Deploying

Neural Algorithmic Reasoning

Petar Veličković
Andreea Deac
Andrew Dudzik

Learning on Graphs Conference
10 December 2022

Intro

Hi!

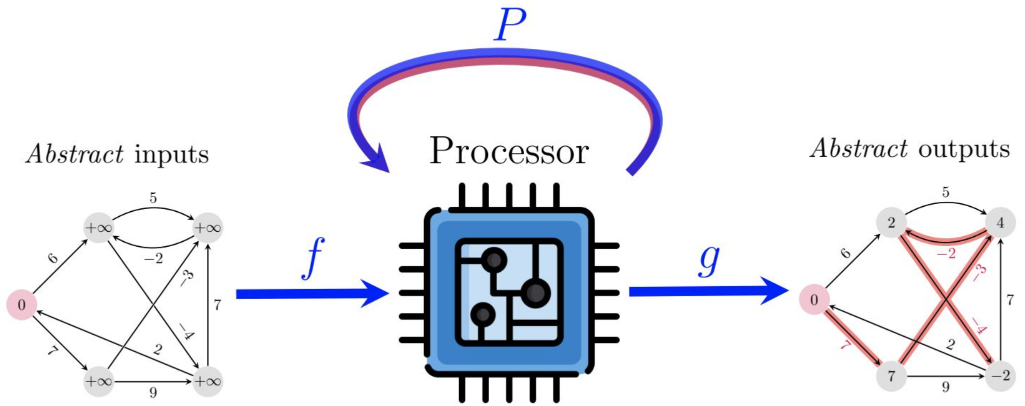
- I’m a fourth year PhD student at Mila/University of Montreal, in the group of Professor Jian Tang.

We will talk about how to use Neural Algorithmic Reasoners!

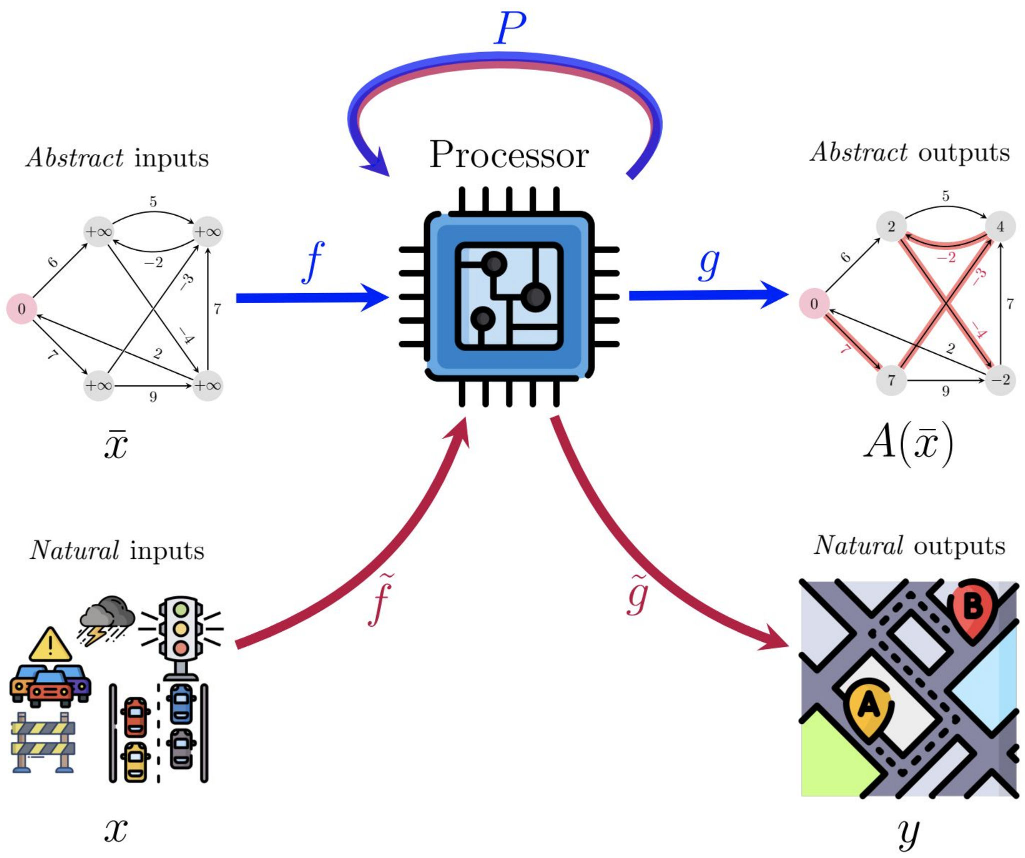
Starting from:

- Graph neural induction of value iteration
- Neural Algorithmic Reasoners are Implicit Planners

The pipeline

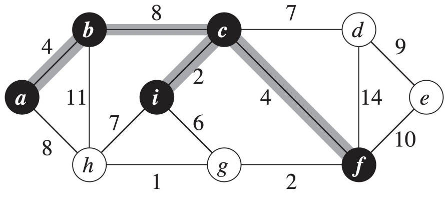


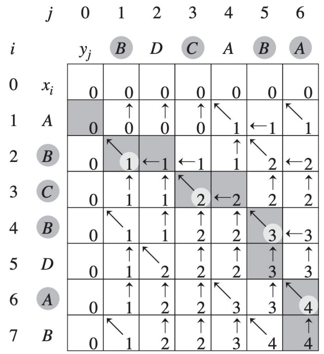
The pipeline

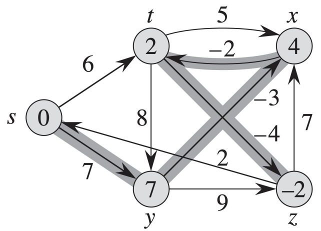


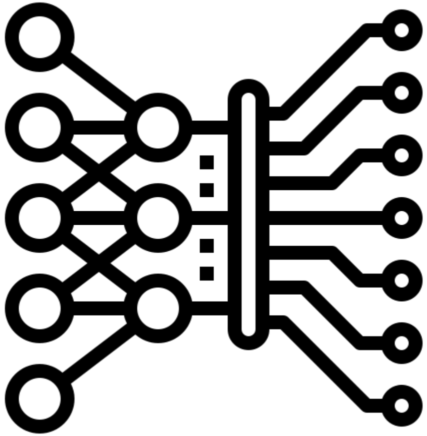
Problem-solving approaches

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```







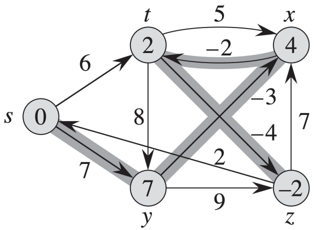


Algorithms

Neural networks

Algorithm figures: Cormen, Leiserson, Rivest and Stein. **Introduction to Algorithms.**

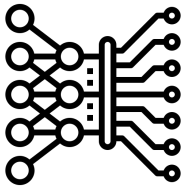
Problem-solving approaches



Algorithms

- \+ Trivially **strongly** generalise
- \+ **Compositional** (subroutines)
- \+ Guaranteed **correctness**
- \+ **Interpretable** operations
- \- Inputs must match **spec**
- \- Not **robust** to task variations

