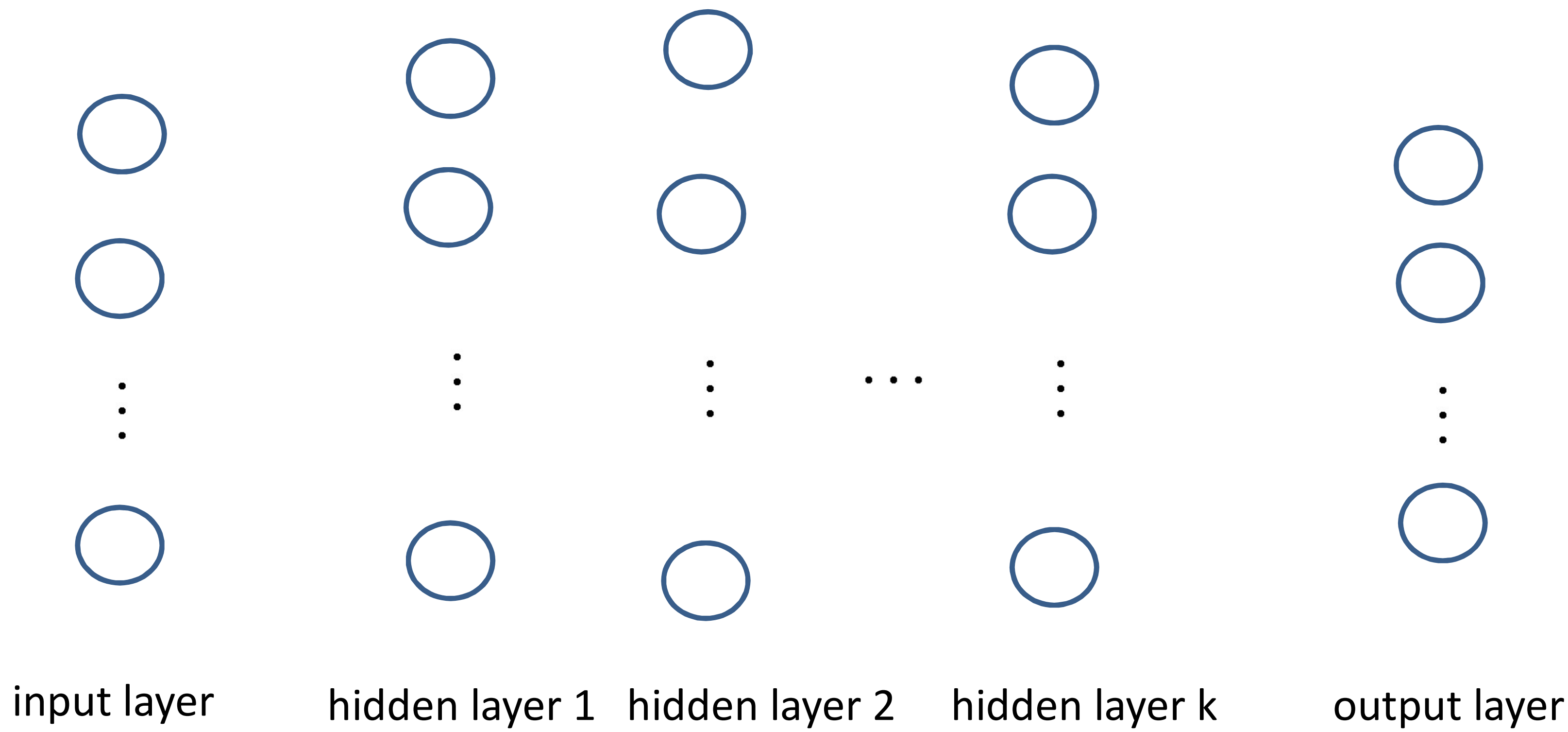
- Small and medium instances: GCNN is faster, and SCIP uses fewer nodes.
- Large instances: SVMRANK (Khalil et al. (2016) is faster, and SCIP uses fewer nodes.

