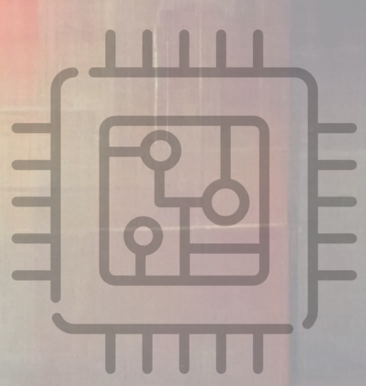
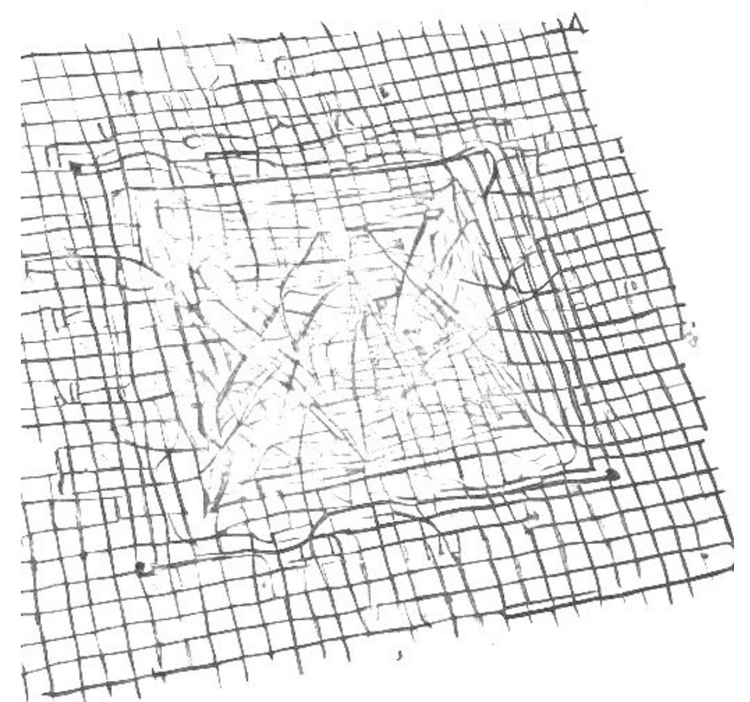
Deepening

Neural Algorithmic Reasoning



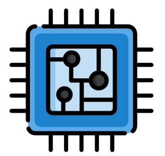
Petar Veličković
Andreea Deac
Andrew Dudzik

Learning on Graphs Conference
10 December 2022



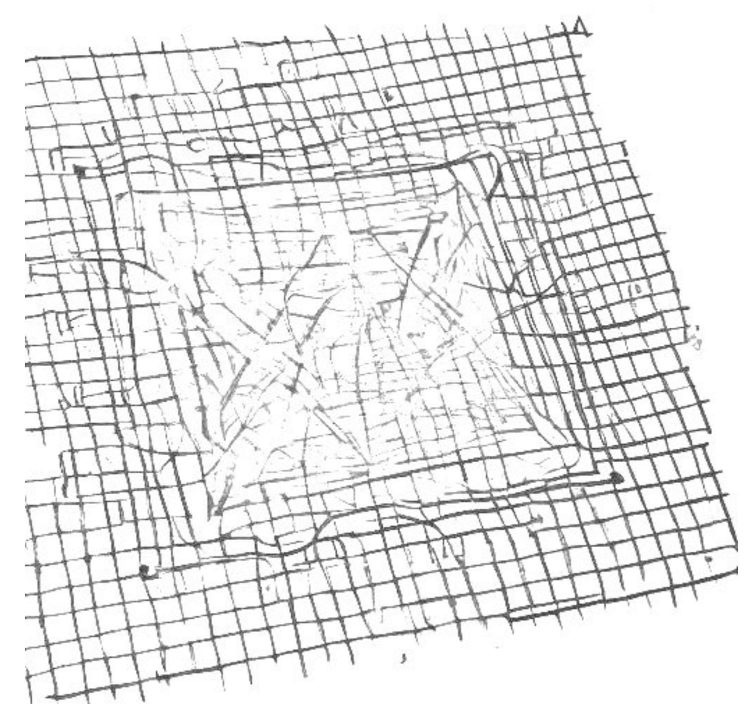
Overview

Overview



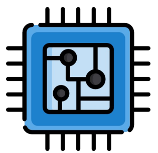
In this tutorial, I'll explain why we care, and what we know, about algorithmic alignment. The rough structure is:

- A brief description of alignment (Xu et al.'s definition and beyond)
- Convolutions, Polynomials, and Integral Transforms
- Code Examples
- Further Theory (monads!)



Alignment:
a prehistory

Alignment: a prehistory



The idea of “algorithmic alignment” was introduced in Xu et al. (ICLR 2020).

Roughly, the idea is that a **network** will generalise better on a **reasoning task** if the two share some structure.

In particular, **GNNs** and **Dynamic Programming** are a natural match.

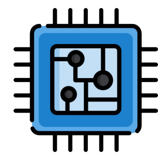
WHAT CAN NEURAL NETWORKS REASON ABOUT?

Keyulu Xu[†], Jingling Li[‡], Mozhi Zhang[‡], Simon S. Du[§], Ken-ichi Kawarabayashi[¶], Stefanie Jegelka[†]
[†]Massachusetts Institute of Technology (MIT)
[‡]University of Maryland
[§]Institute for Advanced Study (IAS)
[¶]National Institute of Informatics (NII)
{keyulu, stefje}@mit.edu

ABSTRACT

Neural networks have succeeded in many reasoning tasks. Empirically, these tasks require specialized network structures, e.g., Graph Neural Networks (GNNs) perform well on many such tasks, but less structured networks fail. Theoretically, there is limited understanding of why and when a network structure generalizes better than others, although they have equal expressive power. In this paper, we develop a framework to characterize which reasoning tasks a network can learn well, by studying how well its computation structure aligns with the algorithmic structure of the relevant reasoning process. We formally define this algorithmic alignment and derive a sample complexity bound that decreases with better alignment. This framework offers an explanation for the empirical success of popular reasoning models, and suggests their limitations. As an example, we unify seemingly different reasoning tasks, such as intuitive physics, visual question answering, and shortest paths, via the lens of a powerful algorithmic paradigm, dynamic programming (DP). We show that GNNs align with DP and thus are expected to solve these tasks. On several reasoning tasks, our theory is supported by empirical results.

