Deep Reinforcement Learning for Vehicle Routing Problems

Kevin Tierney

kevin.tierney@uni-bielefeld.de
Professor for Decision and Operation Technologies
Department of Management Science and Business Analytics
Bielefeld University

EURO PhD School Data Science Meets
Combinatorial Optimization – September 4, 2023

UNIVERSITÄT
BIELEFELD

Faculty of Business Administration
and Economics

About me

Prof. Dr. Kevin Tierney

UNIVERSITÄT BIELEFELD

($\geq$ 2018) Professor Universität Bielefeld

PADERBORN UNIVERSITY
The University for the Information Society

($<$ 2018) Juniorprofessor Uni Paderborn

Ph.D. IT University of Copenhagen

IT University of Copenhagen

Sc.M. Brown University

BROWN

B.S. Rochester Institute of Technology

R·I·T

Outline

Session 1: Learning and routing

- **Motivation** & managing expectations
- **Background:** ML, DNNs, CO
- **GPU:** strengths and weaknesses
- **Getting started:** Routing with ML
- **Attention!**

Session 2: Training some models

- The **rl4co** package
- NLNS, EAS, SGBS

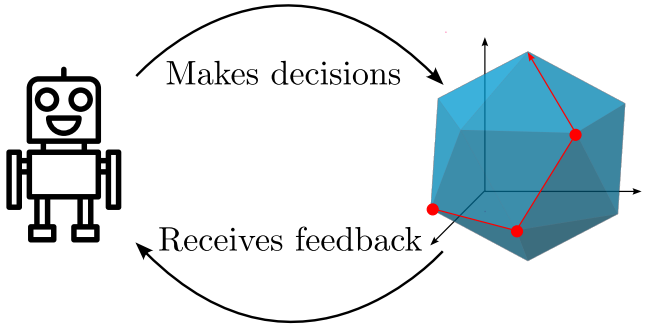
Session 3: Notebooks, outlook, advice

- EAS in rl4co
- **Outlook** & conclusion

Thanks to all of my co-authors on these works: André Hottung, Jinho Choo, Yeong-Dae Kwon Jihoon Kim, Jeongwoo Jae, and Youngjune Gwon.

Combining ML and optimization: towards automated development

Idea: Learn how to **solve an optimization problem** through **reinforcement learning**.



Combining ML and optimization: towards automated development

Idea: Learn how to **solve an optimization problem** through **reinforcement learning**.

Advantages

- **Lower barrier to entry:** OR problem turns into a data science problem
- **Customized heuristics** for the type of instances at hand

Disadvantages

- Learning phase **computationally expensive**; requires at least a GPU
- Algorithmic **interpretability** likely **low**er than for handcrafted solutions

Managing expectations for learning to optimize

Expectation

- Query a model, get an optimal solution
- Solve any CO problem a user provides
- Scale to any problem size

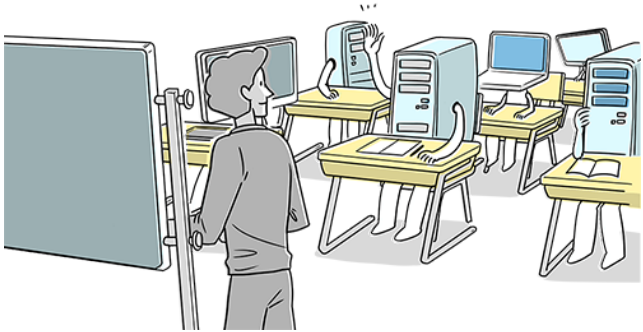
Reality

- Query a model, get a solution, not necessarily good
- Solve simple classes of problems ("easy" side constraints!)
- Scale... somewhat.

"Machine learning and combinatorial optimization"

Background: ML & DNNs

Supervised learning



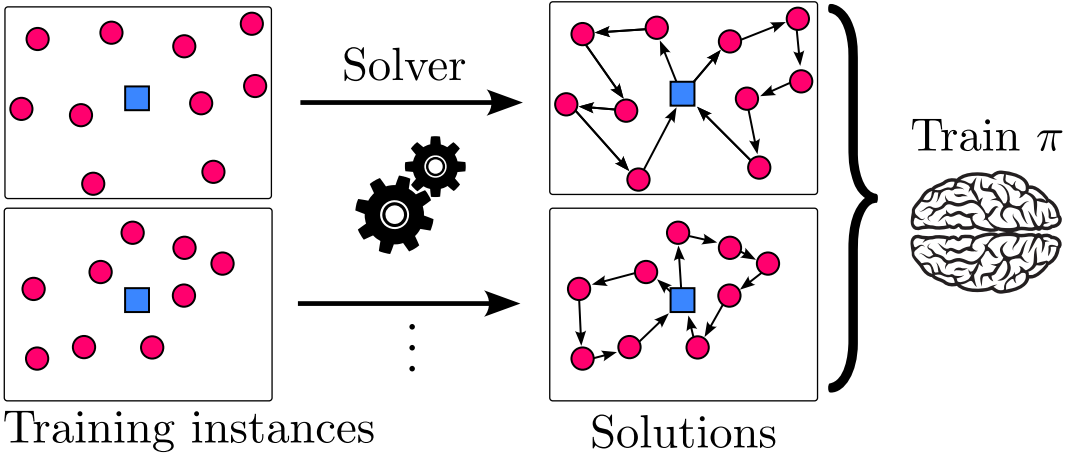
Source unknown

Terminology:

- n training examples $(\mathbf{x}_i, y_i)$
- $\mathbf{x}_i$ is the **feature vector**
- y_i is the **label**
- $\pi : X \to Y$ is a **model**
- $L : Y \times Y \to \mathbb{R}$ is the **loss function** (how good is g at predicting $\hat{y}_i$ from x_i?)

Goal: Learn a model π from the data (x_i, y_i) such that the loss L is minimized.

Supervised learning in the context of routing



Features?

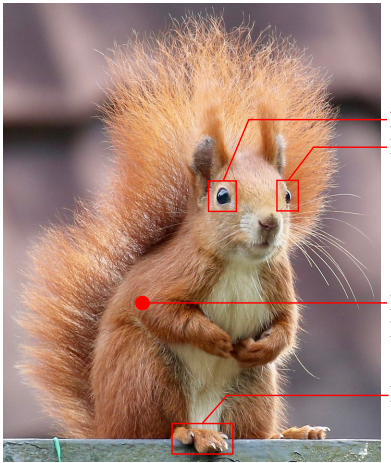
a.k.a attributes or variables

Definition: Feature

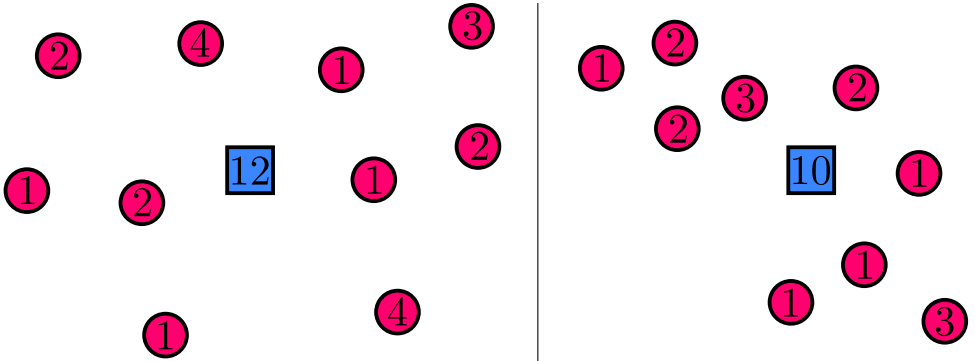
A feature is an individual measurable property or characteristic of a phenomenon. – Bishop (2006)

Why?

- Differentiate different instances/examples from each other!



Features for a VRP

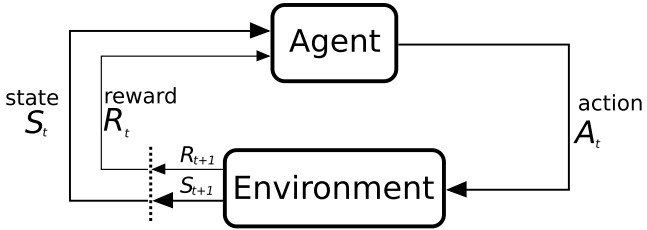


Instance 1 Instance 2

Given: Euclidean CVRP instances with **demands** and **vehicle capacity**, as indicated.

Task: Identify features to differentiate the instances.

Reinforcement learning



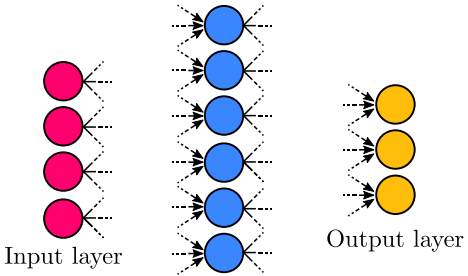
Wikipedia user EBatlleP

In RL, you are **not provided** $(\mathbf{x}_i, y_i)$.

Goal: Determine a policy π for the agent to perform "actions" to minimize a loss function L

Routing context: Learn where to go next without any previously solved instances!

Deep neural network: overview

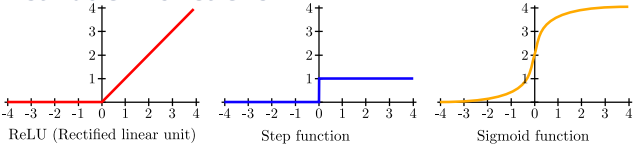


Edges transform data according to specified functions

Feedforward NN:

- **Input:** fixed input size / statistics about problem
- **Hidden layers:** multi-layer perceptron
- **Output:** Softmax over available actions

Activation functions:



Why use DNNs?

Strengths:

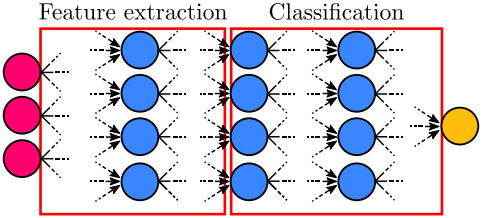
- Supports structured data
- GPU-based training/execution
- Batching

Weaknesses:

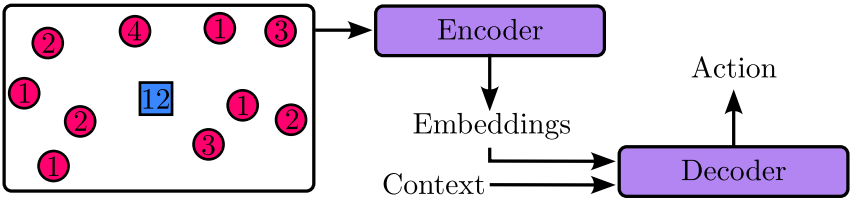
- Require large amounts of data
- Prone to overfitting
- Non-trivial to determine architecture / hyperparameters

Primary strength for solving VRPs/-COPs:

Automatic feature extraction



The transformer architecture



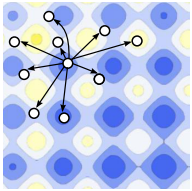
- **Encoder:** Assign vectors to the components of the input
- **Context:** Tells the decoder what part of the problem to address
- **Decoder:** Translates embeddings and the query into an action
- **Action:** What to do next! (node to visit, etc.)