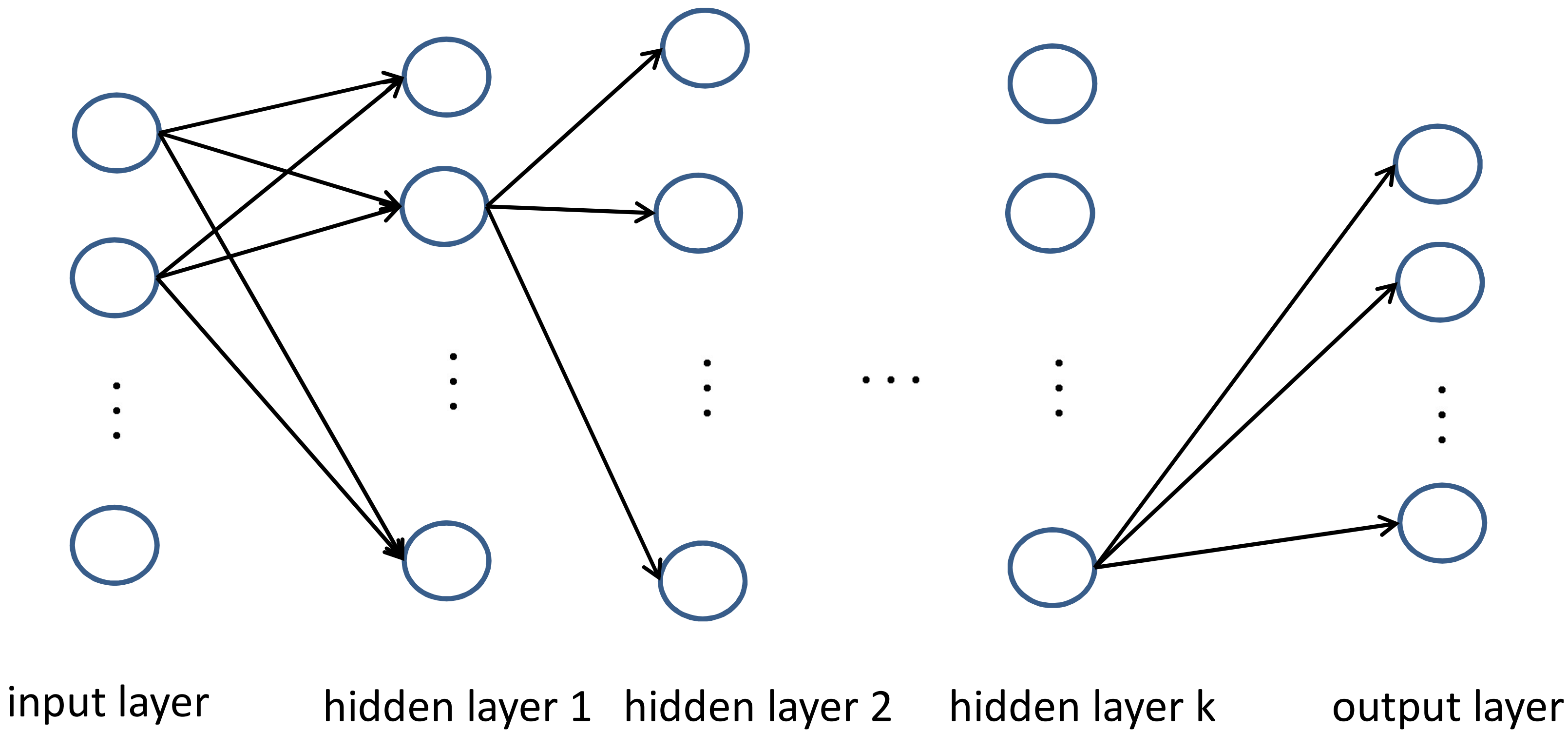
Part II: Some relations between Integer (Linear) Optimization and Deep Learning

A bit more on Deep Learning

Deep Neural network (DNN):