Alignment: a prehistory



Definition 3.3. (PAC learning and sample complexity). Fix an error parameter $\epsilon > 0$ and failure probability $\delta \in (0, 1)$. Suppose $\{x_i, y_i\}_{i=1}^M$ are i.i.d. samples from some distribution $\mathcal{D}$, and the data satisfies $y_i = g(x_i)$ for some underlying function g. Let $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$ be the function generated by a learning algorithm $\mathcal{A}$. Then g is (M, ϵ, δ)-*learnable* with $\mathcal{A}$ if

$$\mathbb{P}_{x \sim \mathcal{D}} \left[\| f(x) - g(x) \| \le \epsilon \right] \ge 1 - \delta. \tag{3.1}$$

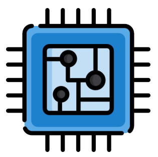
The *sample complexity* $\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta)$ is the minimum M so that g is (M, ϵ, δ)-learnable with $\mathcal{A}$.

With the PAC learning framework, we define a numeric measure of algorithmic alignment (Definition 3.4), and under simplifying assumptions, we show that the sample complexity decreases with better algorithmic alignment (Theorem 3.6).

Formally, a neural network aligns with an algorithm if it can simulate the algorithm via a limited number of modules, and each module is simple, i.e., has low sample complexity.

Definition 3.4. (Algorithmic alignment). Let g be a reasoning function and $\mathcal{N}$ a neural network with n modules $\mathcal{N}_i$. The module functions $f_1, ..., f_n$ generate g for $\mathcal{N}$ if, by replacing $\mathcal{N}_i$ with f_i, the network $\mathcal{N}$ simulates g. Then $\mathcal{N}$ (M, ϵ, δ)-algorithmically aligns with g if (1) $f_1, ..., f_n$ generate g and (2) there are learning algorithms $\mathcal{A}_i$ for the $\mathcal{N}_i$'s such that $n \cdot \max_i \mathcal{C}_{\mathcal{A}_i}(f_i, \epsilon, \delta) \le M$.

Alignment: a prehistory



In other words, a network is aligned to a "reasoning function" if it can be decomposed into modules, each of which can easily learn a corresponding module in the function.

This idea has been hugely influential—just over 2.5 years later, the paper has 165 citations.

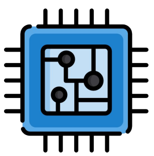
WHAT CAN NEURAL NETWORKS REASON ABOUT?

Keyulu Xu[†], Jingling Li[‡], Mozhi Zhang[‡], Simon S. Du[§], Ken-ichi Kawarabayashi[¶], Stefanie Jegelka[†]
[†]Massachusetts Institute of Technology (MIT)
[‡]University of Maryland
[§]Institute for Advanced Study (IAS)
[¶]National Institute of Informatics (NII)
{keyulu, stefje}@mit.edu

ABSTRACT

Neural networks have succeeded in many reasoning tasks. Empirically, these tasks require specialized network structures, e.g., Graph Neural Networks (GNNs) perform well on many such tasks, but less structured networks fail. Theoretically, there is limited understanding of why and when a network structure generalizes better than others, although they have equal expressive power. In this paper, we develop a framework to characterize which reasoning tasks a network can learn well, by studying how well its computation structure aligns with the algorithmic structure of the relevant reasoning process. We formally define this algorithmic alignment and derive a sample complexity bound that decreases with better alignment. This framework offers an explanation for the empirical success of popular reasoning models, and suggests their limitations. As an example, we unify seemingly different reasoning tasks, such as intuitive physics, visual question answering, and shortest paths, via the lens of a powerful algorithmic paradigm, dynamic programming (DP). We show that GNNs align with DP and thus are expected to solve these tasks. On several reasoning tasks, our theory is supported by empirical results.

Alignment: a prehistory



However, their definition of DP leaves open quite a lot of interpretation of what alignment really means:

- How can we align update functions?
- What if alignment requires sparse updates?
- What if we have to update non-differentiable inputs, like pointers?
- Is it possible for a network to align to more than one algorithm?

Definition 3.4. **(Algorithmic alignment).** Let g be a reasoning function and $\mathcal{N}$ a neural network with n modules $\mathcal{N}_i$. The module functions $f_1, ..., f_n$ generate g for $\mathcal{N}$ if, by replacing $\mathcal{N}_i$ with f_i, the network $\mathcal{N}$ simulates g. Then $\mathcal{N}$ (M, ϵ, δ)-algorithmically aligns with g if (1) $f_1, ..., f_n$ generate g and (2) there are learning algorithms $\mathcal{A}_i$ for the $\mathcal{N}_i$'s such that $n \cdot \max_i C_{\mathcal{A}_i}(f_i, \epsilon, \delta) \leq M$.

4.3 DYNAMIC PROGRAMMING

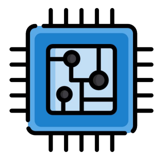
We observe that a broad class of relational reasoning tasks can be unified by the powerful algorithmic paradigm *dynamic programming (DP)* (Bellman, 1966). DP recursively breaks down a problem into simpler sub-problems. It has the following general form:

$$\text{Answer}[k][i] = \text{DP-Update}(\{\text{Answer}[k-1][j]\}, j = 1...n), \quad (4.1)$$

where $\text{Answer}[k][i]$ is the solution to the sub-problem indexed by iteration k and state i, and DP-Update is an task-specific update function that computes $\text{Answer}[k][i]$ from $\text{Answer}[k-1][j]$'s.

