INSTITUTO
DE INFORMÁTICA
UFRGS

LEARNING TO SOLVE $\mathcal{NP}$-COMPLETE PROBLEMS

Marcelo Prates Pedro Avelar Henrique Lemos
Luis Lamb Moshe Vardi

November 13, 2018

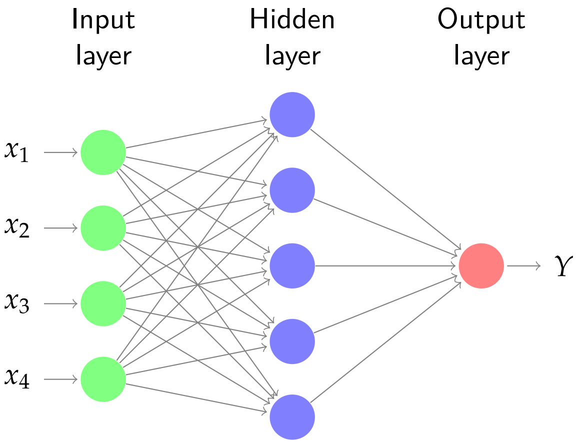
OUTLINE

NEURAL SYMBOLIC LEARNING

- Deep learning has been successfully applied to a wide range of domains (images, audio, NLP)
- Until now, however, DL has mostly succeeded in applications where the inputs are numerical signals (i.e. pixels)
- How do you design a DL model to learn on a molecule / social network / relational database / multiparticle system / symbolic expressions?
- Classical DL does not understand relational input!
- Combining DL with combinatorial generalization is seen as a key step forward for AI [Battaglia *et al.*2018]

ARTIFICIAL NEURAL NETWORKS

- A typical ANN is composed of multiple neurons connected according to a given topology



RECURRENT NEURAL NETWORKS

- A recurrent neural network unit is just a neural unit with a self-loop: it is fed with its own output

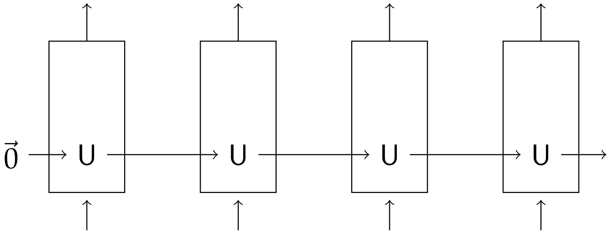


Figure 1: Pictorial representation of the unrolling of a recurrent unit "U" into four iterations. Because the parameters of all four blocks are shared, the resulting network can be thought of as iterating the same operation that many times.

PARAMETER SHARING IN NEURAL NETWORKS

- Parameter sharing is powerful because it allows us to exploit **redundancy** in the problem domain
- Instead of a RNN, we could just instantiate 4 unrelated neural units and connect them. The model could learn the correct association for 4 fixed iterations, but would not learn the "for loop"
- This would be equivalent to writing a C program that repeats the same subroutine 4 times. It works, but does not generalize for $\neq 4$
- Key insight: if your problem has redundancy, you can use parameter sharing to do the same work while spending much less parameters (easier training)

CONVOLUTIONAL NEURAL NETWORKS

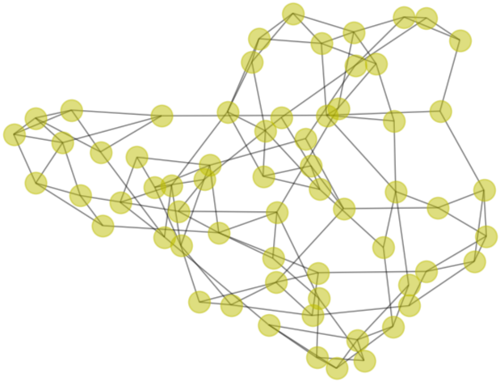
- Convolutional neural networks employ redundancy in space (i.e. learn a number of kernels which will be repeatedly applied throughout the image)

CONVOLUTIONAL NEURAL NETWORKS

- Convolutional neural networks employ redundancy in space (i.e. learn a number of kernels which will be repeatedly applied throughout the image)
- CNNs assume a grid structure (matrix of pixels). What if this topology changed with the input?