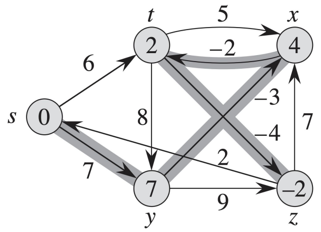
Problem-solving approaches

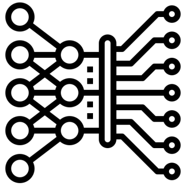


Neural networks

- \+ Operate on **raw** inputs
- \+ Generalise on **noisy** conditions
- \+ Models **reusable** across tasks
- \- Require **big data**
- \- Unreliable when **extrapolating**
- \- Lack of **interpretability**

Our problem-solving approach





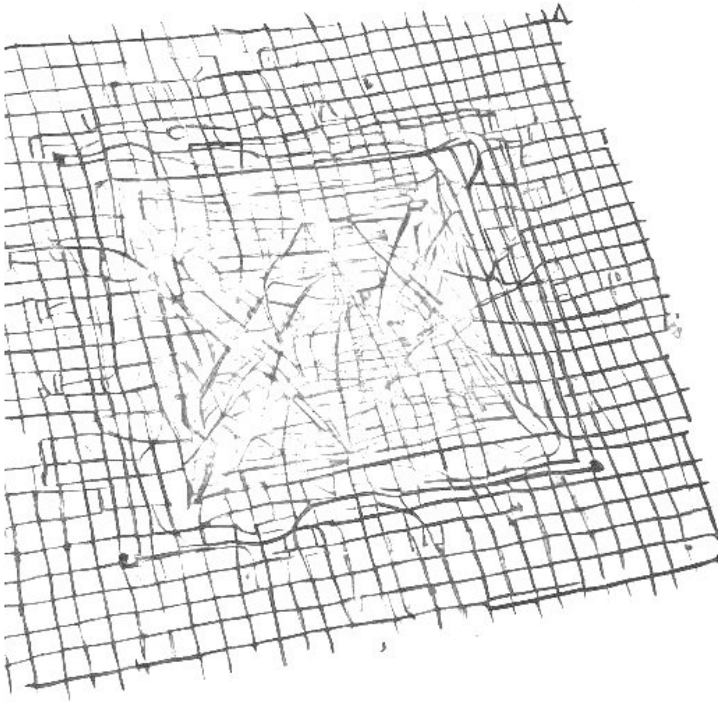
Algorithms

- \+ Trivially **strongly** generalise
- \+ **Compositional** (subroutines)
- \+ Guaranteed **correctness**
- \+ **Interpretable** operations
- \- Inputs must match **spec**
- \- Not **robust** to task variations

Neural networks

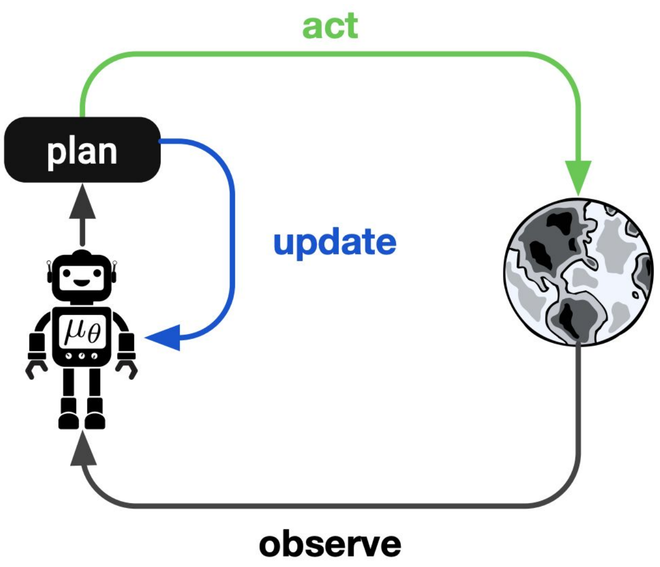
- \+ Operate on **raw** inputs
- \+ Generalise on **noisy** conditions
- \+ Models **reusable** across tasks
- \- Require **big data**
- \- Unreliable when **extrapolating**
- \- Lack of **interpretability**

Can we get the best of both worlds?