DNN experts will notice that I've kind of made up this part. This image is meant to be a general view of the encoder/decoder architecture as we will need it for routing problems.

"discrete optimization polytope, geometric shape"

Background: Optimization

Exact vs. heuristic optimization



(Modified from original; Original from Wikipedia User Tos.)

Heuristic algorithm:

- No guarantees
- Fast runtime

Integrating ML: (e.g.)

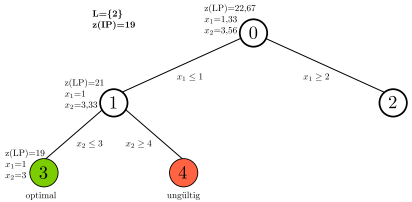
- Choose next neighbor
- Restart strategy

Exact algorithm:

- Guarantees optimality
- Reports infeasibility

Integrating ML:

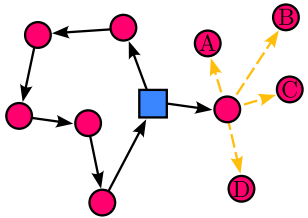
- Learn to branch
- Learn to cut



Heuristic optimization: construction vs. improvement

Construction

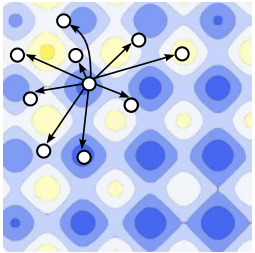
"Builds" a solution one component at a time until it is complete



Examples: GRASP, Ant colony optimization, ...

Improvement

Searches a "neighborhood" of similar solutions.



Examples: Local search, large neighborhood search, ...

GPUs for Optimization

"Graphics card"

CPU vs. GPU

https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (Brian Caulfield)

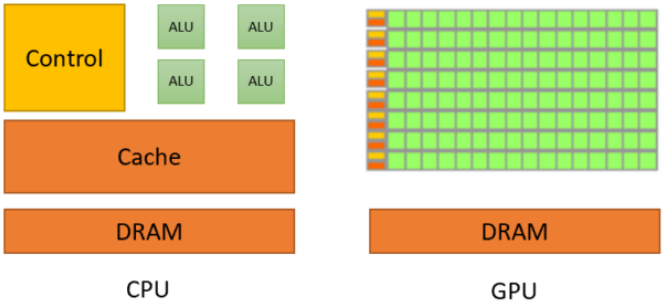
Eric Gaba – Wikimedia Commons: Sting

- General purpose: “broad instruction sets”
- Contains a few cores
- Low latency
- Specialized for **serial processing**

- Originally specialized on graphics
- Many cores
- High throughput
- Can perform thousands of operations at once

User GBPublic_PR (flickr)

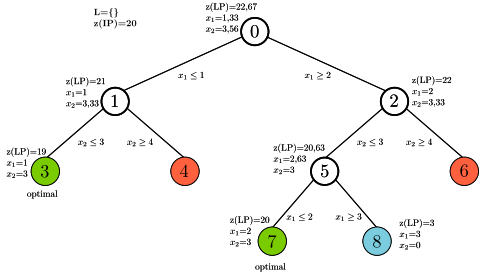
CPU vs. GPU



Pradeep Gupta (Nvidia)

Summary: **GPUs** are strong at performing **the same computation many times simultaneously**, whereas CPUs specialize at complicated, sequential computations.

A conundrum for optimization



GPU parallelization does **not** lend itself to branch-and-bound or metaheuristics

- Different operations in different branches
- The GPU wants to do the **same operation** on all threads
- Latency between CPU/GPU makes short/small computations on GPU not worth it

The conundrum: How to **efficiently** use a GPU for optimization?

A first attempt at solving the conundrum

Bach, Hasle, and Schulz (2019). "Adaptive Large Neighborhood Search on the Graphics Processing Unit" (EJOR)

Idea: Apply large neighborhood search (LNS) on the GPU

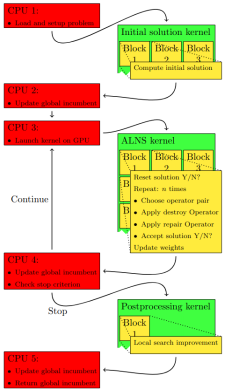
Challenge: Diverse LNS operators can't be easily streamlined for GPU

Solution:

- Specialized, GPU-adjusted operators
- Run the same operator in parallel on multiple solutions

Drawback:

- Complicated heuristic
- Hard to generalize to any problem



GPU-ALNS algorithm (Bach et al. (2019))

"A wizard telling delivery trucks on a street where to go with magic"

Solving VRPs with DNNs

History

Not complete; selected "hits"

Hopfield and Tank (1985) – Solve TSP with a neural network

Glover, 1986 – Target analysis method presented

Gee and Prager – "Limitations" of NNs on TSP

Budinich, 1996 – Self-organizing neural network for TSP

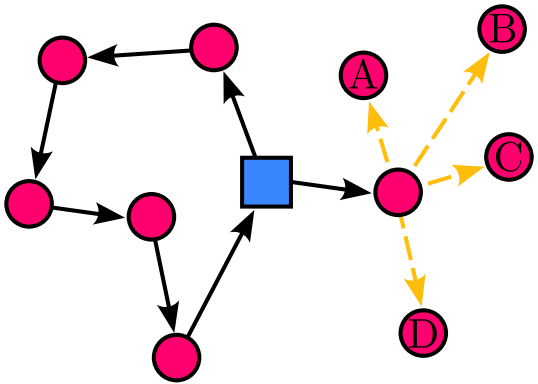
Talavan and Yanez (2002) – Scales the hopfield network to 1000 node TSP

Vinyals et al. 2015 – Pointer network introduced

Moral: This field isn't new. What is new, is the great performance.

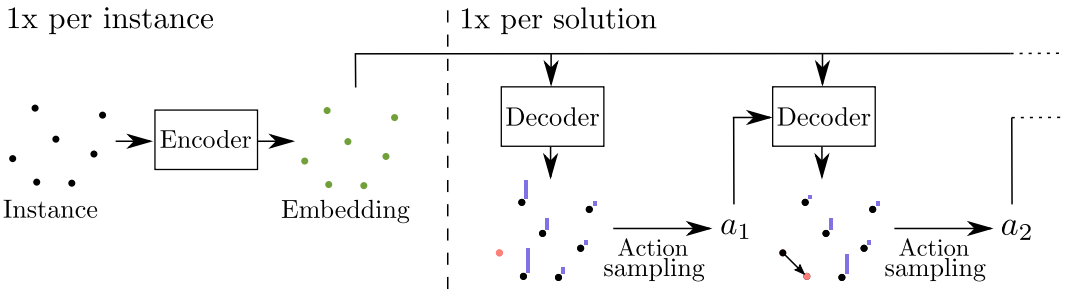
Solving the CVRP: Construction

RL construction: create a solution through a **sequential decision process**



Decide where to go next: Ask the DNN to **construct** a solution!

Encoder/decoder architecture for routing problems

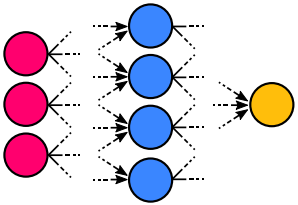


Advantage of architecture: (vs. feedforward) Accepts variable sized input!

More information on the DNNs used for routing: Kool et al. (2018) "Attention, learn to solve routing problems!"

Training: Supervised learning or DRL?

How can we train the model's weights?



Supervised learning

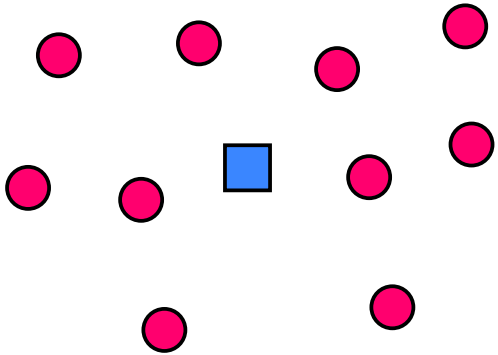
- **Pro:** Can train on optimal or very good solutions (can speed up training)
- **Con:** "Traditional" OR approach necessary

Reinforcement Learning

- **Pro:** No existing algorithm necessary
- **Con:** Approach will start out rather dumb and must become smart

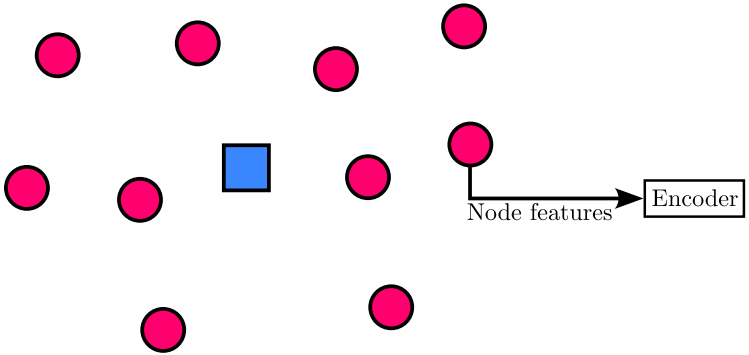
Loss/Reward function: objective function of the solution

Summary so far: generating a solution for the CVRP



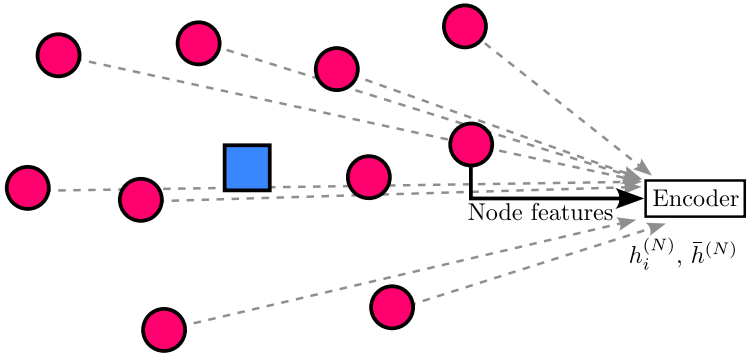
*Note: some details missing; see later slides

Summary so far: generating a solution for the CVRP



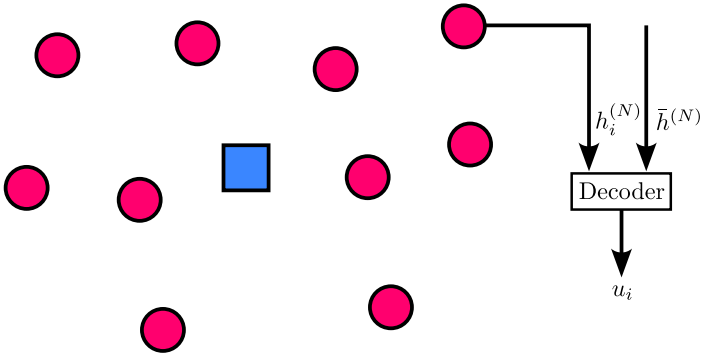
*Note: some details missing; see later slides