GNNs algorithmically align with a class of DP algorithms. We can interpret GNN as a DP algorithm, where node representations $h_i^{(k)}$ are $\text{Answer}[k][i]$, and the GNN aggregation step is the DP-Update. Therefore, Theorem 3.6 suggests that a GNN with enough iterations can sample efficiently learn any DP algorithm with a simple DP-update function, e.g. sum/min/max.

Alignment: a prehistory

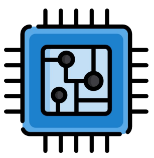


Even for the simple case of Bellman-Ford, the alignment of GNNs to DP isn't as simple as substituting an update function—there are three different modules!

$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v\in\mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right)$$

$$d_u \quad \leftarrow \quad \min\left(d_u, \min_{v\in\mathcal{N}_u} d_v + w_{v\to u}\right)$$

Alignment: a prehistory



To try to explain these three components of DP, we wrote a very strange paper. It uses the idea of a *transform* to formalize a broad class of fixed-memory DP algorithms, as well as message passing in GNNs.

This is not the complete picture! But it's all we know, and it led to some new architectures, so let me tell you about it.

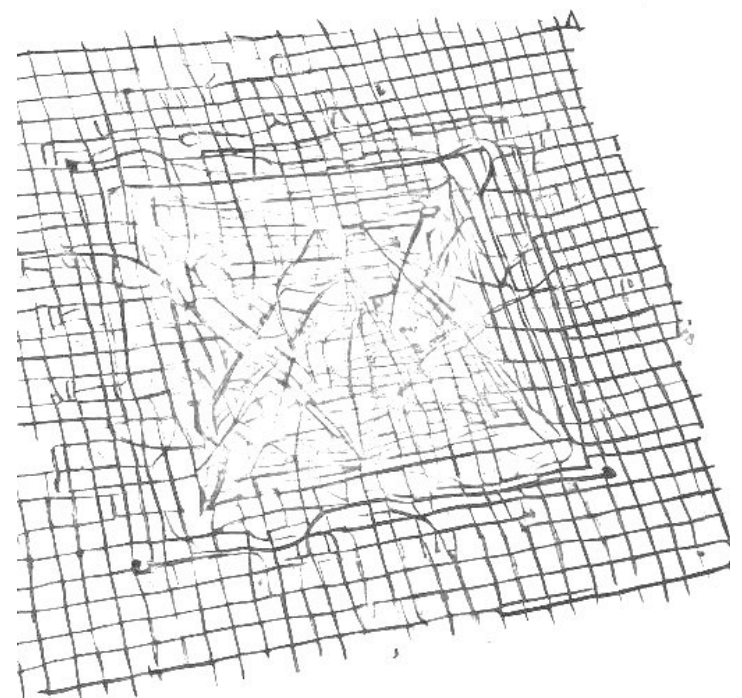
Graph Neural Networks are Dynamic Programmers

Andrew Dudzik*
DeepMind
adudzik@deepmind.com

Petar Veličković*
DeepMind
petarv@deepmind.com

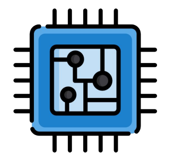
Abstract

Recent advances in neural algorithmic reasoning with graph neural networks (GNNs) are propped up by the notion of algorithmic alignment. Broadly, a neural network will be better at learning to execute a reasoning task (in terms of sample complexity) if its individual components align well with the target algorithm. Specifically, GNNs are claimed to align with dynamic programming (DP), a general problem-solving strategy which expresses many polynomial-time algorithms. However, has this alignment truly been demonstrated and theoretically quantified? Here we show, using methods from category theory and abstract algebra, that there exists an intricate connection between GNNs and DP, going well beyond the initial observations over individual algorithms such as Bellman-Ford. Exposing this connection, we easily verify several prior findings in the literature, produce better-grounded GNN architectures for edge-centric tasks, and demonstrate empirical results on the CLRS algorithmic reasoning benchmark. We hope our exposition will serve as a foundation for building stronger algorithmically aligned GNNs.



What is a Convolution?

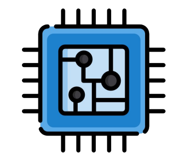
What is a convolution?



This is a convolution:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

What is a convolution?



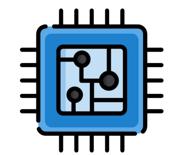
This is a convolution:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$$

And this is a convolution:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)\,dy,$$

What is a convolution?



But *transforms* are also convolutions:

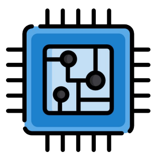
$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$$

$$c_n = \frac{1}{P} \int_P f(x)\, e^{-i2\pi \frac{n}{P} x}\, dx$$

$$s_n = \sum_{k=0}^{n} (-1)^k \binom{n}{k} a_k$$

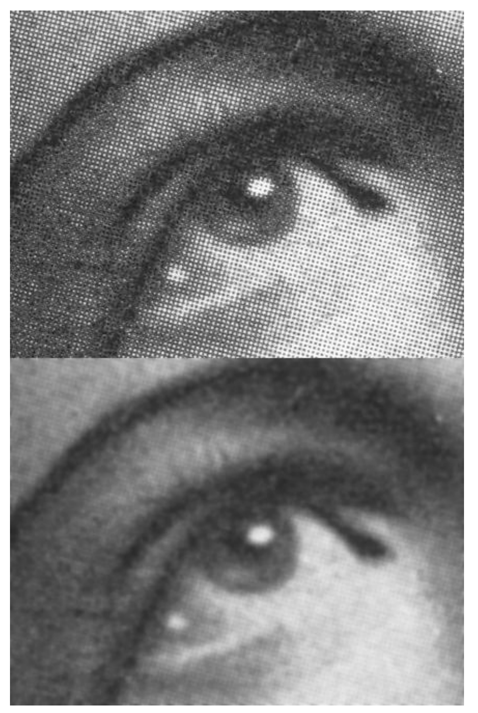
$$\mathcal{L}\{f(t)\}(s) = \int_0^\infty f(t) e^{-st}\, dt$$