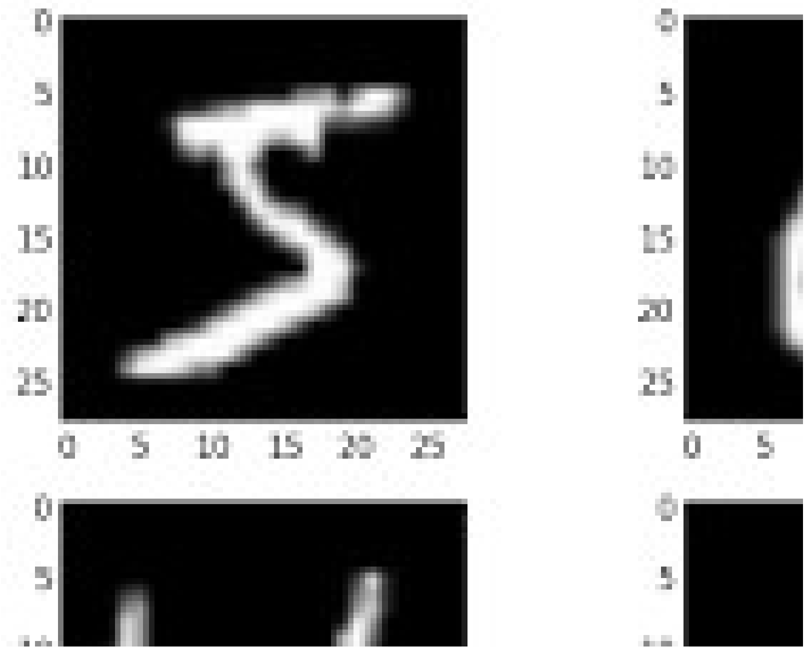
Directed acyclic network with weigths on every edge and vertex





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

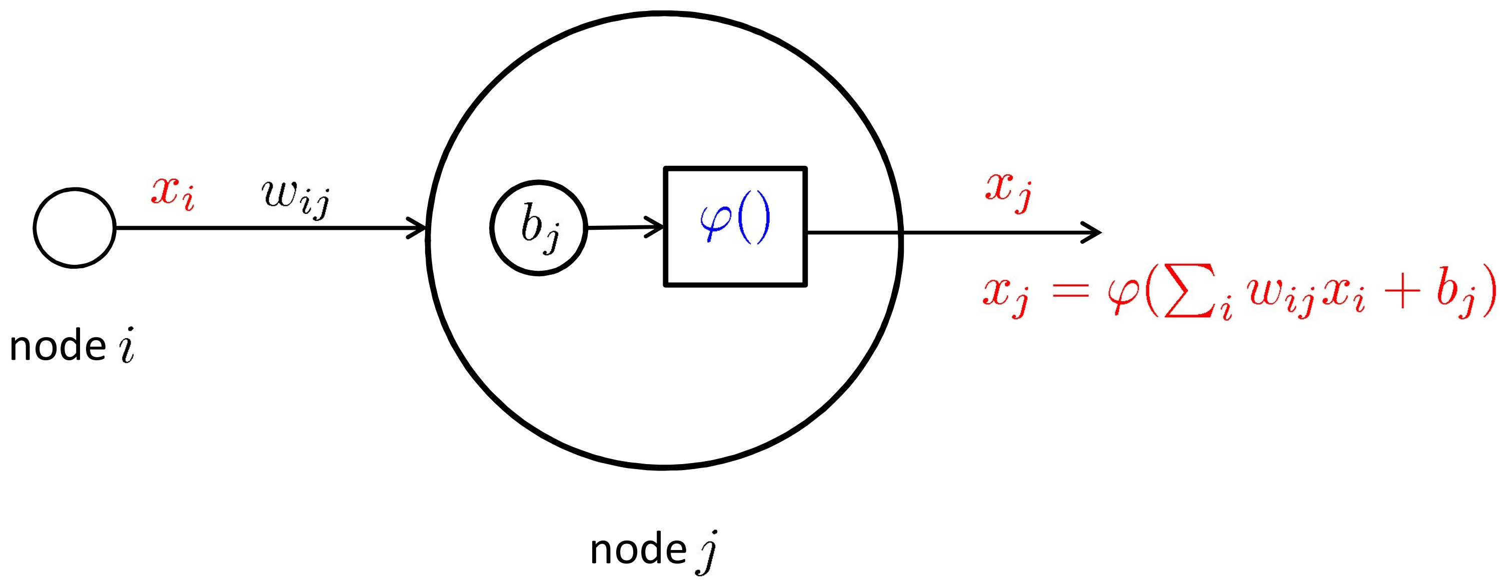
<u>Example</u>: we wish to recognize hand written numbers.



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?