Summary so far: generating a solution for the CVRP



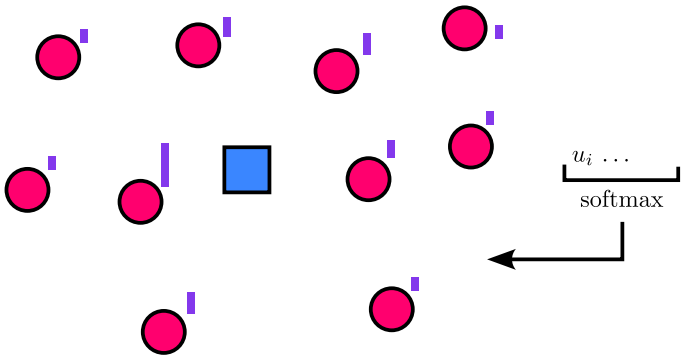
*Note: some details missing; see later slides

Summary so far: generating a solution for the CVRP



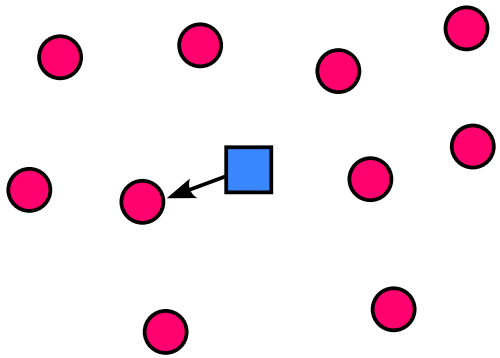
*Note: some details missing; see later slides

Summary so far: generating a solution for the CVRP



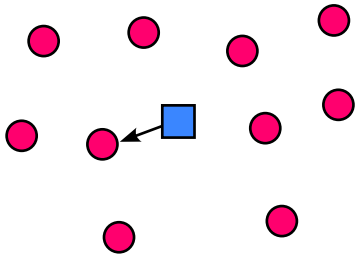
*Note: some details missing; see later slides

Summary so far: generating a solution for the CVRP



*Note: some details missing; see later slides

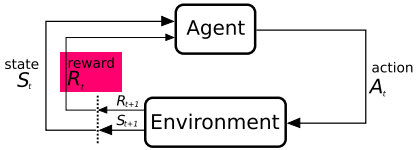
Summary so far: generating a solution for the CVRP



Now **repeat the process** ignoring *(masking)* the node that has been visited.

Note: Some methods update their embeddings based on the current route(s). *Note: some details missing; see later slides

Loss/Reward functions



Wikipedia user EBatlleP (modified)

"Standard RL"

Discounted sum of rewards:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \dots$$

- Prevent agent from focusing only on short-term gain

RL4CO

Reward: Just use the solution's objective function value!

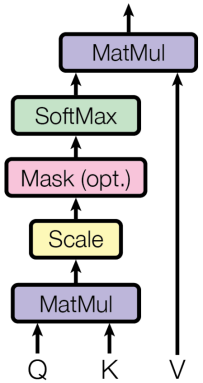
- Reason: We achieve complete solutions; discounting thus unnecessary

"Attention is all you need. The context is machine learning transformers."

Attention and transformers

Understanding attention

Figures/math from "Attention is all you need" (Vaswani et al. (2017))



Dims: d_k, d_k, d_v

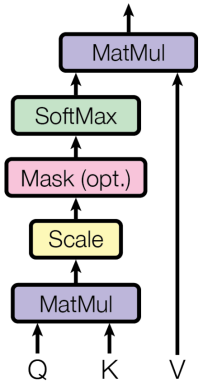
Goal: Focus on "important" parts of the input data.

Inputs:

- **Query (Q):** What are we "looking for"?
- **Key (K):** Elements used to assess the relevance or importance compared to the query.
- **Value (V):** Contains the actual information or content that will be retrieved if the key matches the query.

Understanding attention

Figures/math from "Attention is all you need" (Vaswani et al. (2017))



Dims: d_k, d_k, d_v

Components:

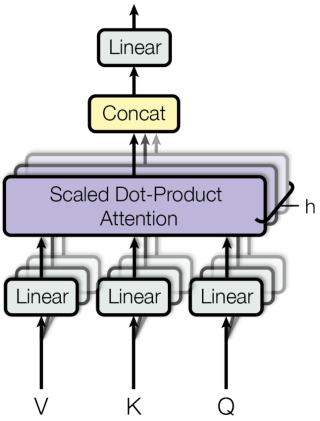
- MatMul Matrix product of two arrays.
  $\rightarrow$ Quadratic (d_k^2 complexity)
- Scale Divide by $\sqrt{d_k}$
- Mask Ignore certain parts of the input
- SoftMax Form a probability distribution

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Model weights: See next slide.

Multi-headed attention (MHA)

Figures/math from "Attention is all you need" (Vaswani et al. (2017))



Idea: Focus "attention" on multiple aspects of the input, rather than just one.

Components:

- Linear Linear transformation, see any layer in the feed-forward network previously shown
- Scaled Dot-Product Attention The attention mechanism from the previous slide
- Concat Concatenate everything together

Weight matrices: W^V, W^K, W^Q, W^O

$$\mathrm{MHA}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h) W^O$$

$$\text{where } \mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Attention! Learn to solve routing problems.

Kool, van Hoof, and Welling (ICLR 2019)

ATTENTION, LEARN TO SOLVE ROUTING PROBLEMS!

Wouter Kool
University of Amsterdam
ORTEC
w.w.m.kool@uva.nl

Herke van Hoof
University of Amsterdam
h.c.vanhoof@uva.nl

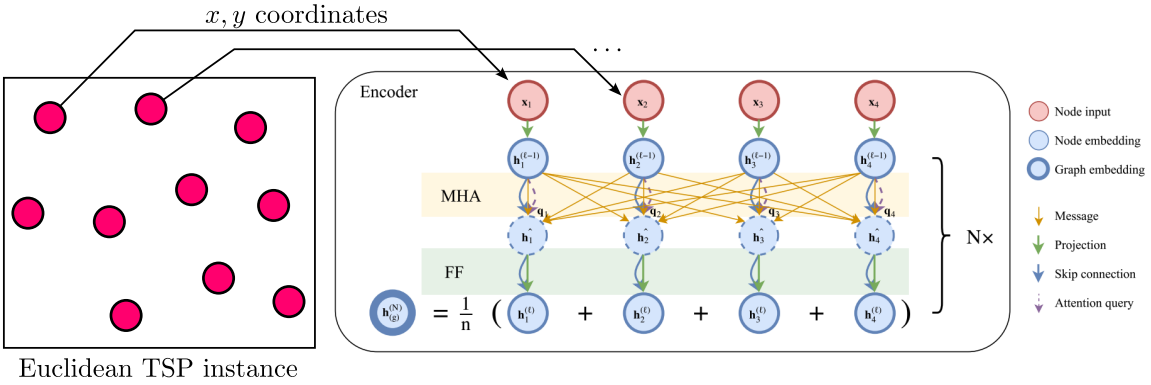
Max Welling
University of Amsterdam
CIFAR
m.welling@uva.nl

Key contribution: Shows how to apply the transformer model to routing problems.

From now on, this paper will be referred to as **AM**

AM: Encoder

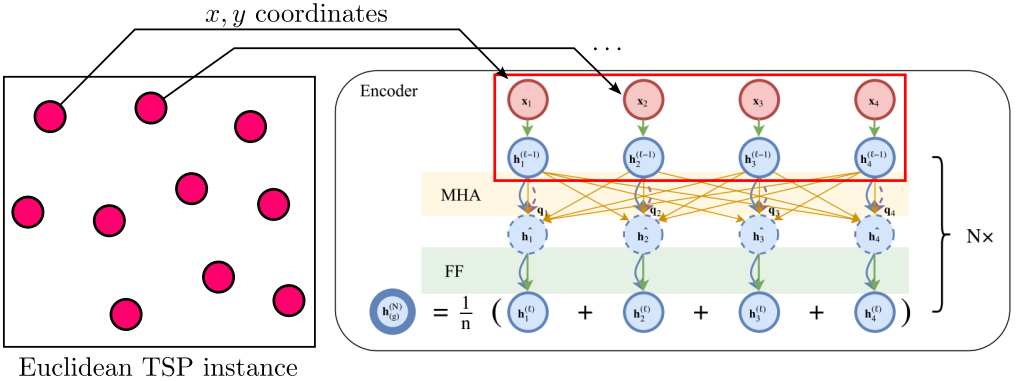
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



1. Insert problem features into input layer

AM: Encoder

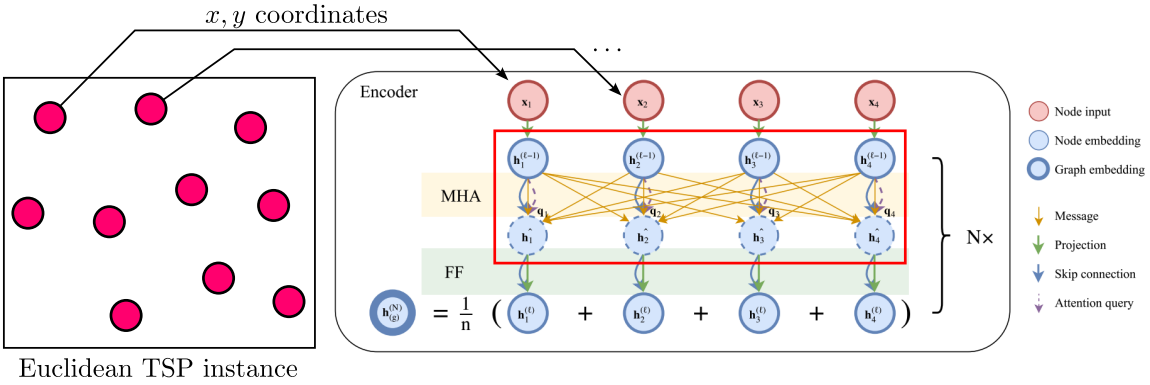
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



2. **Embedding:** Project node features into d_h dimensions (Linear)