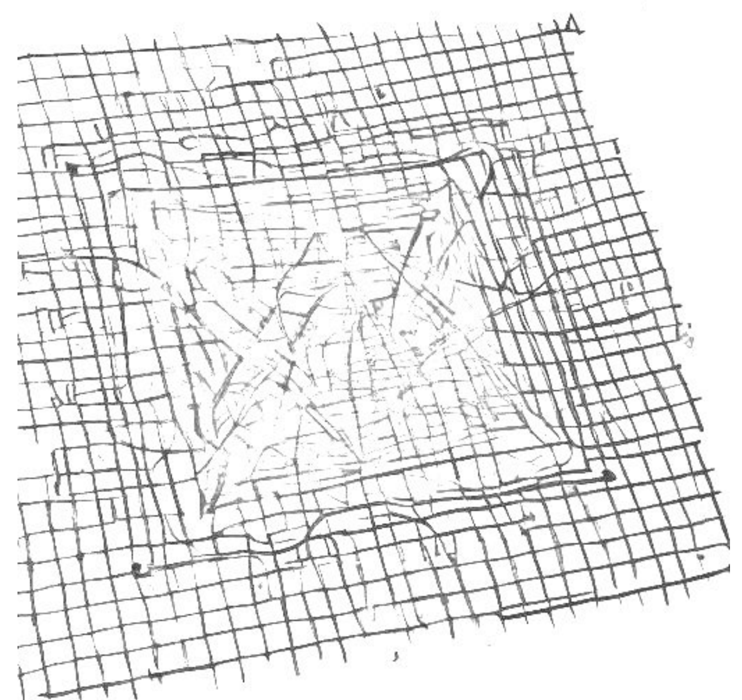
What is a convolution?



So what do all of these have in common?

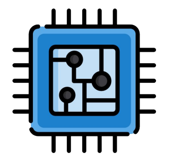
The answer should be an algebraic structure that is flexible enough to not only describe diverse phenomena, but rigid enough to support rigorous, provable connections between them.



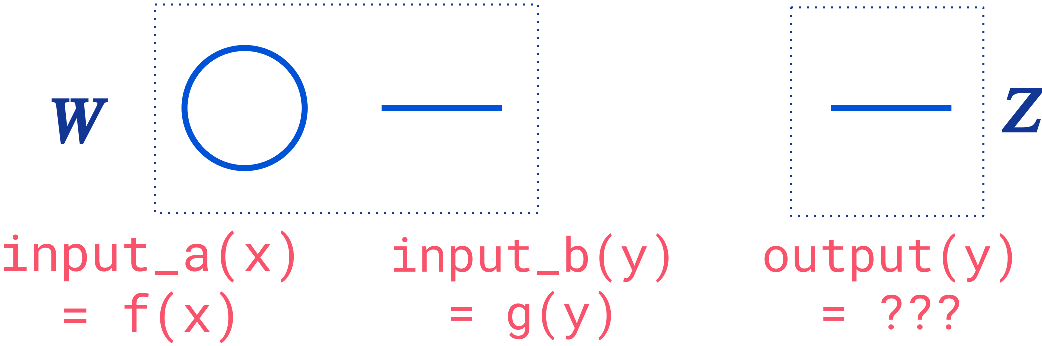


Integral Transforms

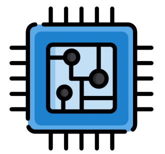
The integral transform, geometrically



Let's imagine we want to "send a message" from a circle and a line (a set W), onto a (potentially different) line (a set Z).

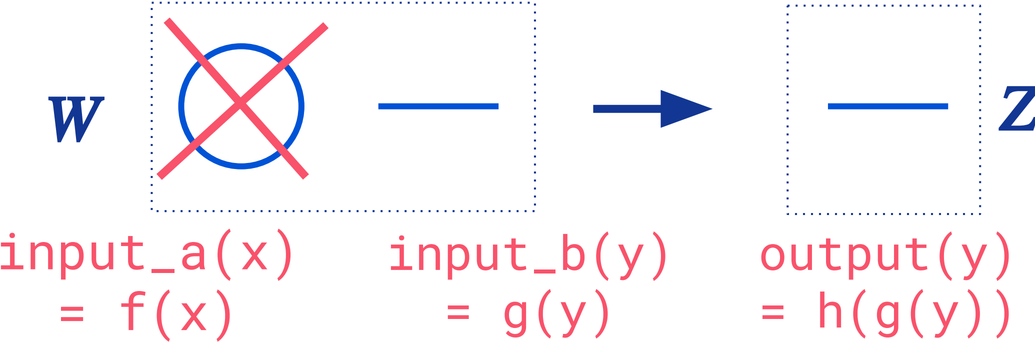


The integral transform, geometrically

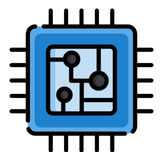


Let's imagine we want to "send a message" from a circle and a line (a set W), onto a (potentially different) line (a set Z).

We could just discard the circle data, but this is clearly suboptimal.
We need a *"carrier"* object, on which we can carry a *message* from W to Z
Any suggestions?