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

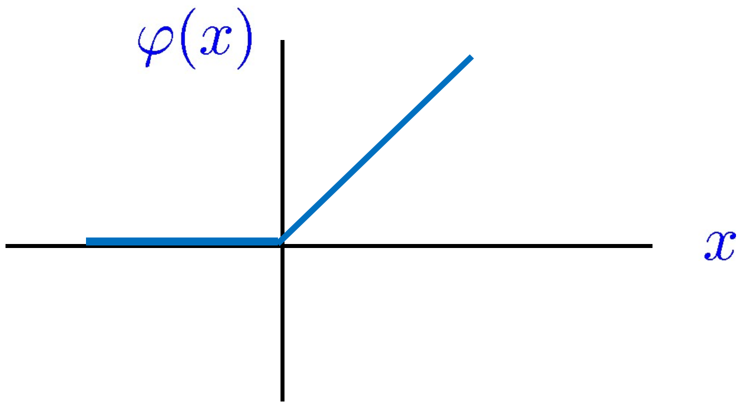
Examples: ReLU (Rectified Linear Unit) $\varphi(x) = \max\{0, x\}$

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

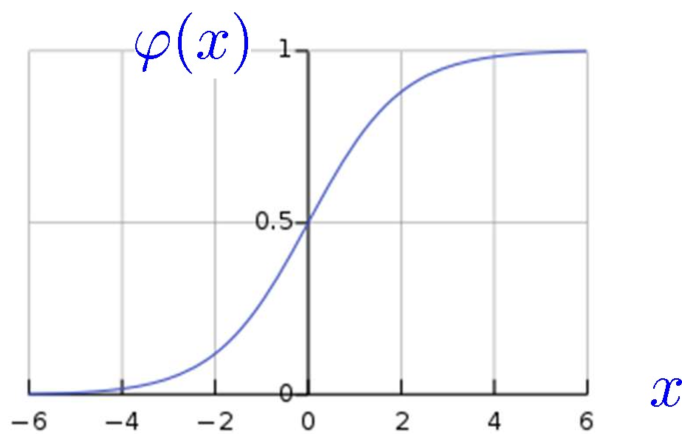
Examples: ReLU (Rectified Linear Unit) $\varphi(x) = \max\{0, x\}$

Sigmoid $\varphi(x) = \dfrac{e^x}{1 + e^x}$

ReLU:



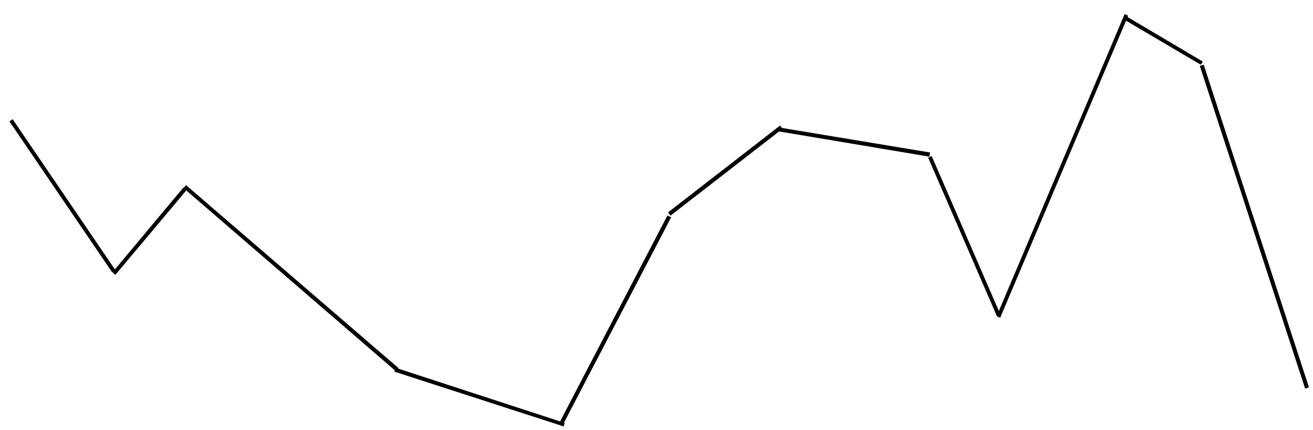
Sigmoid:



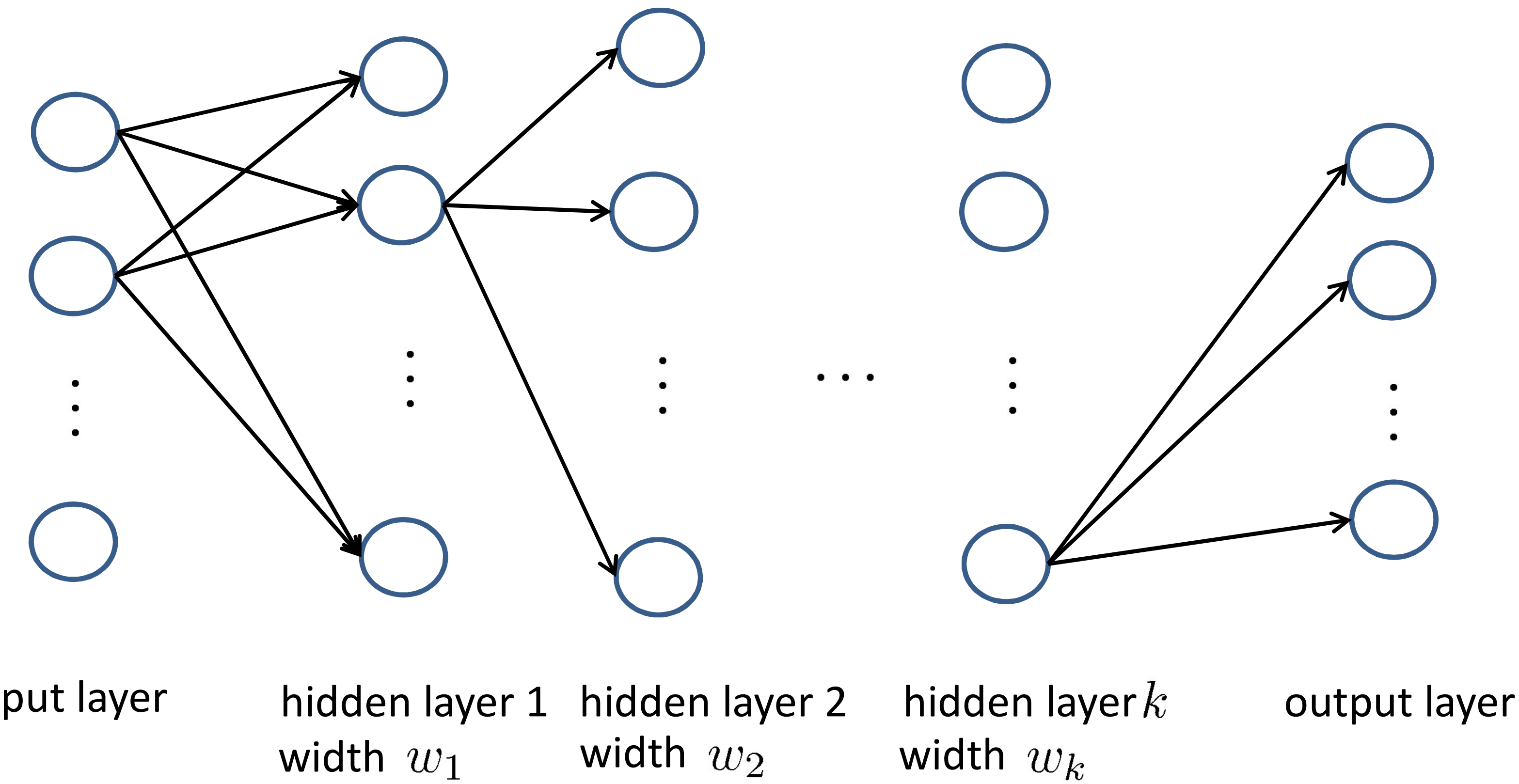
Some theoretical results for DNNs

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 piecewize 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:

hidden layers

size

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 $(\boldsymbol{x}, \boldsymbol{y})$, find weights for the best fit function.

Theorem: (Arora et al, 2018) Let n and w be natural numbers, and $(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^D, \boldsymbol{y}^D)$ 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:

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

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

Polynomial in data size D for fixed n en w.

Bienstock, Muñoz, Pokutta (2018) generalize and extend the training results og Arora et al (2018).

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.