AM: Encoder

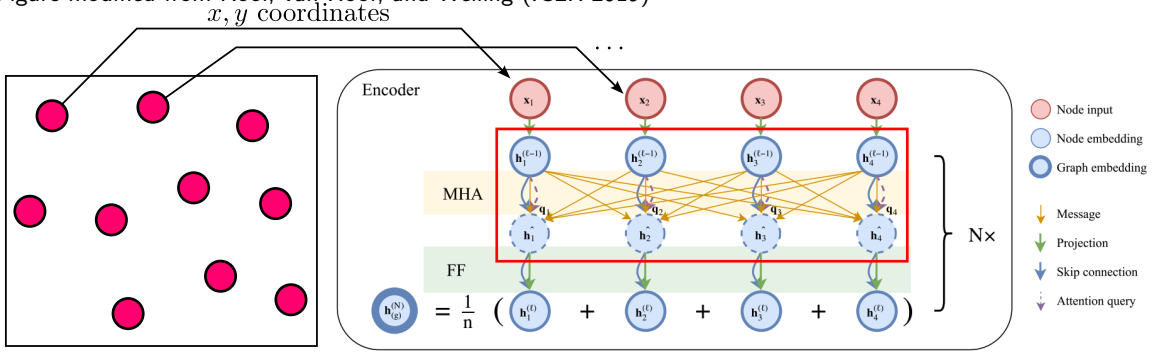
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



3. **MHA:** Pass the embeddings into the attention layer: $Q = K = V$

AM: Encoder

Figure modified from Kool, van Hoof, and Welling (ICLR 2019)

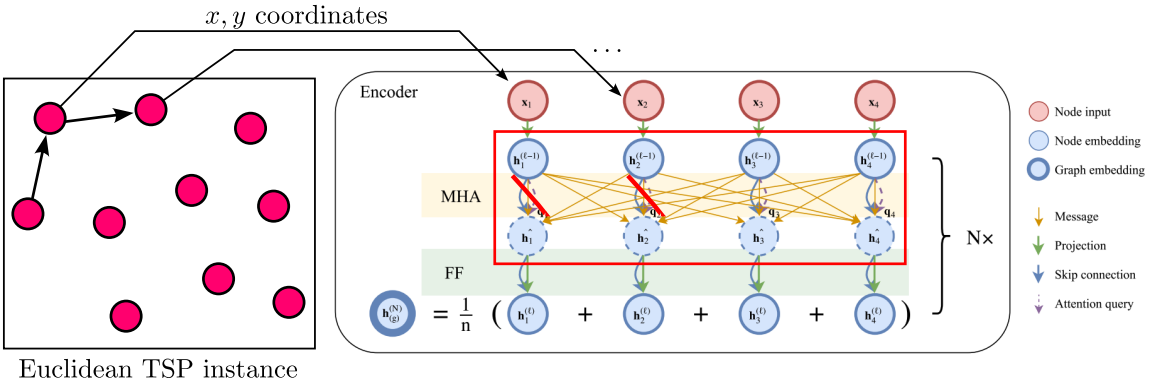


Euclidean TSP instance

3. **MHA:** Pass the embeddings into the attention layer: $Q = K = V$
4. **Skip connections:** MHA can be partially skipped (provide less-processed information deeper in the network; reduce vanishing gradient problem)

AM: Encoder

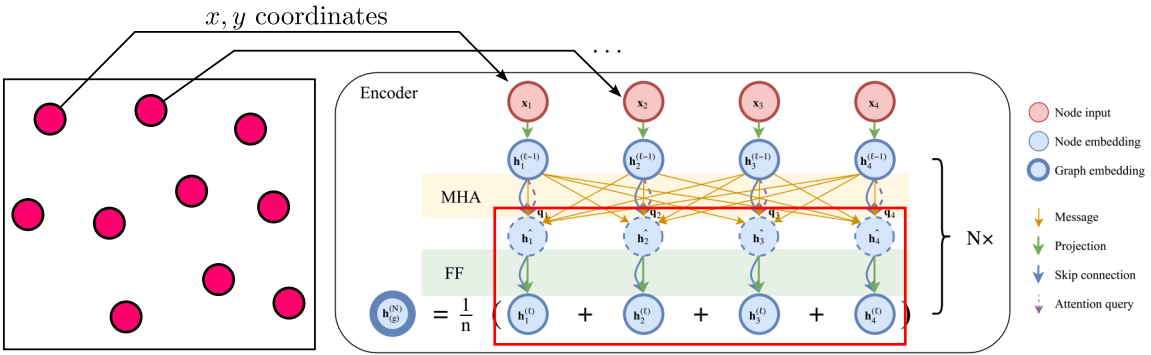
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



4. **Masking:** Mask any nodes already on the tour (*Note: in AM not actually used; encoder only runs once!)

AM: Encoder

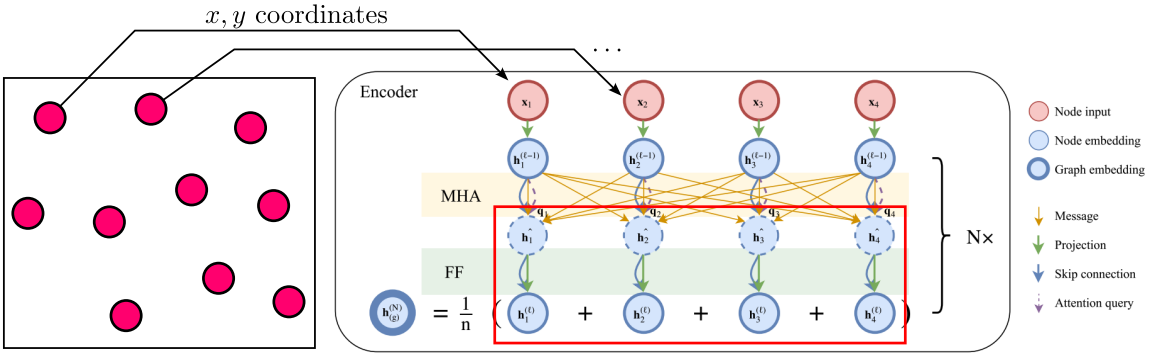
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



5. **Feedforward:** Transform MHA output with ReLu activation.

AM: Encoder

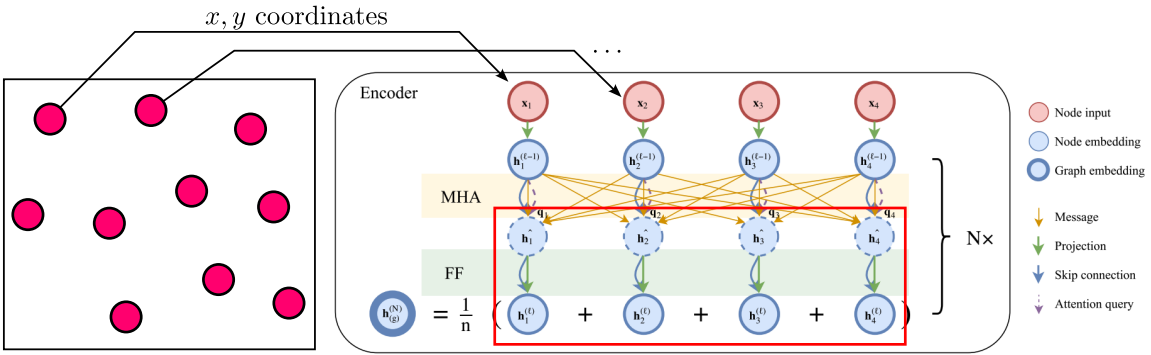
Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



6. **Skip connections** (again)

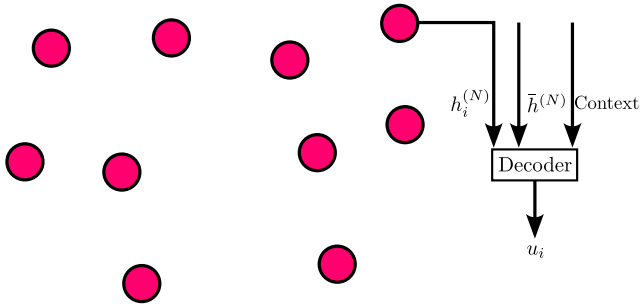
AM: Encoder

Figure modified from Kool, van Hoof, and Welling (ICLR 2019)



7. **Batch Normalization:** Stabilize outputs

AM: Encoder output

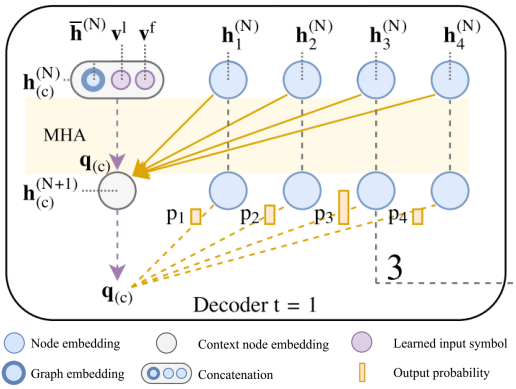


Remember the overview example from before?

The nodes are now encoded. Time to decode.

AM: Decoder

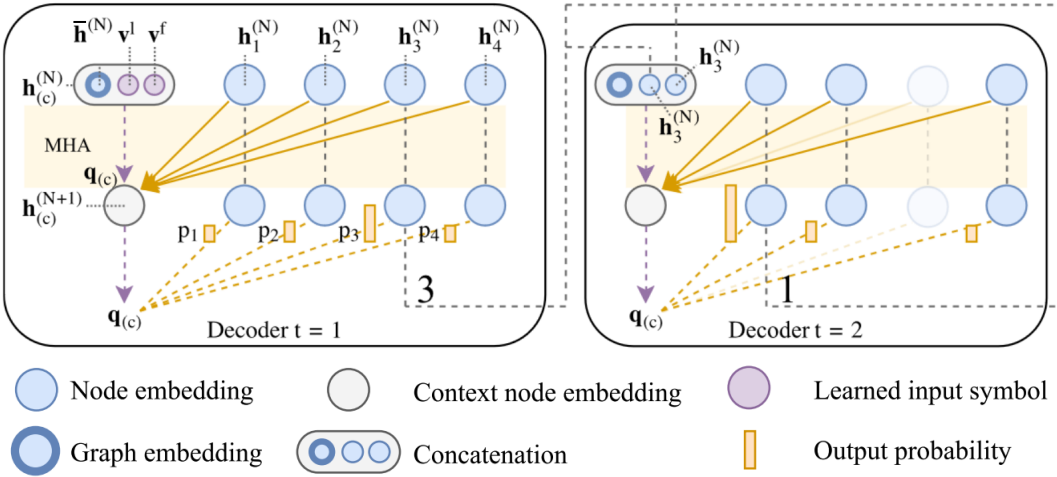
Figures modified from Kool, van Hoof, and Welling (ICLR 2019)



- **Context node:** Extra, augmented node to assist in decoding (for efficiency; see paper)
- $\bar{\mathbf{h}}^{(N)}$ – Graph embedding
- $\mathbf{h}_i^{(N)}$ – Node embedding
- $\mathbf{h}_{(c)}^{(N)}$ – Embedding of the **context node**
- $\mathbf{v}^l, \mathbf{v}^f$ – Trainable parameters; placeholders for first iteration
- $\mathbf{q}_{(c)}$ – MHA Query (context node)
- p_i – Probability of selecting node i

AM: Decoder

Figures modified from Kool, van Hoof, and Welling (ICLR 2019)



AM: Examining the math

Notation from from Kool, van Hoof, and Welling (ICLR 2019)

The previous attention model defines **stochastic policy** $p(\boldsymbol{\pi}|s)$ where π is a solution and s is a problem instance.