MESSAGE PASSING IN GRAPHS

- Suppose you want to perform some computation on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
- Envision a distributed algorithm in which nodes communicate with one another to learn about their neighborhood



MESSAGE PASSING IN GRAPHS

- To archive this knowledge, we must allocate a “memory” x_i for each node $v_i \in \mathcal{V}$ (which can be initialized randomly or with zeros)
- Then each node v_i can compute a “message” $msg(x_i)$ to send to each of its neighbors
- Upon receiving a set of messages $X = \{msg(x_j) \mid v_j \in \mathcal{N}(v_i)\}$ from its neighbors v_j, each node v_i can update its memory through some “update” function $x_i' \leftarrow update(x_i, X)$

MESSAGE PASSING IN GRAPHS

- Conceivably, if the functions $msg()$ and $update()$ are well-designed, upon many iterations each node could be able to “enrich” its memory with valuable information about its neighborhood
- So what if $msg()$ and $update()$ are *learned*?
- Concretely: what if $msg()$ is modelled as a MLP and $update()$ as a RNN (specifically LSTM)?

GRAPH NEURAL NETWORKS

- If we instantiate $msg() : \mathbb{R}^d \rightarrow \mathbb{R}^d$ as a MLP and $update() : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ as a LSTM, we can use them as "neural modules" to assemble different neural architectures
- For a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$:
  1. Assign a multidimensional vector $x_i \in \mathbb{R}^d$ to each vertex $v_i \in \mathcal{V}$ and collect all vectors into a matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$
  2. Compute a $(|\mathcal{V}| \times d)$ matrix of messages $MSGS \leftarrow msg(\mathbf{X})$
  3. Multiply by $\mathcal{G}$'s adjacency matrix: $\mathbf{A} \times MSGS$
     - This yields a $|\mathcal{V}| \times d$ matrix where the i-th line is the sum of all the messages received by vertex v_i
  4. Now pass through the update function: $update(\mathbf{X}, \mathbf{A} \times MSGS)$
     - This yields a $|\mathcal{V}| \times d$ matrix where the i-th line is the updated vector of vertex v_i

GRAPH NEURAL NETWORKS

- Note that this process is completely differentiable and can be accomplished just by function composition and matrix multiplication
- Also note that in principle we can iterate this process by "unrolling" (i.e. composing the function with itself n times)
- What we obtain is a *end-to-end differentiable* message-passing algorithm on graphs, which outputs a set of refined vertex embeddings at the end

GRAPH NEURAL NETWORKS

- If we want to learn to compute for example a decision problem on graphs, we can just "reduce" the matrix $\mathbf{X}$ to a scalar. For example by appending a mean operator at the end of the pipeline, like $\text{mean}(\mathbf{X})$
- Finally: if we perform gradient descent on $loss = (Y - \text{mean}(\mathbf{X}))^2$, we can learn to solve a decision problem on graphs, given enough examples
- Note that the input for each problem is just a adjacency matrix $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$, and the output is just a boolean $Y \in \{0, 1\}$

GRAPH NEURAL NETWORKS

- In summary: the core of a GNN is just a RNN applied over a matrix multiplication between an adjacency matrix and a function applied linewise on a matrix of embeddings. Iterated many times
- Or, in math talk:

$$\mathbf{X}^{(t+1)} \leftarrow update(\mathbf{X}^{(t)}, \mathbf{A} \times msg(\mathbf{X}^{(t)})) \qquad (1)$$

NEUROSAT

- [Selsam *et al.*2018]: You can learn to solve boolean satisfiability (SAT) with the following GNN:
- $\mathbf{A} \in \{0,1\}^{|\mathcal{C}|\times|\mathcal{L}|}$ is an adjacency matrix between clauses and literals

$$\begin{aligned} \mathbf{C}^{(t+1)} &\leftarrow update(\mathbf{C}^{(t)}, \mathbf{A} \times msg_{L\to C}(\mathbf{L}^{(t)})) \\ \mathbf{L}^{(t+1)} &\leftarrow update(\mathbf{L}^{(t)}, \mathbf{A}^T \times msg_{C\to L}(\mathbf{L}^{(t)}), F(\mathbf{L}^{(t)})) \end{aligned} \tag{2}$$