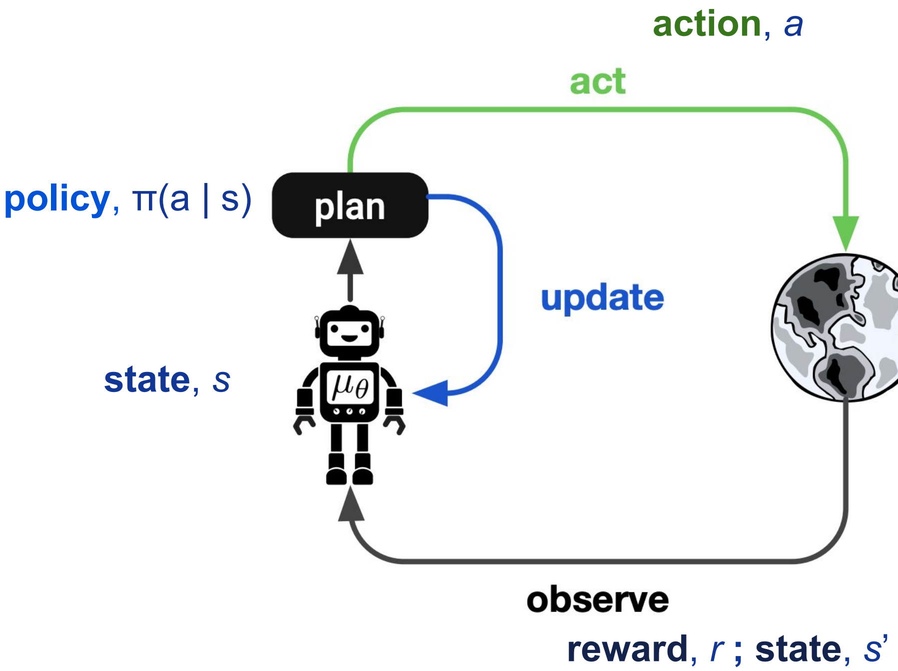


Our case study

Reinforcement learning setup



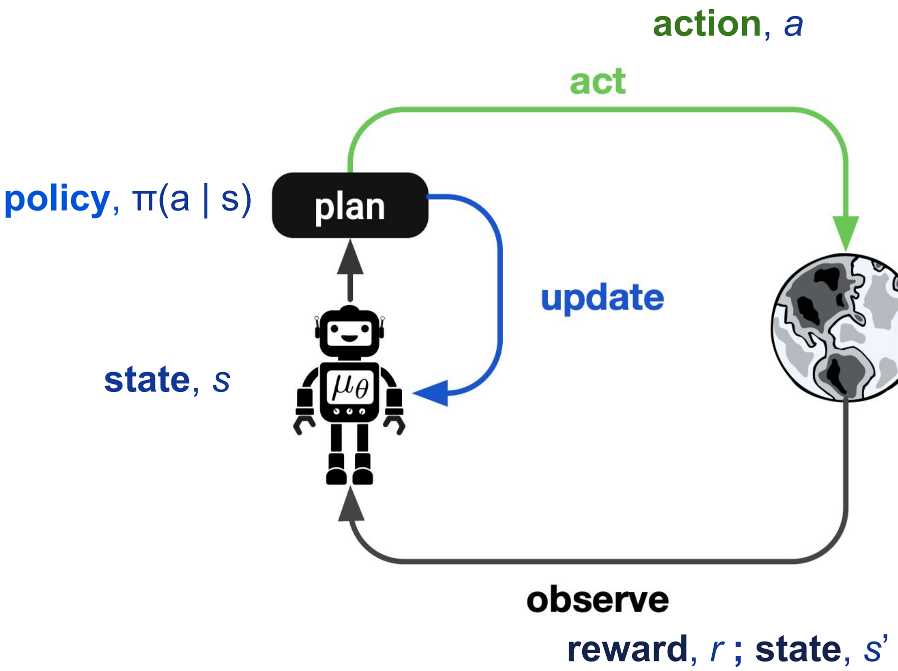
Reinforcement learning setup



transitions, $P(s' \mid s, a)$

rewards, $R(s, a)$

Reinforcement learning setup



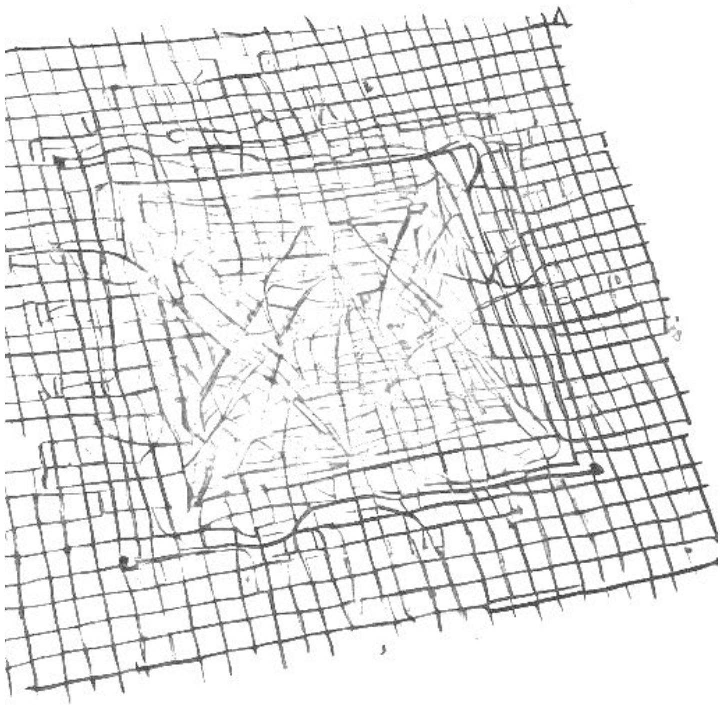
transitions, P(s' | s, a)

rewards, R(s, a)

Want to optimise:

Discounted cumulative reward

$G = \sum_{t \geq 0} \gamma^t r_t$



Code time!

Planning

Policies acting purely through adapting to observed rewards are often called **reactive.**

In many cases, they require large quantities of **data** and are slow to **adapt.**

Planning ameliorates such issues by maintaining an explicit **model** of the world:

- *State transition model*: $s' \sim f_T(s, a)$
- *Reward model*: $r \sim f_R(s, a)$
- Typically trained from **observed trajectories**

Using these models, a planner can **simulate** the effects of actions before taking them!

- Comes with many benefits if done properly...

Planning benefits

Gains in **data efficiency**: Good model implies fewer interactions are needed to learn to act

Strong models allow quickly **adapting** to previously unexplored situations

Being mindful of the consequences of acting enables better **safety**

Allowing to explicitly account for external factors (e.g. **human** interactions)

Impactful for **game-playing** (AlphaGo) and across the **sciences** (Segler *et al.*, Nature'18)

Encouraging theoretically: **perfect** models allow for planning **perfect** policies!

Algorithm to the rescue

Value Iteration: dynamic programming algorithm for perfectly solving an RL env.

$$v^{(t+1)}(s) = \max_{a \in \mathcal{A}_s} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^{(t)}(s')$$

where v(s) corresponds to the value of state s.

Guaranteed to converge to optimal solution (fixed-point of Bellman opt. equation)!

$$V^\star(s) = \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\star(s') \right)$$

Optimal policy takes actions that maximise expected value

Value iteration in *grid worlds*

$$v^{(t+1)}(s) = \max_{a \in \mathcal{A}_s} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^{(t)}(s')$$

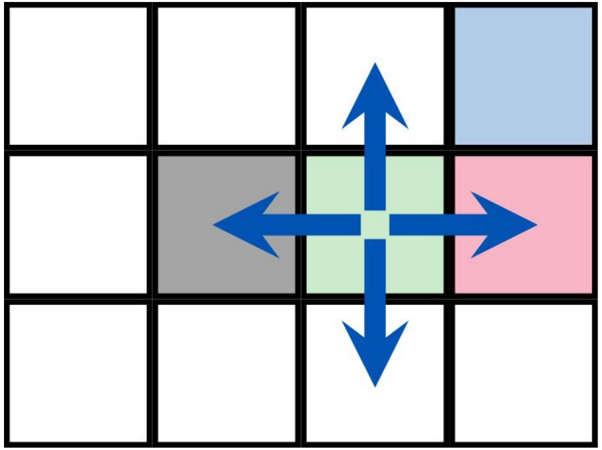
A simpler case: P is assumed **fixed** and **known**.

Each state has **known** neighbours

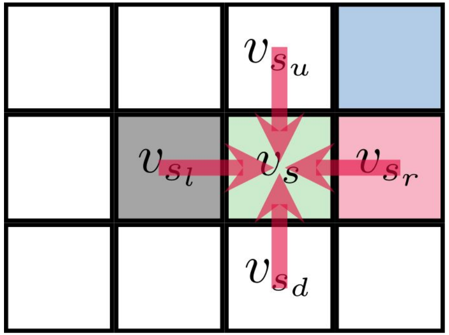
Actions are **deterministic**

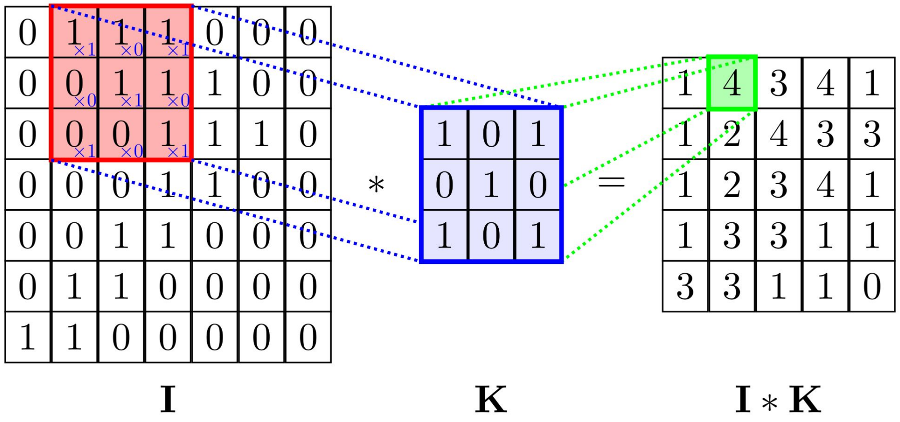
In this setting, VI amounts to…

- Computing **sum** of neighbouring values!