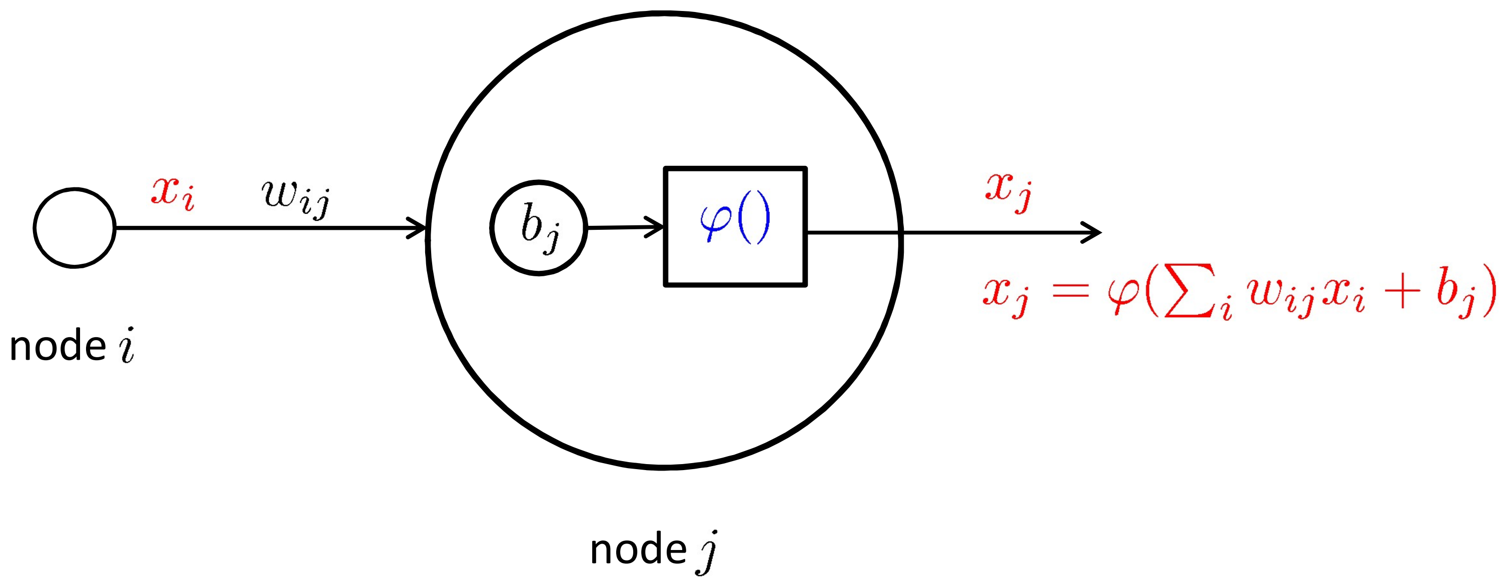
Make use of Bienstock & Muñoz reformulation of non-convex problems to approximate LPs.

Adversarial machine learning and MIP

M. Fischetti, J. Jo (2018). Deep neural networks and mixed integer linear optimization. Constraints 23:296-309.

Model a ReLU-DNN as a MIP:

What goes on in the hidden layers?



$\varphi() : \mathbb{R} \to \mathbb{R}$ is called an "activation" function

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

Adversarial machine learning and MIP

M. Fischetti, J. Jo (2018). Deep neural networks and mixed integer linear optimization. Constraints 23:296-309.

Model a ReLU-DNN as a MIP:

Recall: ReLU (Rectified Linear Unit) $x_j = \varphi(\sum_i w_{ij} x_i + b_j)$

$\varphi(x) = \max\{0, x\}$

$$\left.\begin{array}{c} \displaystyle\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - z_j^k, \\ x_j^k,\ z_j^k \geq 0 \end{array}\right] \begin{array}{rcl} k & = & 1, \ldots, K \\ j & = & 1, \ldots, n_k \end{array}$$

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

M. Fischetti, J. Jo (2018). Deep neural networks and mixed integer linear optimization. Constraints 23:296-309.

$$\left.\begin{array}{c}\sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} = x_j^k - z_j^k, \\ x_j^k,\ z_j^k \geq 0\end{array}\right] \begin{array}{rcl} k & = & 1, \ldots, K \\ j & = & 1, \ldots, n_k \end{array}$$