This is **factorized** and **parameterized** by $\boldsymbol{\theta}$ as

$$p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s) = \prod_{t=1}^{n} p_{\boldsymbol{\theta}}\left(\pi_t|s, \boldsymbol{\pi}_{1:t-1}\right)$$

What this says

The probability of a solution π given instance s the joint probability of constructing the solution s given $p_{\boldsymbol{\theta}}$.

AM: Training

Notation from from Kool, van Hoof, and Welling (ICLR 2019)

Training method: REINFORCE with "greedy rollout baseline"

Loss function:

$$\nabla \mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s)} \left[(L(\boldsymbol{\pi}) - b(s)) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s) \right]$$

- $L(\boldsymbol{\pi})$ – Cost of solution $\boldsymbol{\pi}$, e.g., in the TSP the tour length
- $b(s)$ – "Baseline"; Goal is to reduce variance, thus speeding up learning

Options for $B(s)$:

- Actor-critic
- Greedy rollout (Create a solution using the argmax of the decoder output)

AM: REINFORCE (Williams, 1992) algorithm

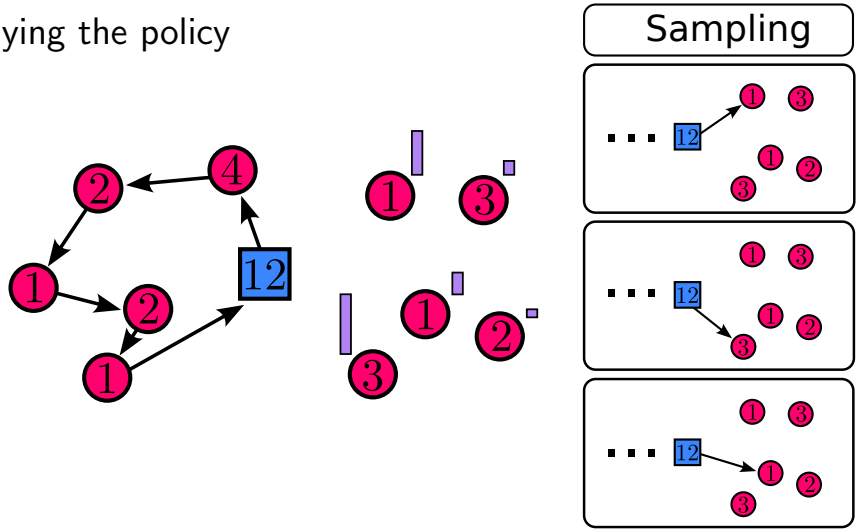
Algorithm 1 from Kool, van Hoof, and Welling (ICLR 2019)

Init $\boldsymbol{\theta}$, $\boldsymbol{\theta}^{\mathrm{BL}} \leftarrow \boldsymbol{\theta}$
for epoch $= 1, \ldots, E$ **do**
  for step $= 1, \ldots, T$ **do**
    $s_i \leftarrow \mathrm{RandomInstance}() \ \ \forall i \in \{1, \ldots, B\}$
    $\boldsymbol{\pi}_i \leftarrow \mathrm{SampleRollout}(s_i, p_{\boldsymbol{\theta}}) \ \ \forall i \in \{1, \ldots, B\}$
    $\boldsymbol{\pi}_i^{\mathrm{BL}} \leftarrow \mathrm{GreedyRollout}(s_i, p_{\boldsymbol{\theta}^{\mathrm{BL}}}) \ \ \forall i \in \{1, \ldots, B\}$
    $\nabla \mathcal{L} \leftarrow \sum_{i=1}^{B} \left(L(\boldsymbol{\pi}_i) - L(\boldsymbol{\pi}_i^{\mathrm{BL}}) \right) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}_i)$
    $\boldsymbol{\theta} \leftarrow \mathrm{Adam}(\boldsymbol{\theta}, \nabla \mathcal{L})$
  end for
  if $\mathrm{OneSidedPairedTTest}(p_{\boldsymbol{\theta}}, p_{\boldsymbol{\theta}^{\mathrm{BL}}}) < \alpha$ **then**
    $\boldsymbol{\theta}^{\mathrm{BL}} \leftarrow \boldsymbol{\theta}$
  end if
end for

Input:

- E – Number of epochs
- T – Steps per epoch
- B – Batch size
- α – Statistical significance
- Adam: Gradient descent algorithm for setting model weights by Kingama and Ba (2015)

AM: Applying the policy



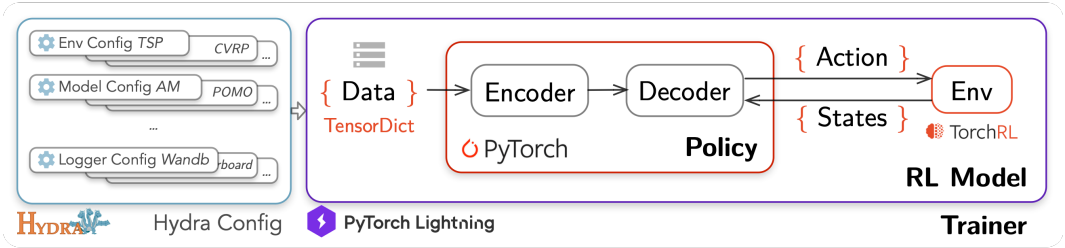
Sampling is naive – We can do better! More later.

RL4CO module

By Federico Berto and Chuanbo Hua and Junyoung Park and Minsu Kim and Hyeonah Kim and Jiwoo Son and Haeyeon Kim and Joungho Kim and Jinkyoo Park

https://github.com/kaist-silab/rl4co

RL4CO architecture



To the notebook!

rl4co notebook: `https://colab.research.google.com/drive/1m8x4vMsFIAnnPLDXeLdCxgYOHtH1BAhn?usp=sharing`

"Jupyter notebook."

Policy Optimization with Multiple Optima for Reinforcement Learning (POMO)

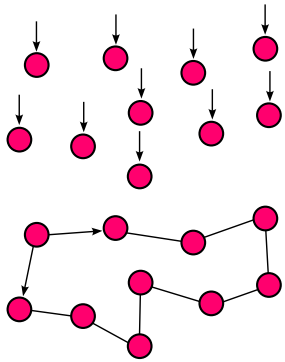
Kwon et al. 2020

Key insight: Symmetries in the solution space cause problems for DRL approaches.

Consider the **TSP:**

1. The starting node is **irrelevant**
2. Hamiltonian cycle $\rightarrow$ two ways to start tour

Solution: Force exploration from all nodes, then roll out



POMO: Shared baseline

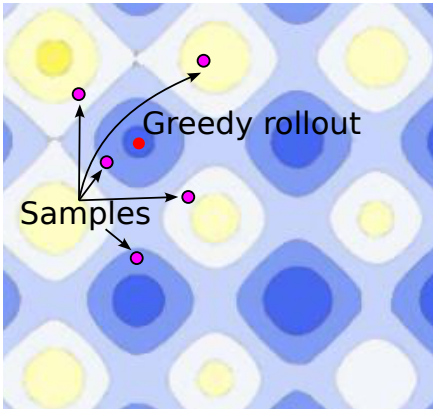
Math from Kwon et al. 2020, adopted with AM notation

AM baseline: Greedy rollout

Potential drawbacks:

- Samples may not surpass greedy rollout
- Highly susceptible to local minima

Figure: None of the samples are better than the greedy rollout, despite one of them being in a new basin of attraction.



(Modified from original; Original from Wikipedia User Tos.)

POMO: Shared baseline

Math from Kwon et al. 2020, adopted with AM notation

Consider our loss function from before*...

$$\nabla \mathcal{L}(\boldsymbol{\theta}|s) = \mathbb{E}_{p_{\boldsymbol{\theta}(\boldsymbol{\pi}|s)}} \left[(L(\boldsymbol{\pi}) - b(s)) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}|s) \right]$$

...and rewrite it, indexing the solutions $\boldsymbol{\pi}^i$ ...

$$\approx \frac{1}{N} \sum_{i=1}^{N} \left[(L(\boldsymbol{\pi}^i) - b(s)) \nabla \log p_{\boldsymbol{\theta}}(\boldsymbol{\pi}^i|s) \right]$$

Shared baseline:

$$b(s) = \frac{1}{N} \sum_{j=1}^{N} L(\boldsymbol{\pi}^j)$$

Idea: Take the average reward of the solutions for each instance!

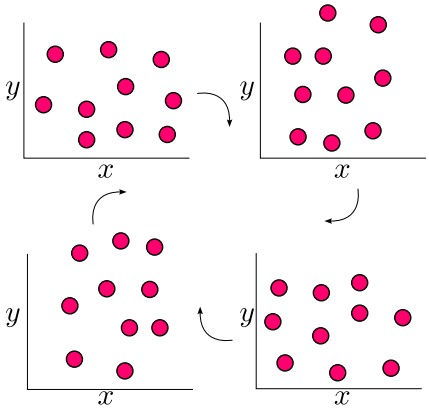
***Note:** I adopt the notation of the AM paper here over the POMO paper for simplicity.

POMO: Instance augmentation

Idea: Exploit instance symmetries to find better solutions

- **Augment** the batch with symmetrical solutions

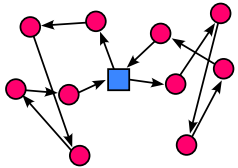
Note: non-trivial on problems with side constraints



"Someone searching for something with a telescope on a boat."

Search

Search



Problem:
What if the solution generated is not good?

Solution 1: Sampling

- **How:** Sample the network's output space and build more solutions
- **Pro:** Highly and trivially parallelizable on the GPU
- **Con:** Quickly diminishing returns

Solution 2: Search

- **How:** Use an "intelligent" high-level heuristic to generate more solutions.
- **Pro:** Various techniques have decent performance
- **Con:** If CPU involved, latency/expensive
- **Pro/Con:** Parallelizable (maybe)

Why search?

(Perceived) goal of (some of the) ML community

Generate an optimal solution to a huge optimization problem in a single pass.

Neural Large Neighborhood Search (NLNS)

Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

- **Insights:**
  1. Constructing from scratch has a low probability of finding great solutions.
  2. Large neighborhood search (LNS) highly successful as a "traditional" OR technique
- **Contribution:** Neural repair operator to construct solutions using a DNN

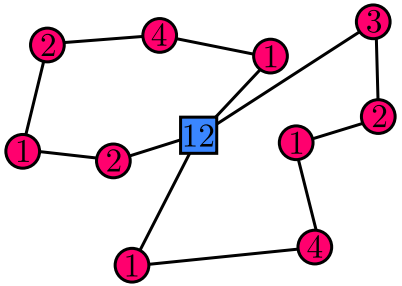
First, **what is LNS?**

Definition: LNS

A metaheuristic search technique based on the concept of iterative *destroy* and *repair*.