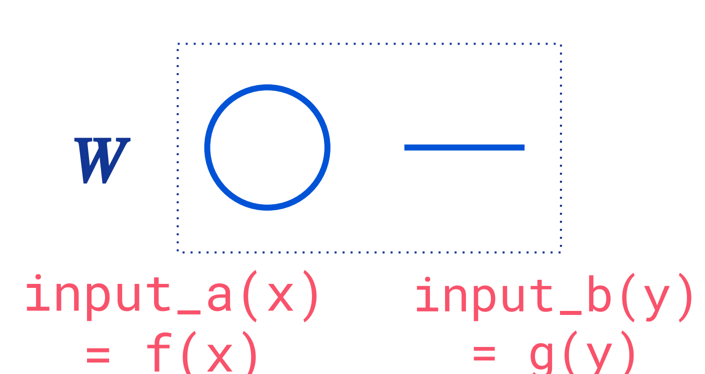


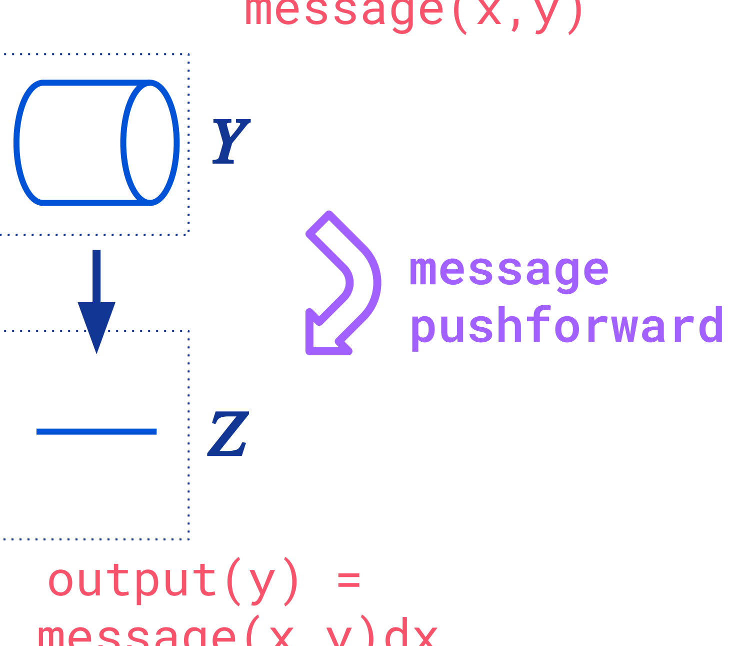
The integral transform, geometrically



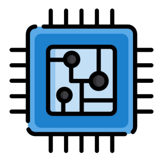
A nice carrier object is the **cylinder**, which we denote **Y**.
We *integrate* over the circle to get data on Z :)

How do we *produce* these messages?





The integral transform, geometrically



We need **arguments** (a set X) for the message computation, which come directly from the input using a copy or *tile* operation.

message(x,y)

X

Y

message pushforward

W

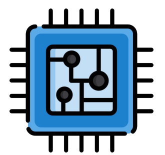
Z

input_a(x) = f(x)

input_b(y) = g(y)

output(y) = $\int_x$ message(x,y)dx

The integral transform, geometrically



Note, morphism $X \to W$ points the “wrong” way :)

argument
pushforward

arg_a(x,y)
= f(x)

arg_b(x,y)
= g(y)

message(x,y) =
f(x) * g(y)

X

Y

pullback

message
pushforward

W

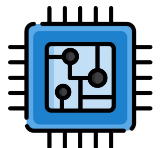
Z

input_a(x)
= f(x)

input_b(y)
= g(y)

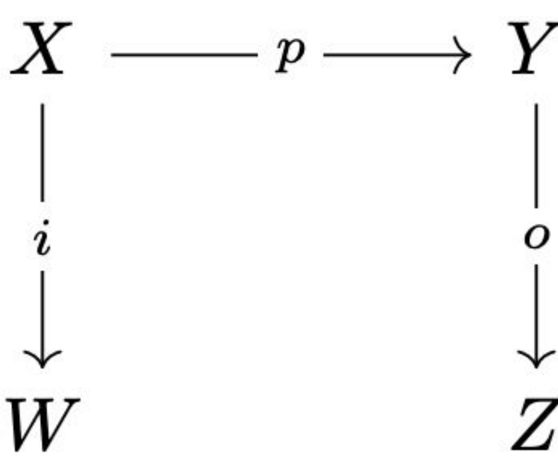
output(y) =
$\int_x$ message(x,y)dx

The integral transform, algebraically

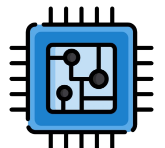


We define a **polynomial** to be a diagram of finite sets and functions between them, of the form given on the right.

To interpret such a diagram in terms of a transform of tensors, we need to fix a set R equipped with some kind of multiplication ⊗ and addition ⊕, a **semiring**.

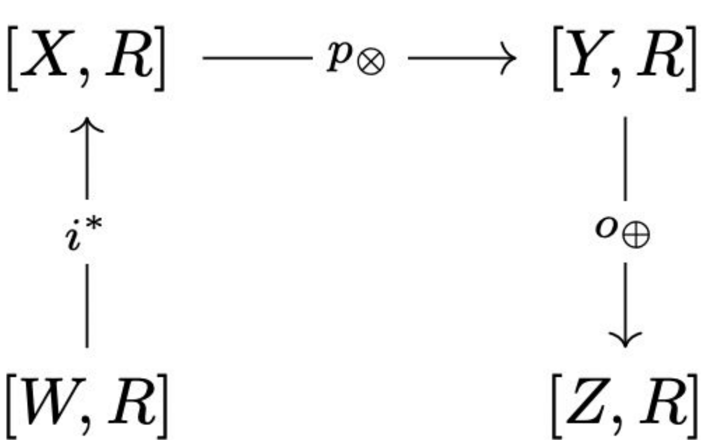


The integral transform, algebraically

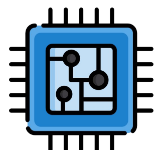


With R a semiring, we can produce R-valued tensors mapping input tensor to output tensors.

We call the three steps the **pullback**, (defined as a tiling operation) the **argument pushforward**, (defined using ⊗) and the **message pushforward** (defined using ⊕).