Grid-world VI ~ Convolution!





Value Iteration Networks

Exactly this idea is leveraged by *Value Iteration Networks* (Tamar *et al.*, NeurIPS'16)

Assuming the underlying MDP is **discrete**, **fixed** and **known**…

We can perform VI-style computation by stacking a **shared** convolutional layer

$\Longrightarrow$ We have our differentiable planning module!

Original VIN paper mainly dealt with grid worlds and hence used CNNs

- Extended to generic MDPs and GNNs by *GVINs* (Niu *et al.*, AAAI'18)

Moving beyond known world-models

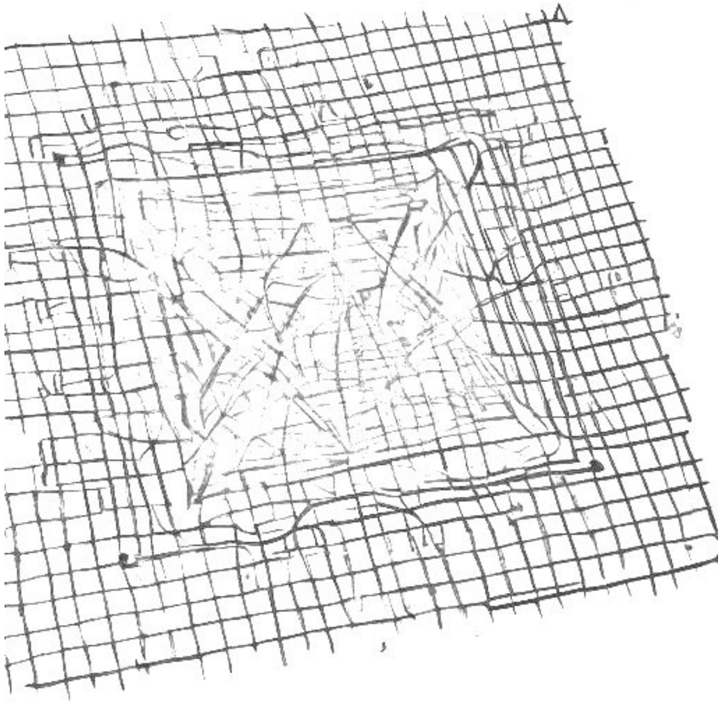
Assuming the MDP is **fixed** and **known** was quite helpful

- We never needed to estimate *transition models*
- Didn't have to deal with *continuous* state spaces

$$v^{(t+1)}(s) = \max_{a \in \mathcal{A}_s} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^{(t)}(s')$$

What about when we don't know the MDP?

While it could learn value iteration, the CNN could also learn anything else.



Bridging the gap between the algorithm and its application

How would a human engineer use VI?

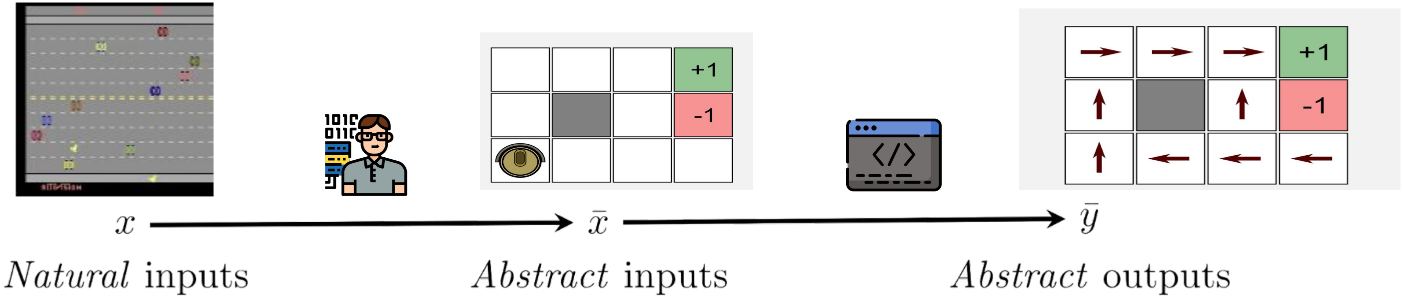
Assume we have encoded our state (e.g. with a NN) into **embeddings**, $z(s) \in \mathbb{R}^k$

To expand a *"local MDP"* we can apply VI over, we can use a *transition model*, T

- It is then of the form $T : \mathbb{R}^k \times A \rightarrow \mathbb{R}^k$
- Optimised such that $T(z(s), a) \approx z(s')$

Many popular methods exist for learning T in the context of *self-supervised learning*

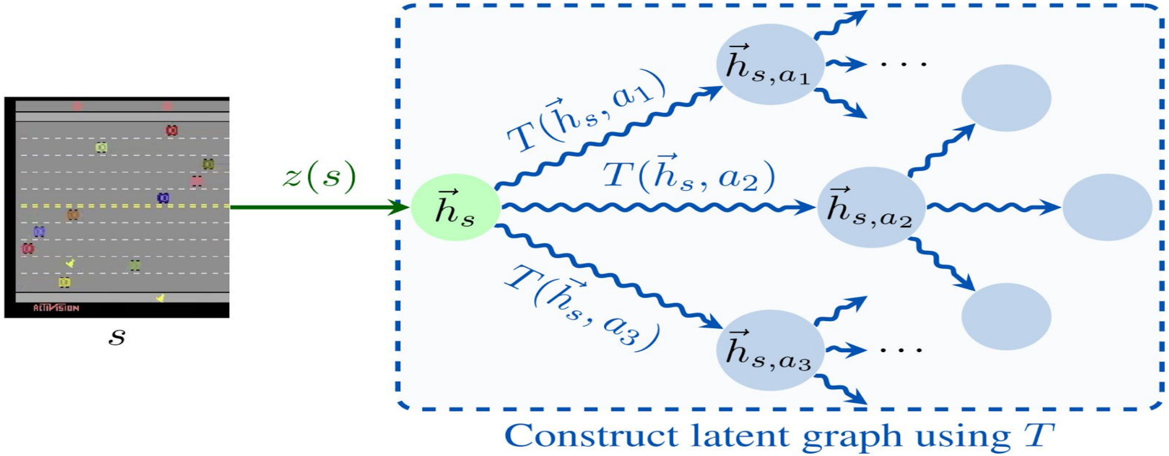
Contrastive learning: discriminate (s, a, s') from negative pairs (s, a, s~)



Using a transition model to *expand*

We can use a learned transition model on **every** action, to be exhaustive (~*breadth-first search*)

- Doesn't **scale** with large action spaces / thinking times; $O(|A|^K)$
- Can find more interesting search strategies



TreeQN/ATreeC

Assume that we have reward/value models, giving us scalar **values** in every expanded node