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 $z_j^k \leq 0$

$$x_j^k \leq M(1 - y_j^k)$$
$$z_j^k \leq M y_j^k$$
$$y_j^k \in \{0, 1\}$$

MIP model:

$$\min \sum_{k=0}^{K} \sum_{j=1}^{n_k} c_j^k x_j^k + \sum_{k=0}^{K} \sum_{j=1}^{n_k} \gamma_j^k y_j^k$$

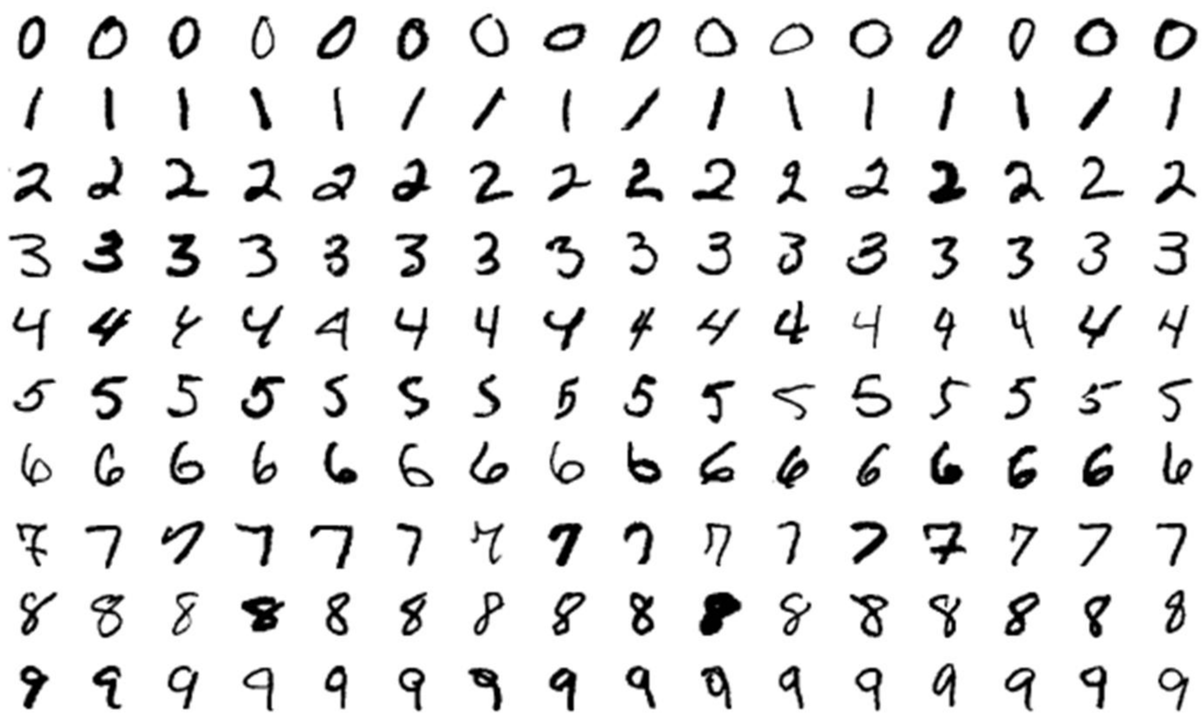
$$\left.\begin{aligned}
\text{s.t.} \quad \sum_{i=1}^{n_{k-1}} w_{ij}^{k-1} x_i^{k-1} + b_j^{k-1} &= x_j^k - z_j^k, \\
x_j^k,\ z_j^k &\geq 0 \\
x_j^k &\leq M(1 - y_j^k) \\
z_j^k &\leq M y_j^k \\
y_j^k &\in \{0, 1\}
\end{aligned}\right\} \begin{array}{rcl} k & = & 1, \ldots, K \\ j & = & 1, \ldots, n_k \end{array}$$

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 $\tilde{x}^0$ correcty 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:

$$x_{5+1}^K \geq 1.2 x_{j+1}^K, \;\; j \in \{0,1,2,3,4,6,7,8,9\}$$

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

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

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:

C.-H. Cheng et al. (2017). Maximum resilience in artificial neural networks. In: D. D'Souza, K. Narayan Kumar (eds.) Automated technology for verification and analysis, Springer pp 251-268.

T. Serra et al. (2017). Bounding and counting linear regions of deep neural networks. CoRR arXiv:1711.02114.

V. Tjeng, R. Tedrake (2017). Verifying neural networks with mixed integer programming. CoRR arXiv:1711.07356.

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”?