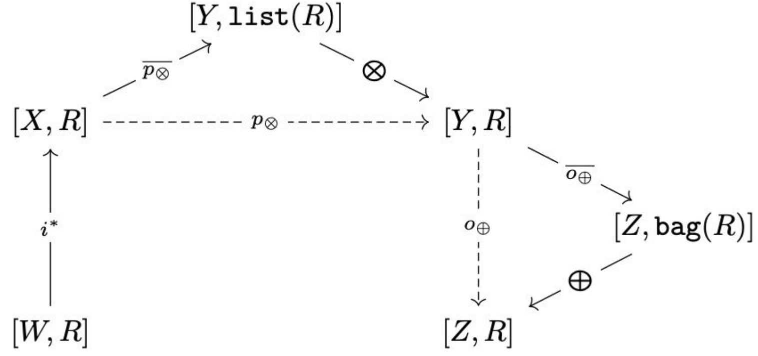


The integral transform, algebraically

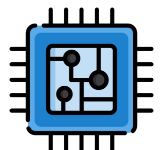


The basic idea is that a message is produced by “multiplying” an (ordered) list of arguments, and the output is produced by “adding” an (unordered) bag of messages.

By picking the right semiring R, we can use the same polynomial to describe either a GNN or a DP algorithm.

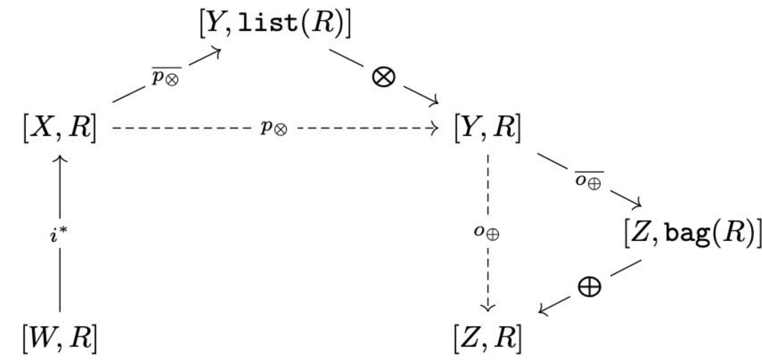


The integral transform, algebraically

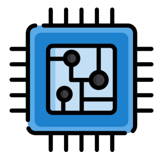


When R is the real numbers with the usual addition and multiplication, this describes convolution in neural networks, including basic message-passing.

When R is something more “tropical”, like the extended natural numbers with min and +, we get some classical algorithms!



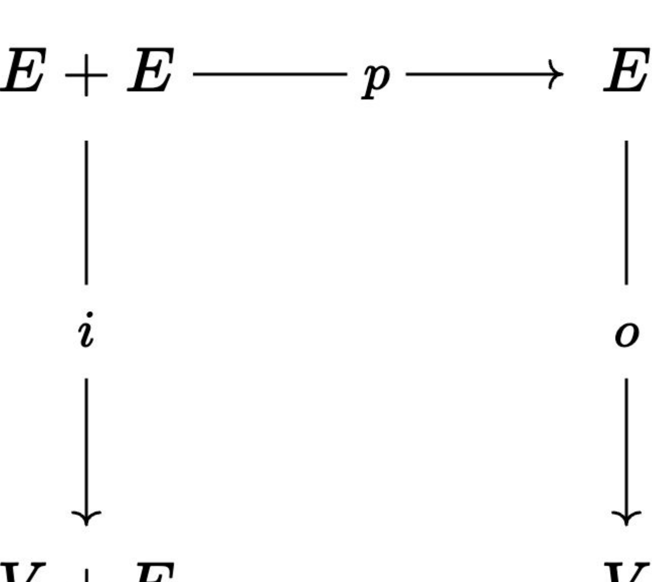
Example: Bellman-Ford



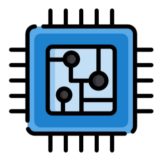
The polynomial on the right represents the Bellman-Ford algorithm.

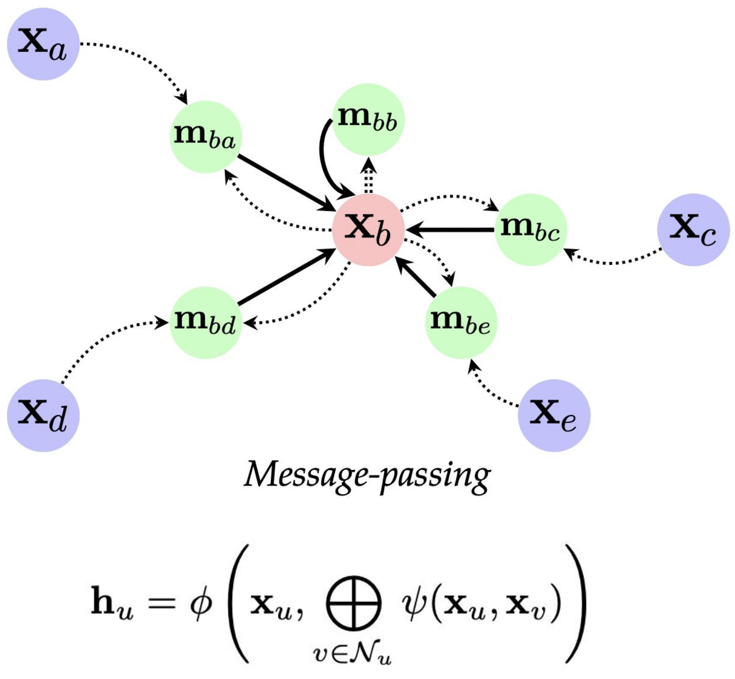
For simplicity, we add a self-edge of weight 0 to each vertex. Then the input consists of node values and edge weights, and the transform follows the usual formula:

$$d_u \quad \leftarrow \quad \min\left(d_u, \min_{v \in \mathcal{N}_u} d_v + w_{v \to u}\right)$$