LNS: Overview

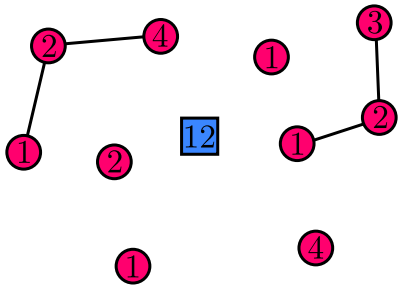
```
1: LNS-Min()
2: s ← StartSolution()
3: s* ← s
4: repeat
5:     s' ← Repair(Destroy(s))
6:     if Accept(s, s') then
7:         s ← s'
8:     end if
9:     if f(s) < f(s*) then
10:        s* ← s
11:    end if
12: until Terminate
13: return s*
```



Generate an initial solution.

LNS: Overview

```
1: LNS-Min()
2: s ← StartSolution()
3: s* ← s
4: repeat
5:     s' ← Repair(Destroy(s))
6:     if Accept(s, s') then
7:         s ← s'
8:     end if
9:     if f(s) < f(s*) then
10:        s* ← s
11:    end if
12: until Terminate
13: return s*
```



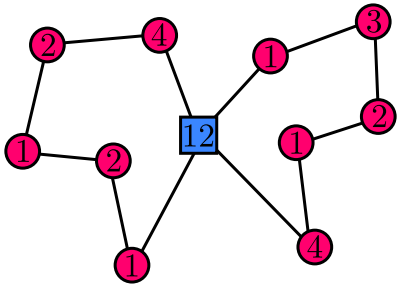
Destroy: remove part of the solution

Example destroy operators:

- Point/geographic destroy
- Destroy (long, short) routes
- SISRs (Christiaens and Vanden Berghe, 2020)

LNS: Overview

```
1: LNS-Min()
2: s ← StartSolution()
3: s* ← s
4: repeat
5:    s' ← Repair(Destroy(s))
6:    if Accept(s, s') then
7:       s ← s'
8:    end if
9:    if f(s) < f(s*) then
10:      s* ← s
11:   end if
12: until Terminate
13: return s*
```



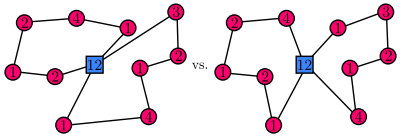
Repair: rebuild a solution

Example repair operators:

- Greedy insertion
- MIP, CP, ...

LNS: Overview

```
1: LNS-Min()
2: s ← StartSolution()
3: s* ← s
4: repeat
5:    s' ← Repair(Destroy(s))
6:    if Accept(s, s') then
7:       s ← s'
8:    end if
9:    if f(s) < f(s*) then
10:      s* ← s
11:   end if
12: until Terminate
13: return s*
```



Accept? Use simulated annealing metropolis criterion to deice whether to accept or not.

- **Note:** other acceptance criteria possible

NLNS: Integrating DRL and LNS

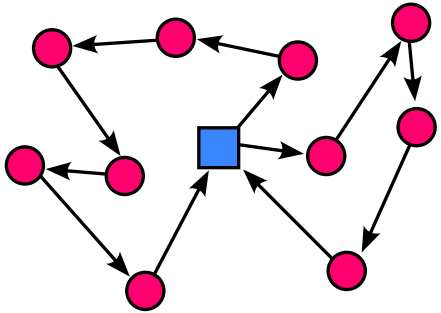
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

Destroy step: Point-based destroy

Repair step:

- Apply argmax policy of model
- Restart rollout at depot until all routes are complete

Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

NLNS: Integrating DRL and LNS

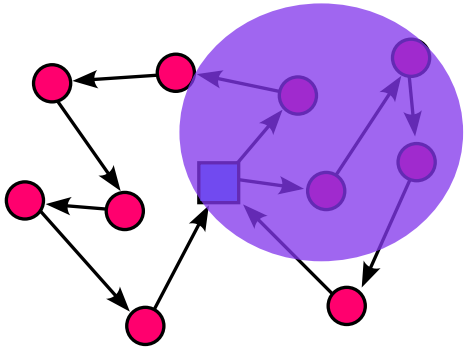
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

★**Destroy step:** Point-based destroy

Repair step:

- Apply argmax policy of model
- Restart rollout at depot until all routes are complete

Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

NLNS: Integrating DRL and LNS

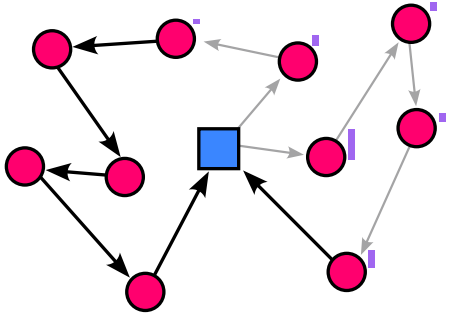
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

Destroy step: Point-based destroy

Repair step:

- ★ Apply argmax policy of model
- Restart rollout at depot until all routes are complete

Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

NLNS: Integrating DRL and LNS

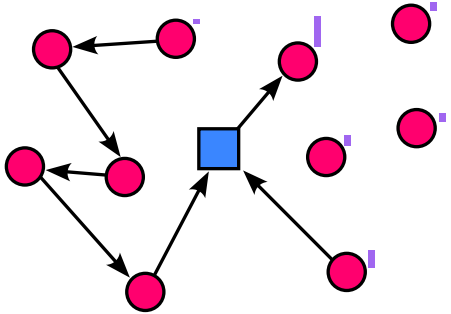
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

Destroy step: Point-based destroy

Repair step:

- ★ Apply argmax policy of model
- Restart rollout at depot until all routes are complete

Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

NLNS: Integrating DRL and LNS

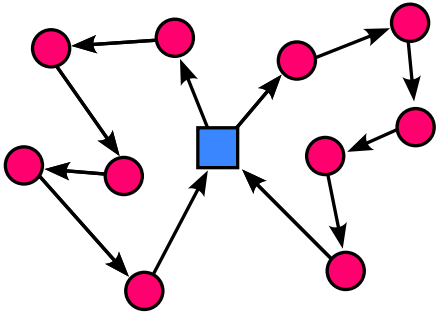
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

Destroy step: Point-based destroy

Repair step:

- Apply argmax policy of model
- ★ Restart rollout at depot until all routes are complete

Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

NLNS: Integrating DRL and LNS

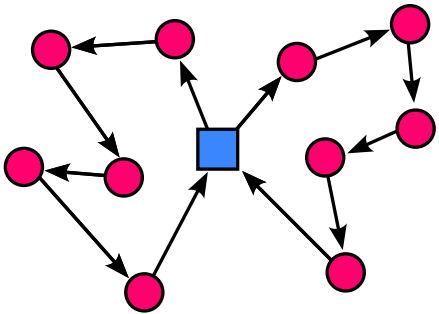
Hottung and Tierney, Artificial Intelligence 2022 / ECAI 2020

Destroy step: Point-based destroy

Repair step:

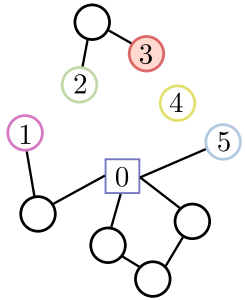
- Apply argmax policy of model
- Restart rollout at depot until all routes are complete

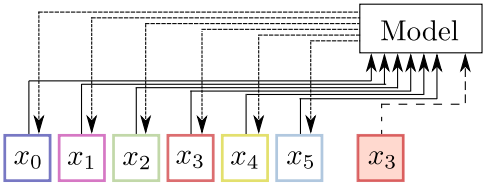
★ Iterate destroy/repair until convergence.



Source code: `https://github.com/ahottung/nlns`

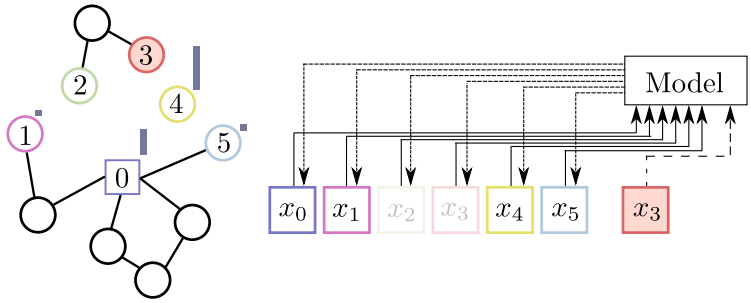
NLNS: The repair operator in a little more detail





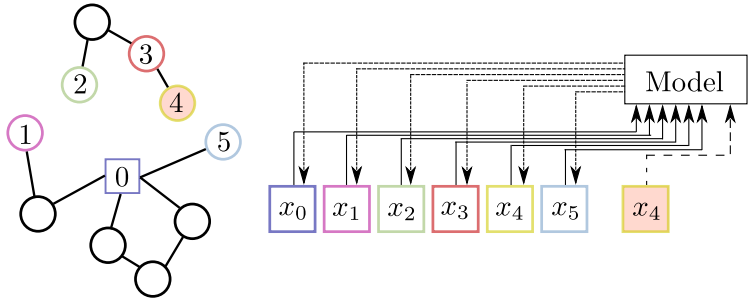
- x_0 represents the depot
- Create an input $(x_1, \ldots, x_5)$ for each end of an incomplete tour not connected to the depot.
- Create an input for the end of a tour that should be connected in the current step (here tour end 3).

NLNS: The repair operator in a little more detail



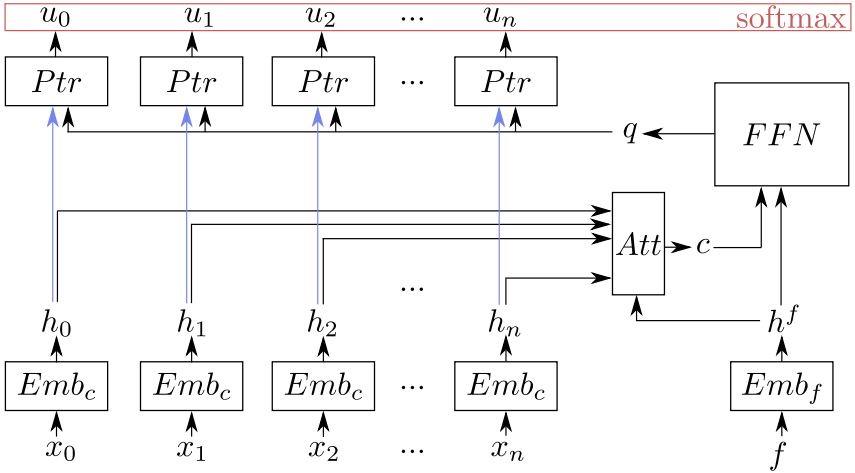
- The inputs x_2 and x_3 are masked.
- The model returns a probability value for each input that can be connected to x_3.

NLNS: The repair operator in a little more detail



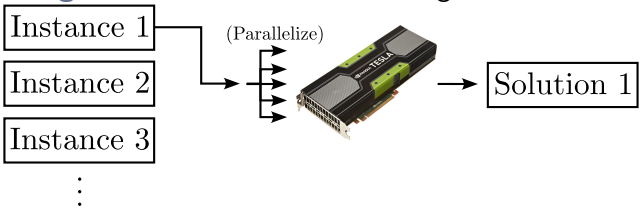
- The end of the tour associated with x_3 is connected to the end of the tour associated with x_4.
- x_4 is selected as the new reference input.