- Clauses send messages to literals, literals send messages to clauses and literals send messages to their negated counterparts (i.e. x_1 and $\neg x_1$)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Given a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a number $C \in \mathbb{R}$:
- Train a GNN to decide whether $\mathcal{G}$ admits a Hamiltonian route with cost no larger than C

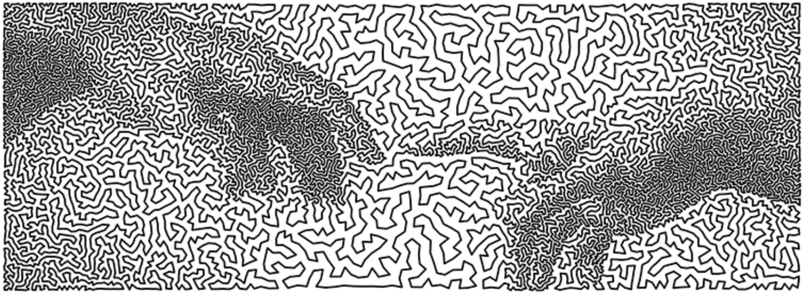


Figure 2: Recreation of Michelangelo's "The Creation of Adam" by Robert Bosch as a TSP solution

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Two problems to address:
  1. How to feed the model with numerical information (edge weights) in addition to relational (graph) data?
  2. How to tell the model the weight of each edge if only vertices have embeddings?
- Maybe assign embeddings to edges!
- Also initialize each edge embedding with its corresponding weight

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Vertices send messages to edges of which they are endpoints (i.e. vertex v sends messages to $\forall (v_1, v_2) \mid v = v_1 \lor v = v_2$)
- Each edge (v_1, v_2) sends a message to v_1 and a message to v_2
- Over t_{max} message-passing iterations, each edge will become enriched with relational & numerical information relevant to the TSP problem
- This information is destilled into a scalar which is interpreted as that edge's "vote" (i.e. the probability with which it thinks that a route exists)
- Finally, all votes are averaged into the final prediction

Algorithm 1 Graph Neural Network TSP Solver

```
1: procedure GNN-TSP($\mathcal{G} = (\mathcal{V}, \mathcal{E}), C$)
2:     $\mathbf{A}[i,j] \leftarrow 1$ iff $(\exists v' | e_i = (v_j, v', w) \vee e_i = (v', v_j, w)) | \forall e_i \in \mathcal{E}, \forall v_j \in \mathcal{V}$
3:     $\overset{(1)}{\mathbf{E}}[i] \leftarrow \underset{init}{E}(w, C) \mid \forall e_i = (s, t, w) \in \mathcal{E}$
4:     for $t = 1 \ldots t_{max}$ do
5:         $\overset{(t+1)}{\mathbf{V}_h}, \overset{(t+1)}{\mathbf{V}} \leftarrow V_u(\overset{(t)}{\mathbf{V}_h}, \mathbf{A} \times \underset{msg}{E}(\overset{(t)}{\mathbf{E}}))$
6:         $\overset{(t+1)}{\mathbf{E}_h}, \overset{(t+1)}{\mathbf{E}} \leftarrow E_u(\overset{(t)}{\mathbf{E}_h}, \mathbf{A}^T \times \underset{msg}{V}(\overset{(t)}{\mathbf{V}}))$
7:     $\mathbf{E_{logits}} \leftarrow \underset{vote}{E}\left(\overset{t_{max}}{\mathbf{E}}\right)$
8:     prediction $\leftarrow$ sigmoid($\langle \mathbf{E_{logits}} \rangle$)
```

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \rightarrow \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \to \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)
  3. A function $V_{msg} : \mathbb{R}^d \to \mathbb{R}^d$ to compute messages to send from vertices to edges (MLP)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \rightarrow \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)
  3. A function $V_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from vertices to edges (MLP)
  4. A function $E_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from edges to vertices (MLP)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \to \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)
  3. A function $V_{msg} : \mathbb{R}^d \to \mathbb{R}^d$ to compute messages to send from vertices to edges (MLP)
  4. A function $E_{msg} : \mathbb{R}^d \to \mathbb{R}^d$ to compute messages to send from edges to vertices (MLP)
  5. A function $V_u : \mathbb{R}^{2d} \to \mathbb{R}^{2d}$ to compute an updated vertex embedding (plus an updated RNN hidden state) given the current RNN hidden state and a message (LSTM)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \rightarrow \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)
  3. A function $V_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from vertices to edges (MLP)
  4. A function $E_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from edges to vertices (MLP)
  5. A function $V_u : \mathbb{R}^{2d} \rightarrow \mathbb{R}^{2d}$ to compute an updated vertex embedding (plus an updated RNN hidden state) given the current RNN hidden state and a message (LSTM)
  6. A function $E_u : \mathbb{R}^{2d} \rightarrow \mathbb{R}^{2d}$ to compute an updated edge embedding (plus an updated RNN hidden state) given the current RNN hidden state and a message (LSTM)