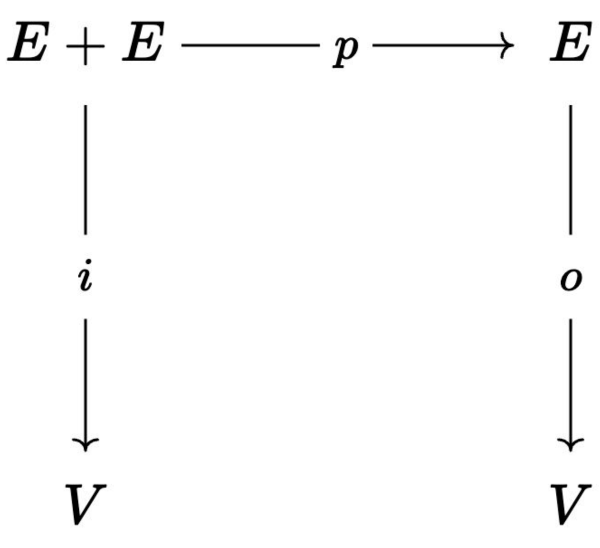
Example: Message passing



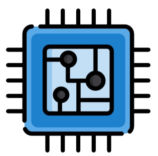


Message-passing

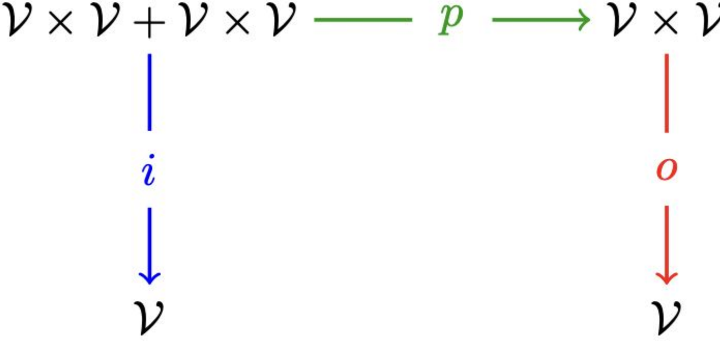
$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right)$$

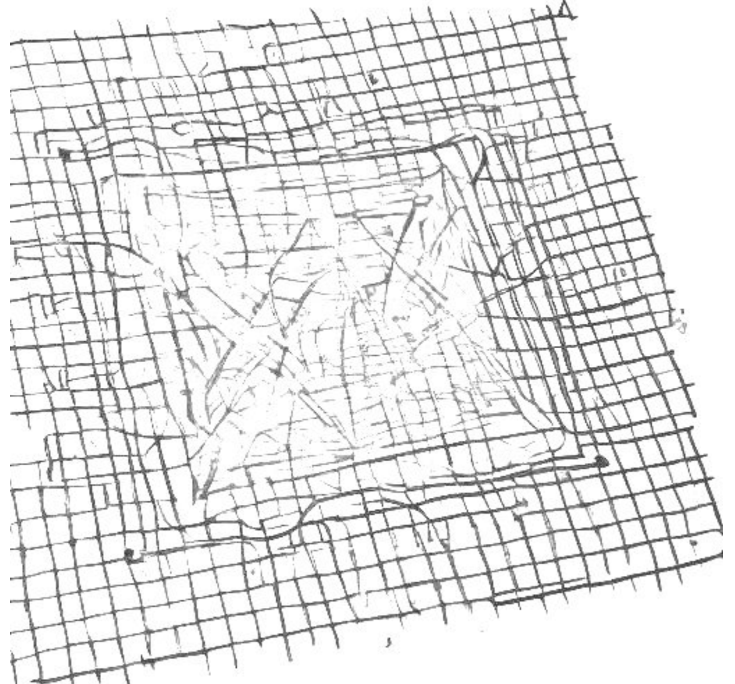


Pseudocode Alignment



Require:
  Node features $\mathbf{X} \in \mathbb{R}^{n\times k}$,
  Message function $\psi : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^m$,
  Update function $\phi : \mathbb{R}^m \to \mathbb{R}^m$
Ensure: Latent features $\mathbf{H} \in \mathbb{R}^{n\times m}$
  $\mathbf{Arg}^{\text{snd}} \leftarrow \texttt{tile}(\mathbf{X}, 0, n)$; // $\mathbf{Arg}^{\text{snd}} \in \mathbb{R}^{n\times n\times k}$
  $\mathbf{Arg}^{\text{rcv}} \leftarrow \texttt{tile}(\mathbf{X}, 1, n)$; // $\mathbf{Arg}^{\text{rcv}} \in \mathbb{R}^{n\times n\times k}$
  for $(u, v) \in \mathcal{V} \times \mathcal{V}$ **do**
    $\mathbf{msg}_{uv} \leftarrow \psi(\mathbf{arg}^{\text{snd}}_u, \mathbf{arg}^{\text{rcv}}_v)$; // $\mathbf{Msg} \in \mathbb{R}^{n\times n\times m}$
  end for
  for $u \in \mathcal{V}$ **do**
    $\mathbf{h}_u \leftarrow \phi(\bigoplus_{v\in\mathcal{V}} \mathbf{msg}_{vu})$;
  end for

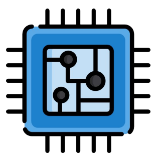




Code Examples

https://github.com/deepmind/clrs/tree/master/clrs/_src/processors.py

Code Examples: PGN



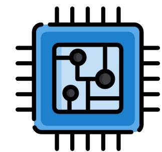
```
z = jnp.concatenate([node_fts, hidden], axis=-1)
m_1 = hk.Linear(self.mid_size)
m_2 = hk.Linear(self.mid_size)
m_e = hk.Linear(self.mid_size)
m_g = hk.Linear(self.mid_size)

o1 = hk.Linear(self.out_size)
o2 = hk.Linear(self.out_size)

msg_1 = m_1(z)
msg_2 = m_2(z)
msg_e = m_e(edge_fts)
msg_g = m_g(graph_fts)
```