Model architecture

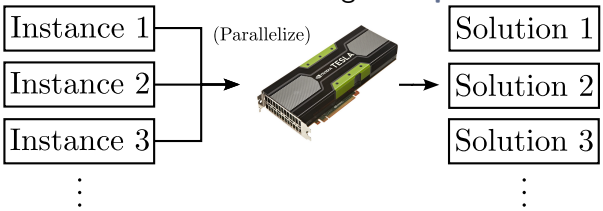


Single instance vs. batch solving

Single instance: Parallelize solving **one** instance.



Batch: Parallelize solving **multiple** instances.



Experimental results: NLNS on CVRP

Single instance solving

		Gap to UHGS			Avg. Time (s)			
Inst. Set	# Cust.	NLNS	LNS	LKH3	NLNS	LNS	LKH3	UHGS
XE_1	100	**0.32%**	0.89%	2.12%	191	192	372	36
XE_2	124	**0.55%**	1.45%	2.33%	191	192	444	64
XE_3	128	**0.44%**	2.05%	0.54%	190	192	122	74
XE_4	161	**0.72%**	8.11%	0.78%	191	194	32	54
XE_5	180	0.58%	2.58%	**0.16%**	191	193	65	86
XE_6	185	**1.09%**	12.14%	1.15%	191	195	100	101
XE_7	199	2.03%	5.78%	**0.72%**	191	195	215	142
XE_8	203	**0.51%**	2.68%	1.37%	612	618	123	103
XE_9	213	2.26%	7.37%	**1.09%**	613	624	66	145
XE_10	218	**0.04%**	1.83%	0.08%	612	616	112	138

UHGS: Vidal et al. (2012) **LKH3**: Helsgaun (2017)

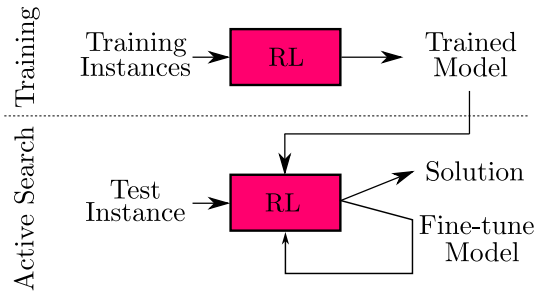
"A detective searching."

Efficient active search

(Efficient) active search

Active Search

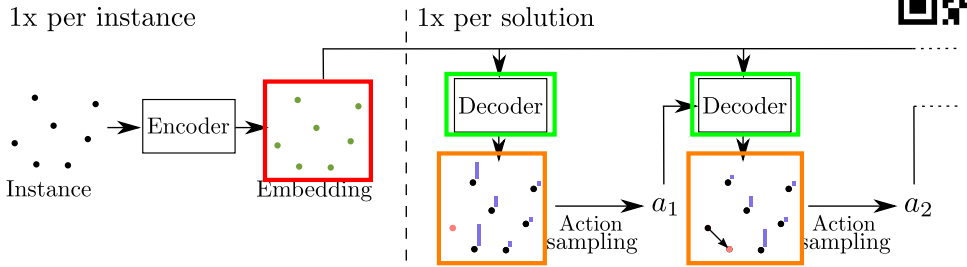
Bello et al. (2016) propose active search, which adjusts the weights of a (trained) model with respect to a single instance at test time using reinforcement learning.



Disadvantage: Fine-tuning a model for each instance is computationally expensive.

Towards efficient active search

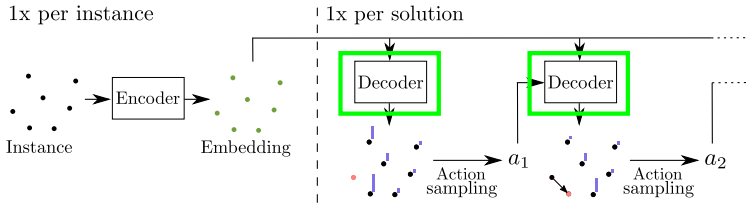
Hottung, Kwon and Tierney (ICLR 2022)



- **Insight:** Why adjust all encoder and decoder weights during active search?
- **Contribution:** We evaluate **three different strategies** that only change a **small subset** of (model) parameters.

Source code: `https://github.com/ahottung/EAS`

EAS Strategy 1: Added layer updates



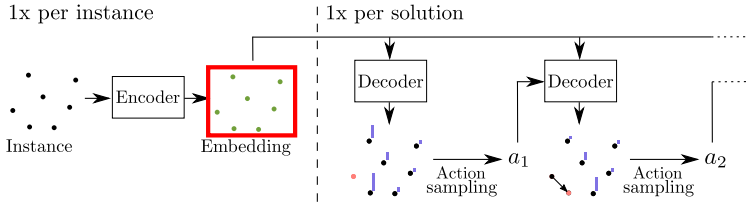
EAS-Lay:

1. Add **instance-specific** residual layers to the decoder
2. Update **only** these layers during the search

Improves **batch search performance**:

- Compute gradients only up to the new layer
- Most network weights are shared across instances
- Most network operations can be applied identically across instances

EAS Strategy 2: Embedding updates



EAS-Emb:

1. Update the instance embeddings using reinforcement learning

Improves **batch search performance**:

- All network weights are shared across instances
- All network operations can be applied identically across instances

EAS embedding updates: loss function

Notation modified from Hottung et al. 2022

Let $\hat{\omega}$ be a subset of the embeddings ω.

$$\nabla \mathcal{L}_{\mathcal{RL}}(\hat{\omega}) = \mathbb{E}_{\pi}\left[(L(\pi) - b(s))\nabla \log q_\theta(\pi|\hat{\omega})\right] \text{ where } q_\theta(\pi|\hat{\omega}) \equiv \prod_{t=1}^{T} q_\theta(a_t|s_t, \hat{\omega})$$

We further define an *imitation* loss for the best solution found so far:

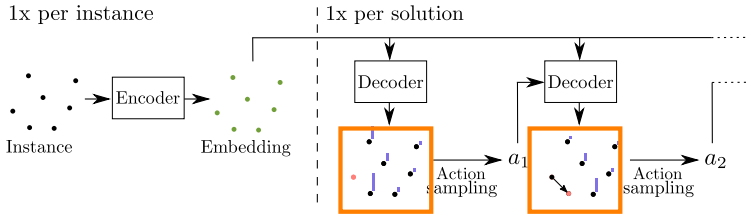
$$\nabla \mathcal{L}_{\mathcal{IL}}(\hat{\omega}) = -\nabla_{\hat{\omega}} \log q_\theta(\bar{\pi}|\hat{\omega} \equiv -\nabla_{\hat{\omega}} \log \prod_{t=1}^{T} q_\theta(\bar{a}_t|s_t, \hat{\omega})$$

$$\nabla \mathcal{L}_{\mathcal{RIL}}(\hat{\omega}) = \nabla \mathcal{L}_{\mathcal{RL}}(\hat{\omega}) + \lambda \nabla \mathcal{L}_{\mathcal{IL}}(\hat{\omega})$$

Notation:

- a_t, s_t – Action, state at time t
- $\bar{\pi}$ – Best solution found in current sample
- $\bar{a}_t$ – Action of best solution at time t
- λ – Adjustable parameter (RL vs. IL loss)

EAS Strategy 3: Tabular updates



EAS-Tab:

- Use a lookup table Q to modify the policy of a given model

$$p_\theta(a_t|s_t)^\alpha \cdot Q_{g(s_t,a_t)}$$

- The table Q is updated using a simple formula. No backpropagation is needed.

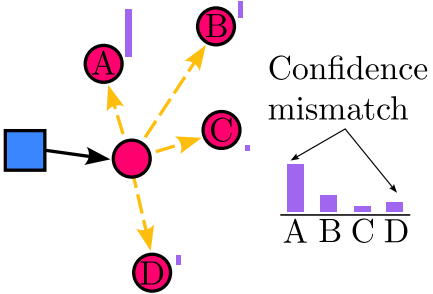
"Illuminating a tree in the night"

Simulation-guided Beam Search

Further enhancing search: simulation-guided beam search

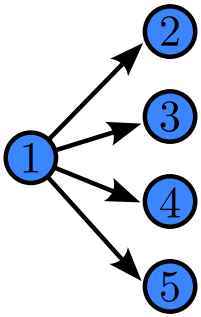
Joint work with Jinho Choo, Yeong-Dae Kwon Jihoon Kim, Jeongwoo Jae, André Hottung, and Youngjune Gwon (NeurIPS 2022)

- **Insight:** Models make stupid mistakes with high confidence (**overconfidence**)
- **Contribution:** Overcome mistakes with beam search with **simulations** to evaluate the quality of nodes



Source code: `https://github.com/yd-kwon/sgbs`

Beam search



What is beam search?

1. Width-limited breadth first.
2. Only investigate the best b nodes ("beam") in each layer

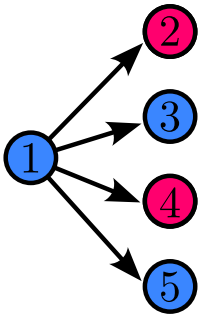
GPU advantage:

- Fixed width allows for easy batching

Weakness:

- Arguably better search strategies exist (least discrepancy, DFS,...)

Beam search



What is beam search?

1. Width-limited breadth first.
2. Only investigate the best b nodes ("beam") in each layer

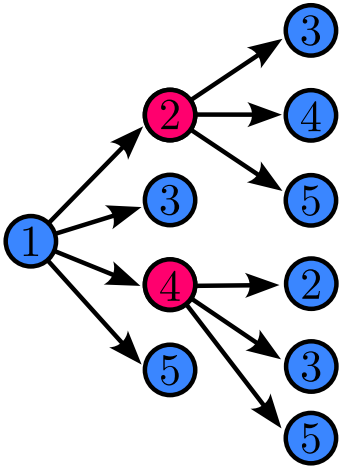
GPU advantage:

- Fixed width allows for easy batching

Weakness:

- Arguably better search strategies exist (least discrepancy, DFS,...)

Beam search



What is beam search?

1. Width-limited breadth first.
2. Only investigate the best b nodes ("beam") in each layer

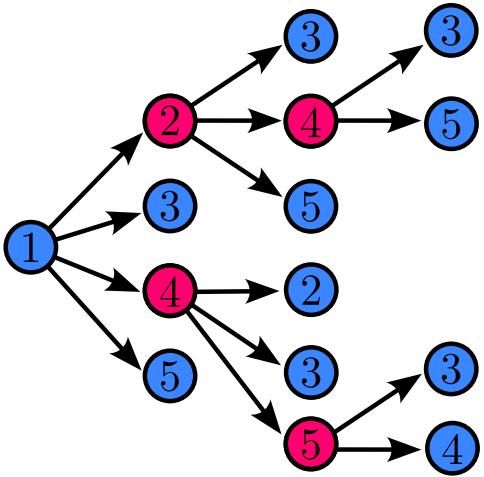
GPU advantage:

- Fixed width allows for easy batching

Weakness:

- Arguably better search strategies exist (least discrepancy, DFS,...)