A GRAPH NEURAL NETWORK FOR THE DECISION TSP

- Upon training, the model learns:
  1. A single $\mathbb{R}^d$ vector which will be used to initialize each vertex embedding
  2. A function $E_{init} : \mathbb{R}^2 \rightarrow \mathbb{R}^d$ to compute an initial edge embedding given an edge weight w and a target cost C (MLP)
  3. A function $V_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from vertices to edges (MLP)
  4. A function $E_{msg} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ to compute messages to send from edges to vertices (MLP)
  5. A function $V_u : \mathbb{R}^{2d} \rightarrow \mathbb{R}^{2d}$ to compute an updated vertex embedding (plus an updated RNN hidden state) given the current RNN hidden state and a message (LSTM)
  6. A function $E_u : \mathbb{R}^{2d} \rightarrow \mathbb{R}^{2d}$ to compute an updated edge embedding (plus an updated RNN hidden state) given the current RNN hidden state and a message (LSTM)
  7. A function $E_{vote} : \mathbb{R}^d \rightarrow \mathbb{R}$ to compute a logit probability given an edge embedding (MLP)

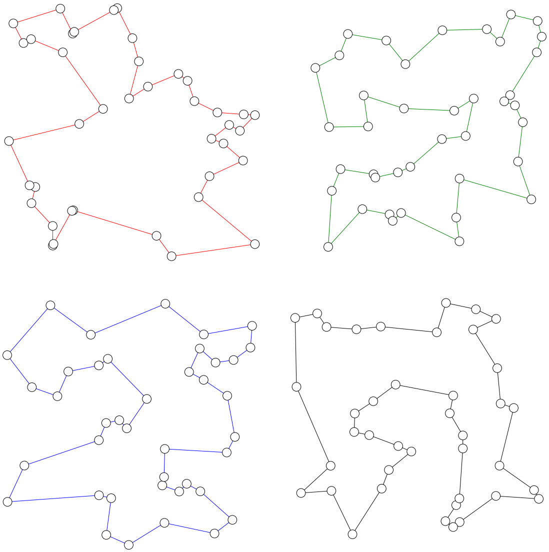
TRAINING

- OK, so how do we train it?
- Idea: show the model very similar instances with opposite answers
- Given a graph $\mathcal{G}$ with optimal TSP cost C^*:
- The model should answer **NO** for inputs $X^- = (\mathcal{G}, 0.98C^*)$ and **YES** for inputs $X^+ = (\mathcal{G}, 1.02C^*)$
- So: create random graphs and feed each graph to the model two times; one with -2% and the other with $+2\%$ deviation from the optimal cost
- Advantage: we can create our own training instances!

TRAINING

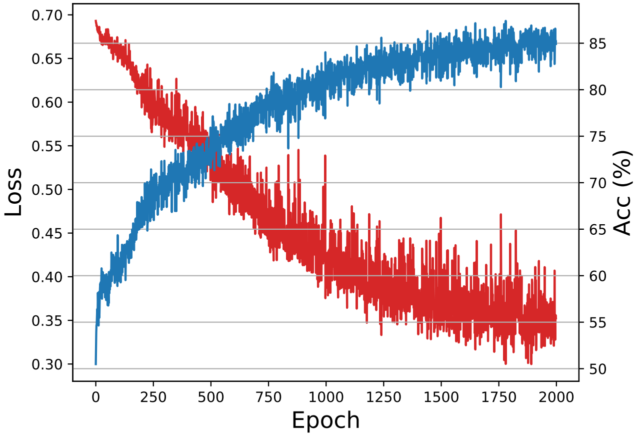
- Batches of $2 \times 8 = 16$ instances
- 128 batches per training epoch
- 2^{20} training instances, to drastically reduce the possibility of overfitting (we're expected to cycle through the entire dataset only after $2^{20-7-4} = 512$ epochs)
- Each instance is a complete, euclidean graph obtained from n points uniformly distributed in the $\frac{\sqrt{2}}{2} \times \frac{\sqrt{2}}{2}$ square
- The size of each graph is chosen uniformly at random between 20 and 40 vertices: $n \sim \mathcal{U}(20, 40)$