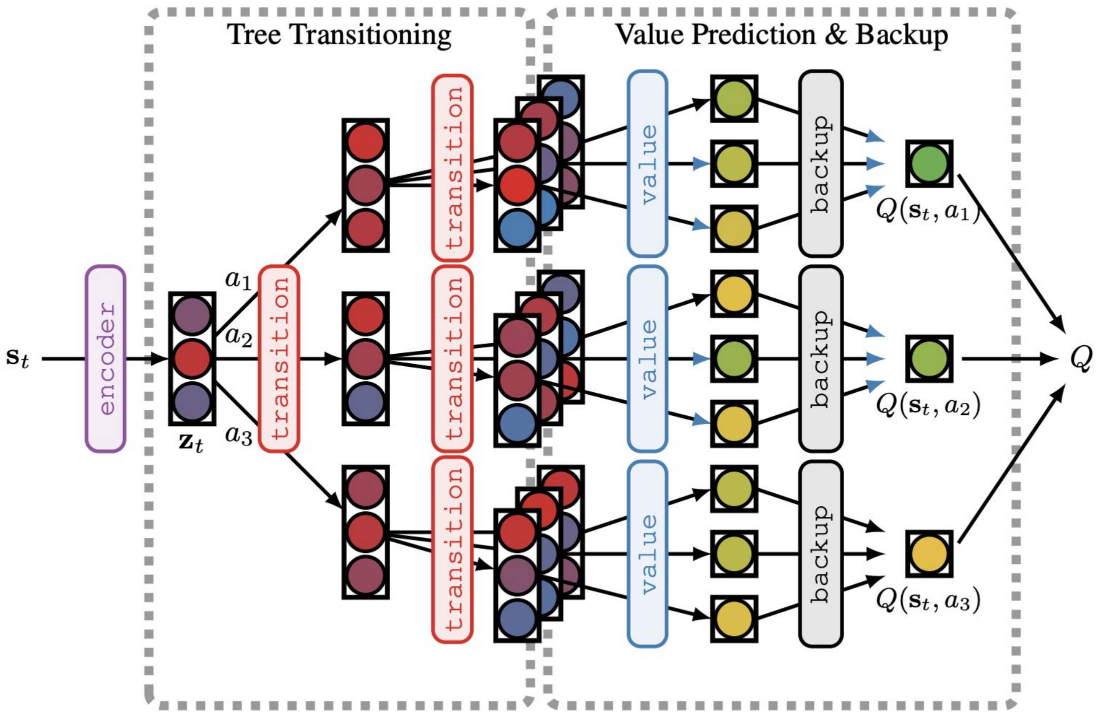
- We can now **directly** apply a VI-style update rule!

$$Q(\mathbf{z}_{l|t}, a_i) = r(\mathbf{z}_{l|t}, a_i) + \begin{cases} \gamma V(\mathbf{z}^{a_i}_{d|t}) & l = d-1 \\ \gamma \max_{a_j} Q(\mathbf{z}^{a_i}_{l+1|t}, a_j) & l < d-1 \end{cases}$$

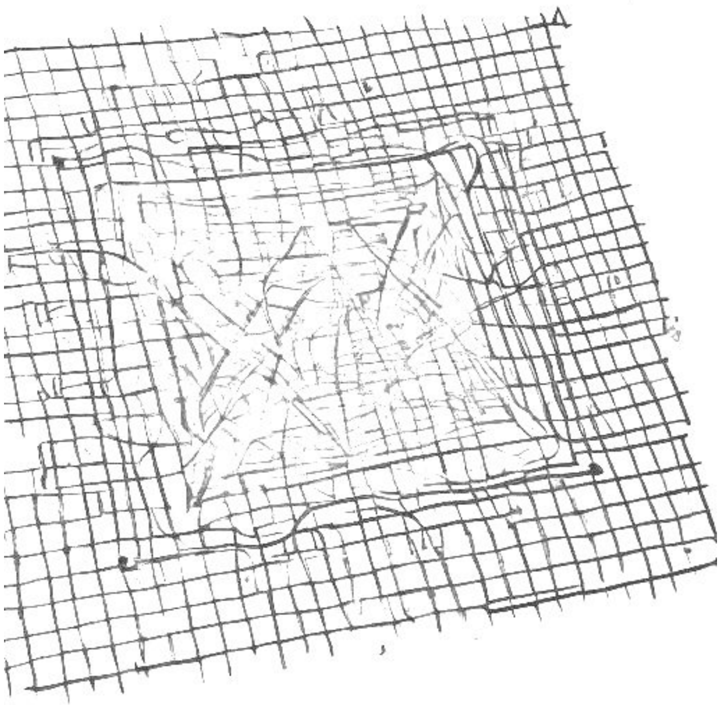
Can then use the computed Q-values **directly** to decide the policy

- Exactly as leveraged by models like TreeQN / ATreeC (Farquhar et al., ICLR'18)
    - Also related: Value Prediction Networks (Oh et al., NeurIPS'17)

TreeQN/ATreeC in action



$$Q(\mathbf{z}_{l|t}, a_i) = r(\mathbf{z}_{l|t}, a_i) + \begin{cases} \gamma V(\mathbf{z}^{a_i}_{d|t}) & l = d-1 \\ \gamma \max_{a_j} Q(\mathbf{z}^{a_i}_{l+1|t}, a_j) & l < d-1 \end{cases}$$



Recap

Recap

We mapped our natural inputs (e.g. pixels) to the space of abstract inputs

- (local MDP + reward values in every node)

This allowed us to execute VI-style algorithms directly on the abstract inputs

- The VI update is differentiable, and hence so is our entire implicit planner.

However...

Algorithmic bottleneck

Real-world data is often incredibly *rich*

We still have to compress it down to **scalar values**

The VI algorithmic solver:

- **Commits** to using this scalar
- Assumes it is **perfect**!

If there are insufficient training data to properly estimate the scalars, we hit ***data efficiency*** issues again!

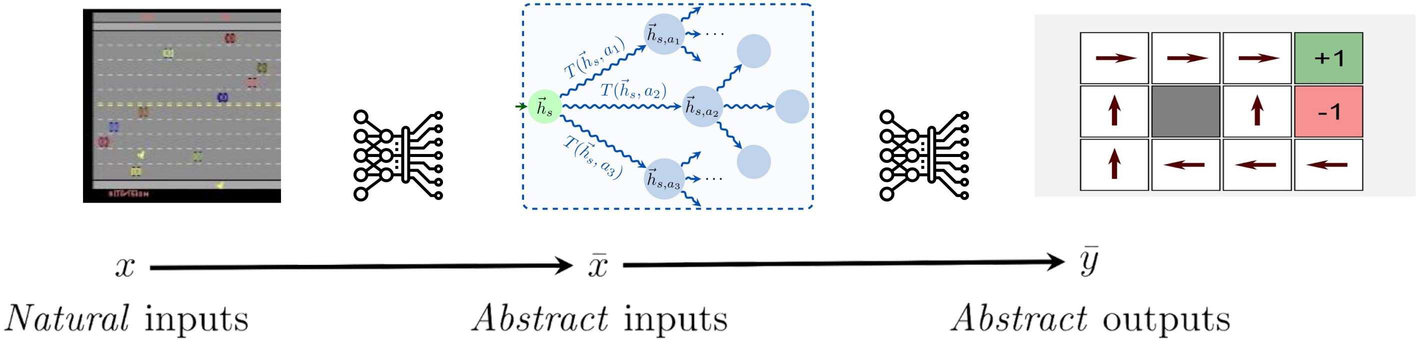
- Algorithm will give a **perfect** solution, but in a ***suboptimal*** environment

Breaking the bottleneck

Neural networks derive great flexibility from their **latent** representations

- They are inherently *high-dimensional*
- If any component is poorly predicted, others can step in and compensate!