Beam search



What is beam search?

1. Width-limited breadth first.
2. Only investigate the best b nodes ("beam") in each layer

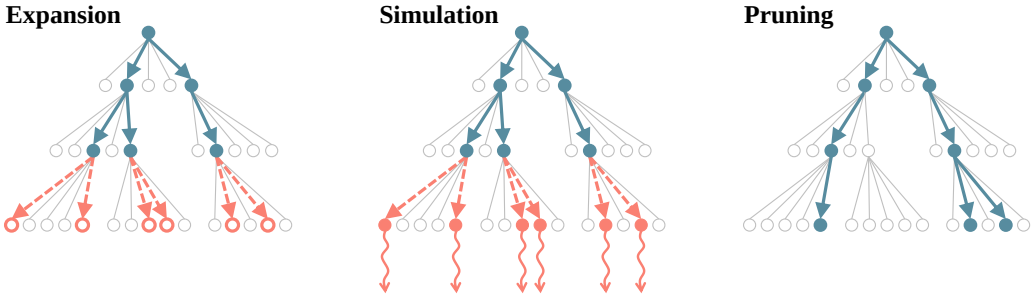
GPU advantage:

- Fixed width allows for easy batching

Weakness:

- Arguably better search strategies exist (least discrepancy, DFS,...)

SGBS: Three phases



1. **Expand** highest ranked nodes (by model) within the beam width
2. **Simulate** the candidate nodes (argmax rollout)
3. **Prune** down to the beam width
4. **Optional:** After solving, use EAS to improve model

SGBS/EAS experimental results on TSP

	Method	**Test** (10K instances) $n = 100$ Obj.	Gap	Time	**Generalization** (1K instances) $n = 150$ Obj.	Gap	Time	$n = 200$ Obj.	Gap	Time
TSP	Concorde	7.765	-	(82m)	9.346	-	(17m)	10.687	-	(31m)
	LKH3	7.765	0.000%	(8h)	9.346	0.000%	(99m)	10.687	0.000%	(3h)
	DACT	7.771	0.089%	(8h)						
	DPDP	7.765	0.004%	(2h)	9.434	0.937%	(44m)	11.154	4.370%	(74m)
	POMO greedy	7.776	0.144%	(1m)	9.397	0.544%	(<1m)	10.843	1.459%	(1m)
	sampling	7.771	0.078%	(3h)	9.378	0.335%	(1h)	10.838	1.417%	(3h)
	EAS	7.769	0.053%	(3h)	9.363	0.172%	(1h)	10.731	0.413%	(3h)
		7.768	0.044%	(15h)	9.358	0.127%	(10h)	10.719	0.302%	(30h)
	SGBS (10,10)	7.769	0.058%	(9m)	9.367	0.220%	(8m)	10.753	0.619%	(14m)
	SGBS+EAS	7.767	0.035%	(3h)	9.359	0.136%	(1h)	10.727	0.378%	(3h)
		7.766	0.024%	(15h)	9.354	0.085%	(10h)	10.708	0.196%	(30h)

Training $n = 100$ instances

Experimental results: EAS/SGBS on CVRP

	Method	**Test** (10K instances) $n = 100$ Obj.	Gap	Time	**Generalization** (1K instances) $n = 150$ Obj.	Gap	Time	$n = 200$ Obj.	Gap	Time
"Traditional OR"	HGS	15.563	-	(54h)	19.055	-	(9h)	21.766	-	(17h)
	LKH3	15.646	0.53%	(6d)	19.222	0.88%	(20h)	22.003	1.09%	(25h)
"Learned models"	DACT	15.747	1.18%	(22h)	19.594	2.83%	(16h)	23.297	7.03%	(18h)
	NLNS	15.994	2.77%	(1h)	19.962	4.76%	(12m)	23.021	5.76%	(24m)
	DPDP	15.627	0.41%	(23h)	19.312	1.35%	(5h)	22.263	2.28%	(9h)
	POMO greedy	15.763	1.29%	(2m)	19.636	3.05%	(1m)	22.896	5.19%	(1m)
	sampling	15.663	0.64%	(6h)	19.478	2.22%	(2h)	23.176	6.48%	(5h)
	EAS	15.618	0.35%	(6h)	19.205	0.79%	(2h)	22.023	1.18%	(5h)
		15.599	0.23%	(30h)	19.157	0.54%	(20h)	21.980	0.98%	(50h)
	SGBS (4,4)	15.659	0.62%	(10m)	19.426	1.95%	(4m)	22.567	3.68%	(9m)
	SGBS+EAS	15.594	0.20%	(6h)	19.168	0.60%	(2h)	21.988	1.02%	(5h)
		15.580	0.11%	(30h)	19.101	0.24%	(20h)	21.853	0.40%	(50h)

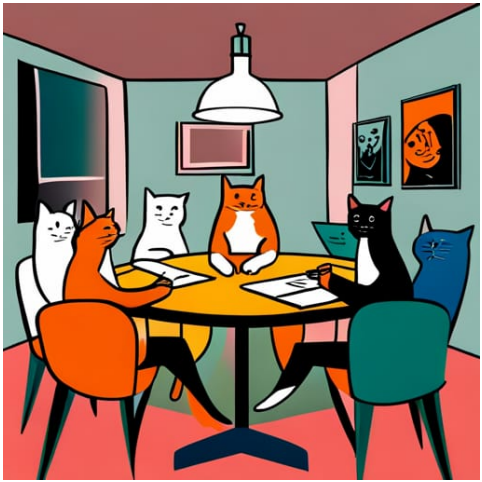
Training $n = 100$ instances Experimental

Experimental results: Flow shop scheduling

Method	FFSP20 Obj.	FFSP20 Gap	FFSP20 Time	FFSP50 Obj.	FFSP50 Gap	FFSP50 Time	FFSP100 Obj.	FFSP100 Gap	FFSP100 Time
CPLEX (60s)	46.37	22.07	(17h)	×			×		
CPLEX (600s)	36.56	12.26	(167h)						
Genetic Algorithm	30.57	6.27	(56h)	56.37	8.02	(128h)	98.69	10.46	(232h)
Particle Swarm Opt.	29.07	4.77	(104h)	55.11	6.76	(208h)	97.32	9.09	(384h)
MatNet greedy	25.38	1.08	(3m)	49.63	1.28	(8m)	89.70	1.47	(23m)
sampling	24.60	0.30	(10h)	48.78	0.43	(20h)	88.95	0.72	(40h)
EAS	24.60	0.30	(10h)	48.91	0.56	(20h)	88.94	0.71	(40h)
	24.44	0.14	(50h)	48.56	0.21	(100h)	88.57	0.34	(200h)
SGBS (5,6)	24.96	0.66	(12m)	49.13	0.78	(47m)	89.21	0.98	(3h)
SGBS+EAS	24.52	0.22	(10h)	48.60	0.25	(20h)	88.56	0.33	(40h)
	24.30	-	(50h)	48.35	-	(100h)	88.23	-	(200h)

Training: FFSP20 instances

EAS notebook & discussion



Notebooks and discussion

Discussion

Task: Let us train and solve some TSP instances.

Question: What is the algorithm doing? Try to visualize the network output.

A secret prompt

Advice and open challenges

Advice and controversial statements

Reviewer of a neural CO paper:

"Does this method scale to a CVRP with 5000 nodes?"

- How many companies are actually solving CVRPs this big with no decomposition whatsoever?
- Will the method still scale with side constraints? (hint: most don't...)

Scaling to absurdly large instances is not the only thing that matters in a method!

Advice and controversial statements

- Real-world instances are not uniformly generated.
- Algorithms need to be tuned/trained/adjusted for specific instance sets. (See: Schede et al. 2022)

Do not only test on uniformly generated instances!

Corollary: Try to use instances that are as real as possible!

Author of a neural CO paper:

"Uniformly generated instances are enough"

Advice and controversial statements

Author of a neural CO paper:

"I will compare my heuristic with OR-tools/CPLEX/Gurobi. Look, it is faster."

- Comparing heuristics with exact solvers is an apples to oranges comparison

Compare heuristics with heuristics! Use exact solvers to compute gaps to optimality!

Advice and controversial statements

- Embed your paper in the existing optimization literature
- Even if you submit to an ML conference!

You aren't the first one to try to solve optimization problem X.

Author of a DRL paper on CVRP:

"What's HGS?"

Advice and controversial statements

Reviewer of any paper:

"This work lacks technical sophistication."

- Keep it simple, stupid.
- Simple is good, don't demand complicated stuff!
- Don't punish authors for explaining their work well.

Don't be "that" reviewer. Reward simplicity.

Open challenges

General:

- In general, DRL rarely outperforms the SotA (e.g., HGS on CVRP) Non-Routing exception: See Hottung, Tanaka and Tierney (2020)!
- Feasibility not well-explored
- Side-constraints in general not well studied
- Trustworthiness (interpretability, explainability, etc.)
- Optimization under uncertainty

Search:

- Optimal tradeoff between "smart" and "fast" remains elusive
- Diversity: See, e.g., Grinsztajn et al. (2022)

Summary

- DRL can perform heuristic decision making to solve routing problems (and a whole lot more problems!)
- Algorithms can be **learned** for solving **specific datasets**
- Learned methods have almost reached the OR state of the art
  - And on container pre-marshalling, we beat traditional OR methods! See: `https://arxiv.org/abs/1709.09972` (Deep learning-assisted heuristic tree search)
- **Search** is an essential component of any learned technique, don't skip it!

Thanks for listening! Questions?

Our papers

On deep learning for CO

Deep-learning assisted heuristic tree search (DLTS) (C&OR 2020)

Neural Large Neighborhood Search (NLNS) (ECAI 2020 / AIJ 2022)

CVAE-Opt (ICLR 2020)

EAS (ICLR 2021)

SGBS (NeurIPS 2022)

AI4TSP (LION 2022)

Additional literature

- Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly. "Pointer networks." Advances in neural information processing systems 28 (2015).
- Kool, Wouter, Herke van Hoof, and Max Welling. "Attention, Learn to Solve Routing Problems!" ICLR 2019.
- Kwon, Yeong-Dae, et al. "POMO: Policy optimization with multiple optima for reinforcement learning." Advances in Neural Information Processing Systems 33 (2020).
- Hottung, André, and Kevin Tierney. "Neural Large Neighborhood Search for the Capacitated Vehicle Routing Problem." ECAI 2020.
- Hottung, André, Yeong-Dae Kwon, and Kevin Tierney. "Efficient Active Search for Combinatorial Optimization Problems." ICLR 2021.
- Choo et al. (2022). "Simulation-guided Beam Search for Neural Combinatorial Optimization." NeurIPS 2022.