TRAINING



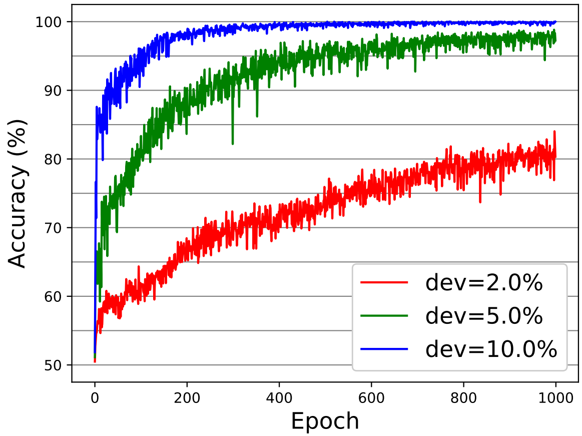
TRAINING

- With $-2\%, +2\%$ deviations we can achieve 85% training accuracy (80% test) in 2000 epochs



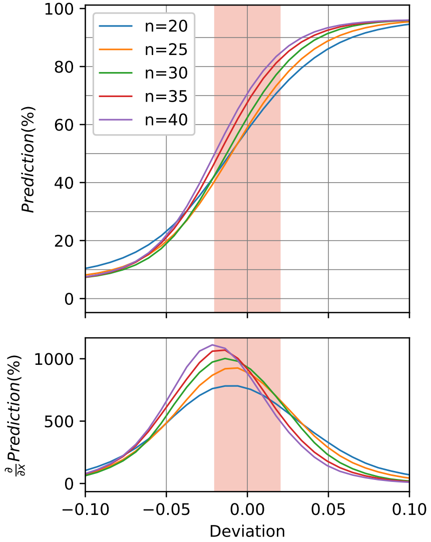
TRAINING

- The greater the deviation, the easier it is to train



EXTRACTING ROUTE COSTS

- The probability with which the model thinks that there is a route undergoes something reminiscent of a phase transition as a function of the deviation from the optimal cost
- Curves for big n are higher because the larger the size the larger the probability of (proportionally) cheap routes

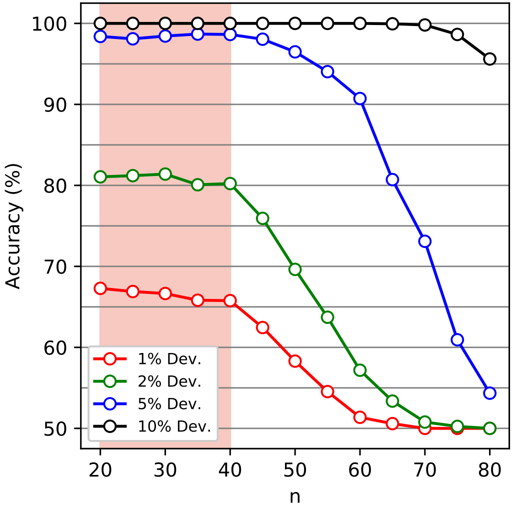


EXTRACTING ROUTE COSTS

- We can exploit these “acceptance curves” to approximate the TSP optimal cost:
- Intuitively, the closer the model is to absolute uncertainty (50%), the closer we are to the optimal cost
- So: compute lower and upper bounds to the optimal cost and perform a binary search
- We can approximate costs within 1.5% of the optimal cost through on average 8.9 iterations of binary search