To *break the bottleneck*, we replace the VI update with a **neural network**!



Breaking the bottleneck with GNNs

GNN over state representations aligns with VI, but may put **pressure** on the planner

- Same gradients used to *construct* correct graphs **and** make VI computations

To alleviate this issue, we **pre-train** the GNN to perform value iteration-style computations (over many **synthetic** MDPs), then deploying it within our planner

This exploits the concept of *algorithmic alignment* (Xu et al., ICLR’20) *[Deepening in Part III of the tutorial]*

$$v^{(t+1)}(s) = \max_{a\in\mathcal{A}_s} r(s,a) + \gamma \sum_{s'\in S} p(s'|s,a)\, v^{(t)}(s') .$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \qquad m_v^{t+1} = \sum_{w\in N(v)} M_t(h_v^t, h_w^t, e_{vw})$$

Synthetic data

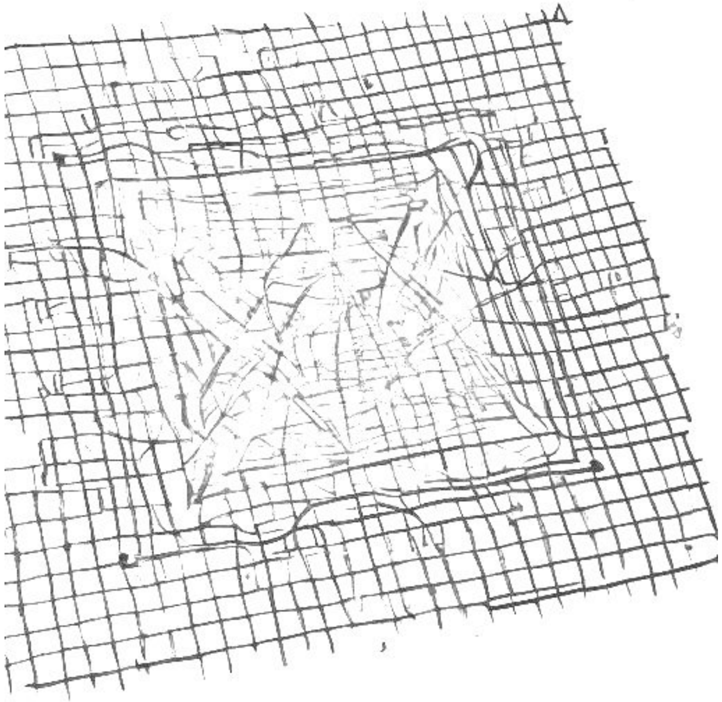
For each action, a graph (V, E) where each vertex represents a state.

- Node attributes: v(s), r(s,a)
- Edge attributes: p(s'|s,a), gamma (mask out 50% of edges at random).

Trained on random P, R for |S|=20 and |A|=5. Tested on |S|={20, 50, 100}, |A|={5, 10, 20}.

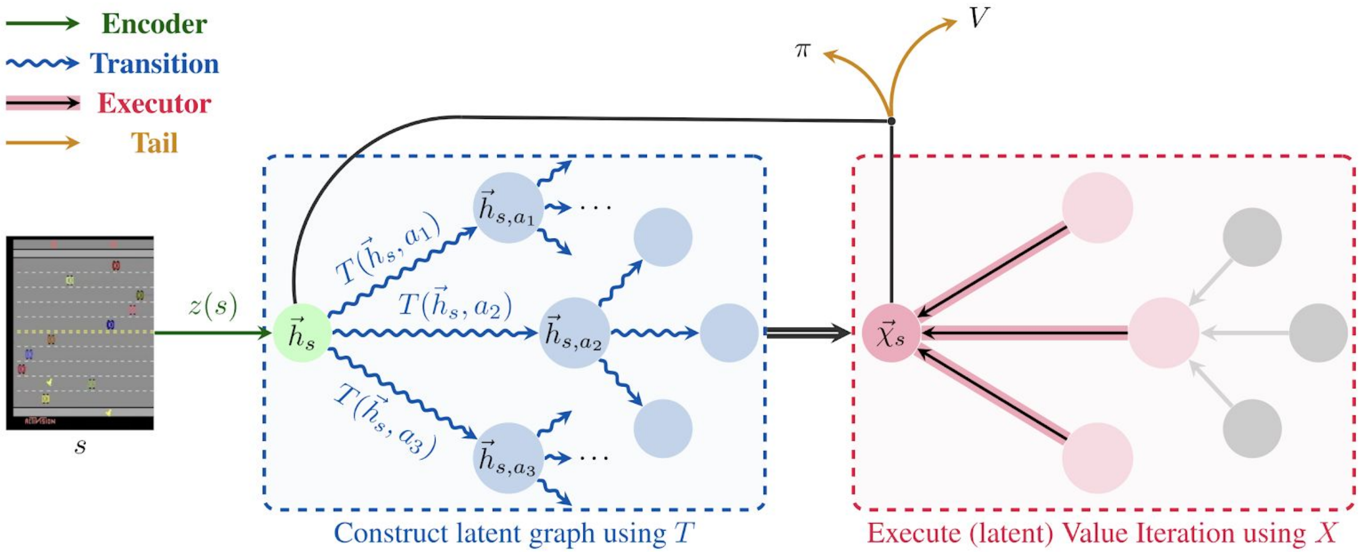
Evaluate **strong generalisation**!

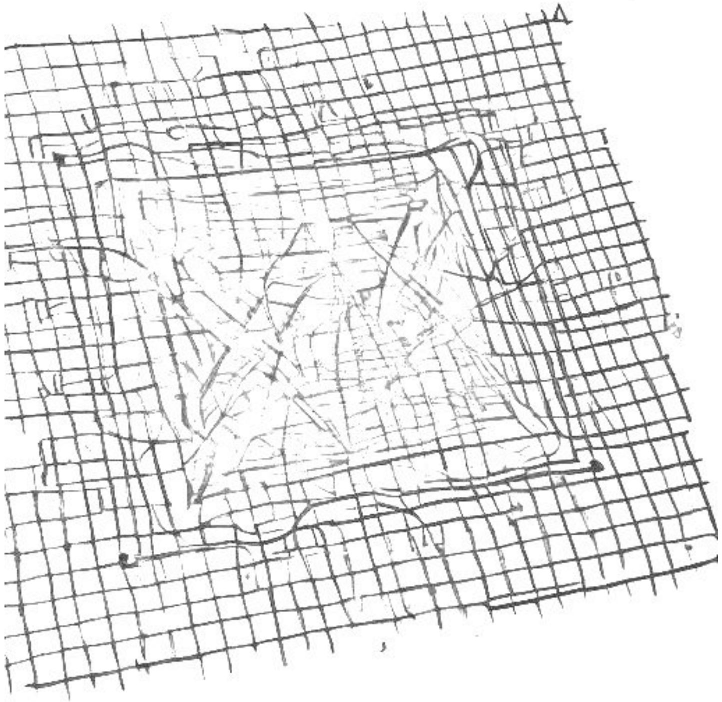
Optimise **MSE** of 1-step dynamics; rollout at test time



Code time!