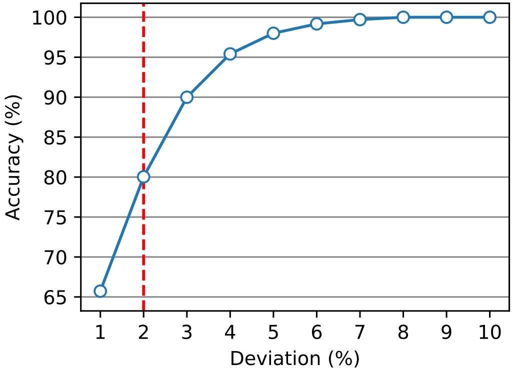
PERFORMANCE ON LARGER INSTANCES

- The model is able to generalize to some extent to larger instances
- Accuracy decreases progressively until the baseline of 50% with increasing n
- Larger deviations yield higher accuracy



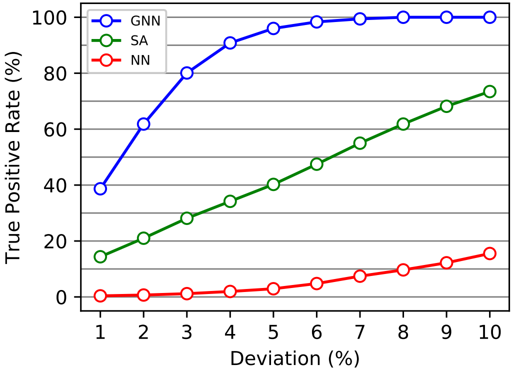
GENERALIZING TO LARGER DEVIATIONS

- Accuracy increases with deviation, approaching 100% for $dev > 8\%$



BASELINE COMPARISON

- We compare the trained model with a predictor for the decision TSP obtained from the solutions yielded by a 1) Nearest Neighbor strategy and 2) a Simulated Annealing strategy (2-exchange)



GENERALIZING TO OTHER DISTRIBUTIONS

- The model (which is trained with euclidean instances) can generalize somewhat to random edge weights **iff** the instances satisfy the triangle inequality

Deviation	Accuracy (%)		
	Euc. 2D	Rand. Metric	Rand.
1	66	57	50
2	80	64	50
5	98	82	50
10	100	96	50

Table 1: Test accuracy averaged over 1024 n-city instances with $n \sim \mathcal{U}(20, 40)$ for varying percentual deviations from the optimal route cost for differing random graph distributions: bidimensional euclidean distances, “random metric” distances and random distances.

GENERALIZING TO OTHER DISTRIBUTIONS

Instance	Size	Relative Deviation (%) GNN	SA
ulysses16[1]	16	-22.80	$+1.94$
ulysses22[1]	22	-27.20	$+1.91$
eil51	51	-18.37	$+18.07$
berlin52	52	-8.73	$+21.45$
st70	70	-11.87	$+14.47$
eil76	76	-13.91	$+19.24$
kroA100	100	-2.00	$+30.73$
eil101	101	-9.93	$+20.46$
lin105	105	$+6.37$	$+17.77$

1 These instances had their distance matrix computed according to Haversine formula (great-circle distance).

IMPLEMENTATION AND REPRODUCIBILITY

- Github repository: https://github.com/machine-reasoning-ufrgs/TSP-GNN
- Our library eases the prototyping of GNNs, which can be described briefly

```
gnn = GraphNN(
    {
        'V': d,
        'E': d
    },
    {
        'EV': ('E','V')
    },
    {
        'V_msg_E': ('V','E'),
        'E_msg_V': ('E','V')
    },
    {
        'V': [
            {
                'mat': 'EV',
                'msg': 'E_msg_V',
                'transpose?': True,
                'var': 'E'
            }
        ],
        'E': [
            {
                'mat': 'EV',
                'msg': 'V_msg_E',
                'var': 'V'
            }
        ]
    })
```

FURTHER WORK

- Ideally we want to obtain Hamiltonian routes from the model (even though it was not explicitly trained to do so)
- [Selsam *et al.*2018] were able to extract satisfying assignments from the literal embeddings by performing 2-clustering, but we were not successful. Hypothesis: euclidean graphs may be too easy
- Train with other graph distributions (variable connectivity, random weights etc.)
- In principle we can train the model to compute (approximated) optimal costs directly

REFERENCES I

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al.
Relational inductive biases, deep learning, and graph networks.
arXiv preprint arXiv:1806.01261, 2018.

Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill.
Learning a SAT solver from single-bit supervision.
arXiv preprint arXiv:1802.03685, 2018.