NAR as Implicit Planner





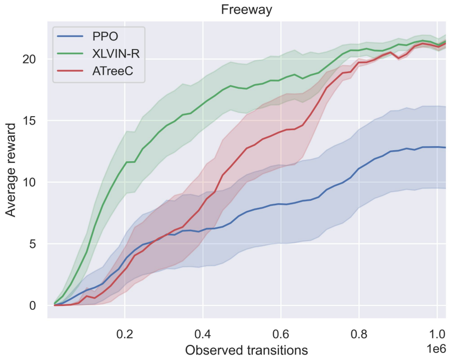
Results

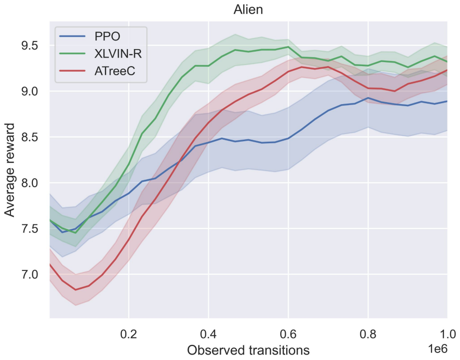
Results on low-data environments

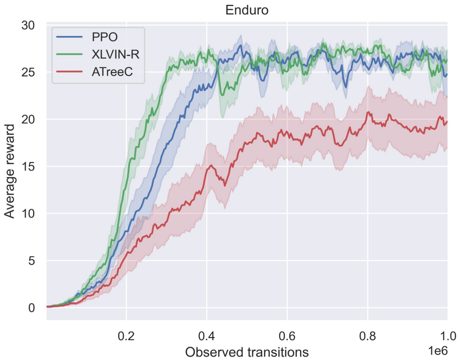
Table 1: Mean scores for low-data CartPole-v0, Acrobot-v1, MountainCar-v0 and LunarLander-v2, averaged over 100 episodes and five seeds.

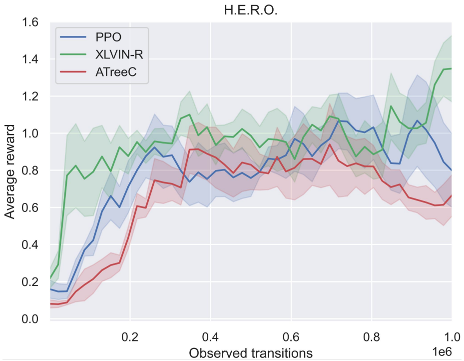
Agent	**CartPole-v0** 10 trajectories	**Acrobot-v1** 100 trajectories	**MountainCar-v0** 100 trajectories	**LunarLander-v2** 250 trajectories
PPO	104.6 ± 48.5	-500.0 ± 0.0	-200.0 ± 0.0	90.52 ± 9.54
ATreeC	117.1 ± 56.2	-500.0 ± 0.0	-200.0 ± 0.0	84.04 ± 5.35
XLVIN-R	**199.2** ± 1.6	-353.1 ± 120.3	-185.6 ± 8.1	**99.34** ± 6.77
XLVIN-CP	**195.2** ± 5.0	**-245.4** ± 48.4	**-168.9** ± 24.7	N/A

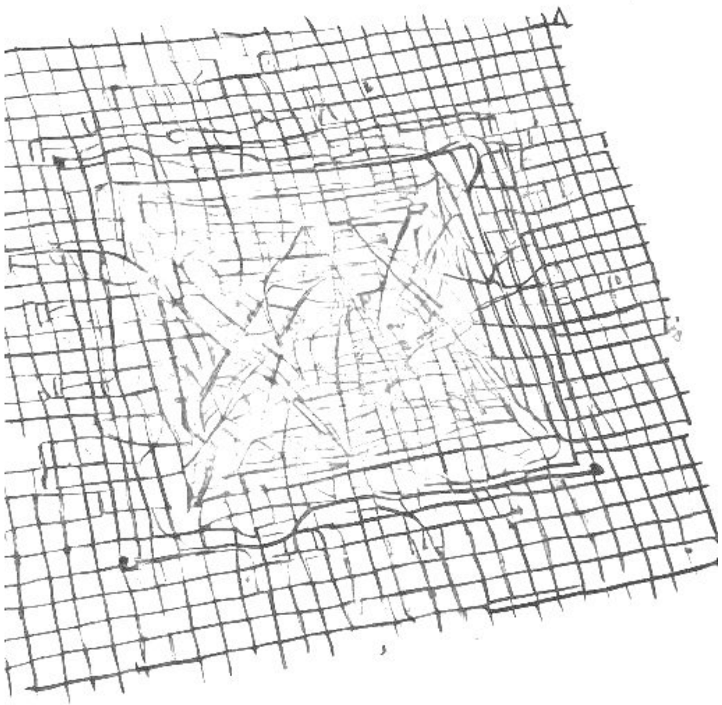
Results on low-data environments











Why did it work?

Studying the executor

Recall, our executor network was pre-trained and frozen

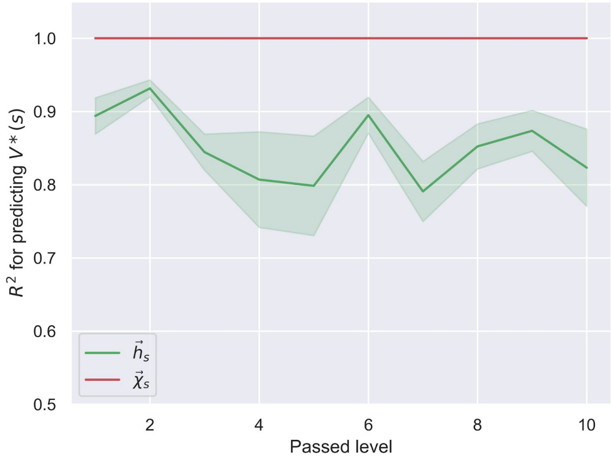
The encoder needed to learn to map **rich** inputs into the executor's latent space
- Analogous to human who tries to map real-world problems to algorithm inputs!

We evaluate the quality of the embeddings **before** and **after** applying the executor.

Here we can compute optimal V*(s)
- Evaluate linear decodability by linear regression!

Results verify our hypothesis!
- Input values are already predictive
- But the executor consistently **refines** them!

Our encoder learnt to correctly map the input to the latent algorithm!



Studying the algorithmic bottleneck

Algorithmic **bottleneck**: inaccuracies in scalar inputs to VI affect performance more than perturbations in high-dimensional state embeddings.

⇒ Algorithmic reasoner sacrifices perfect accuracy to achieve robustness to noise!

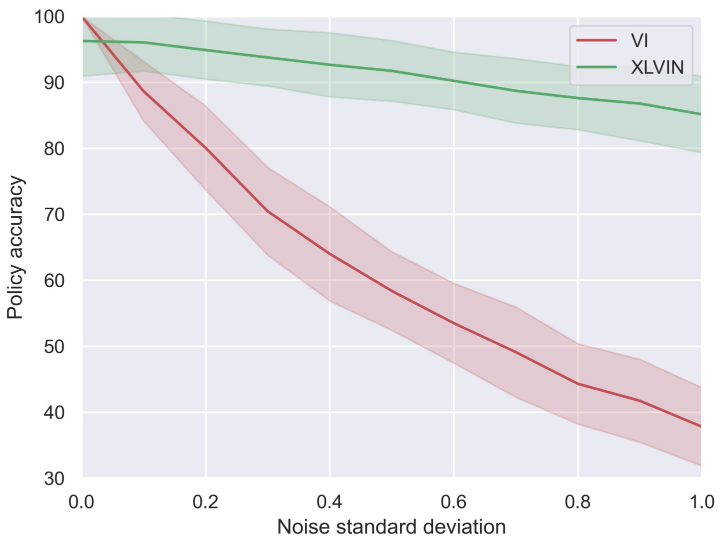
We introduce Gaussian noise
- to VI inputs
- to executor embeddings

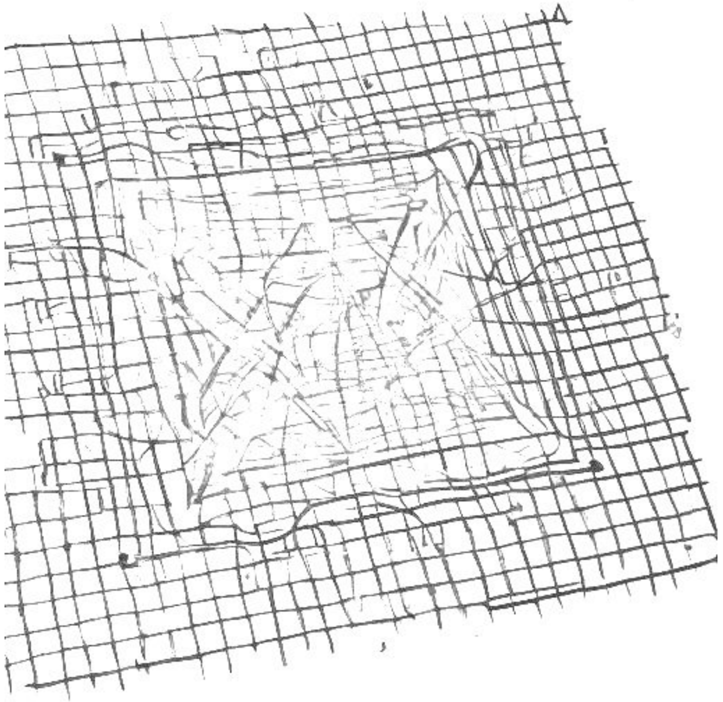
  and monitor policy accuracy

At zero noise, XLVIN is not optimal

But VI degrades much faster!

- Algorithm may give a perfect solution, but in a useless environment





Conclusions

XLVIN-specific conclusions

How to **formulate optimal plans** in a reinforcement learning setting?

- Value Iteration algorithm
- Requires full knowledge of the underlying MDP

How can we **apply** these optimal algorithms even **without privileged information**?

- Environment constraints → *Value Iteration Nets* [Tamar *et al.*, NeurIPS'16]
- Apply the algorithm directly → *Value Prediction Nets* [Oh *et al.*, NeurIPS'17]

Peculiar **bottleneck effects** with applying the algorithm

- Bottleneck implies more data is needed before efficient planning can emerge
- But the very point of planning is data efficiency!

We break the bottleneck using **XLVIN**

- Empirical gains on low-data Atari and classical control
- ATreeC requires more time to catch up

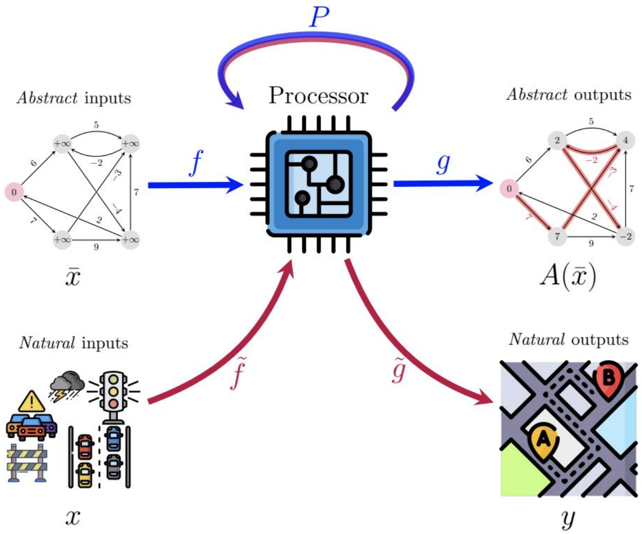
Why does it work?

- Demonstrating the algorithmic bottleneck and alignment to VI

Deploying-NAR conclusions

Real-world solutions can benefit from combining classical algorithms with neural networks.

Graph neural networks are well-suited to learn how to imitate dynamic programming algorithms (e.g. shortest path for navigation).



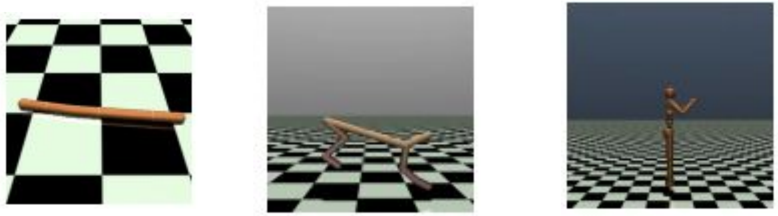
Deploying-NAR next steps

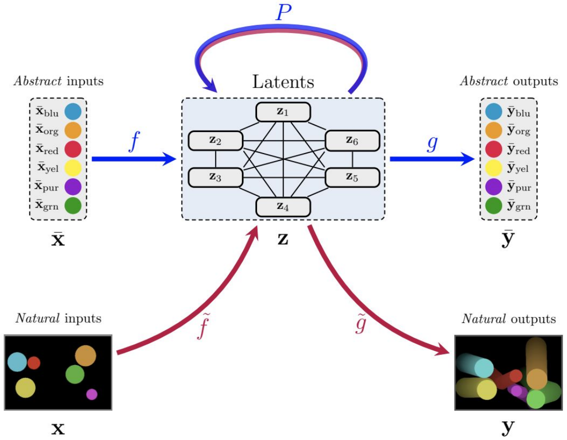
Continuous Neural Algorithmic Planners

(He et al, LoG 2022)

Reasoning-Modulated Representations

(Velickovic and Bosnjak et al, LoG 2022)





Also on the algorithmic side:

How to transfer algorithmic reasoning knowledge to learn new algorithms?

(Xhonneux et al, NeurIPS 2021)

A Generalist Neural Algorithmic Learner

(Ibarz et al, LoG 2022)

Deploying-NAR next steps

Continuous Neural Algorithmic Planners

(He et al, LoG 2022)

Reasoning-Modulated Representations

(Velickovic and Bosnjak et al, LoG 2022)

Also on the algorithmic side:

How to transfer algorithmic reasoning knowledge to learn new algorithms?

(Xhonneux et al, NeurIPS 2021)

A Generalist Neural Algorithmic Learner

(Ibarz et al, LoG 2022)

Especially, in the next part of the tutorial:

Deepening NAR

Thank you!

Questions?

andreeadeac22@gmail.com
https://andreeadeac22.github.io