$$V^2 + (V^2 + V^2) + V^2 \longrightarrow V^2$$

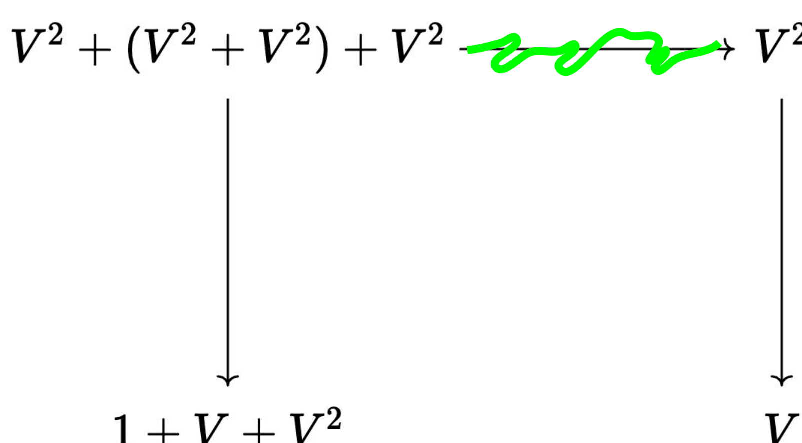
$$\downarrow \qquad\qquad\qquad \downarrow$$

$$1 + V + V^2 \qquad\qquad V$$

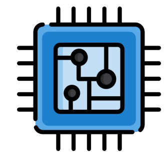
Code Examples: MPNN



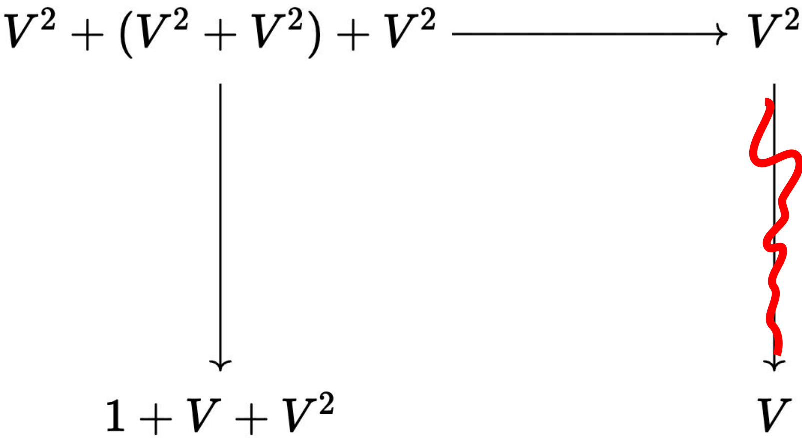
```
msgs = (
    jnp.expand_dims(msg_1, axis=1) + jnp.expand_dims(msg_2, axis=2) +
    msg_e + jnp.expand_dims(msg_g, axis=(1, 2)))
```



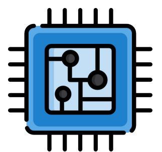
Code Examples: MPNN



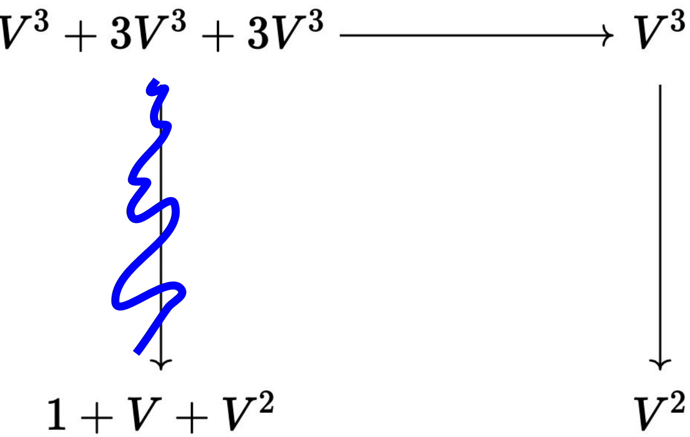
```
maxarg = jnp.where(jnp.expand_dims(adj_mat, -1),
                   msgs,
                   -BIG_NUMBER)
msgs = jnp.max(maxarg, axis=1)
```



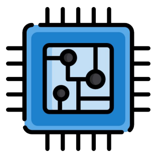
Code Examples: PGN-triplets



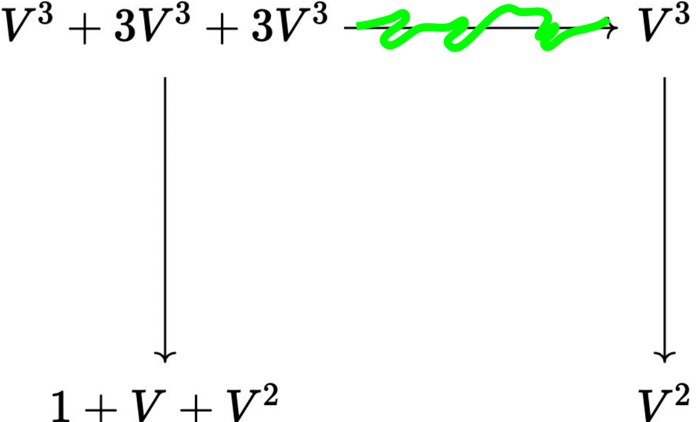
```
t_1 = hk.Linear(nb_triplet_fts)
t_2 = hk.Linear(nb_triplet_fts)
t_3 = hk.Linear(nb_triplet_fts)
t_e_1 = hk.Linear(nb_triplet_fts)
t_e_2 = hk.Linear(nb_triplet_fts)
t_e_3 = hk.Linear(nb_triplet_fts)
t_g = hk.Linear(nb_triplet_fts)

tri_1 = t_1(z)
tri_2 = t_2(z)
tri_3 = t_3(z)
tri_e_1 = t_e_1(edge_fts)
tri_e_2 = t_e_2(edge_fts)
tri_e_3 = t_e_3(edge_fts)
tri_g = t_g(graph_fts)
```



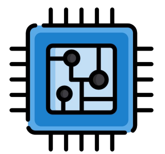
Code Examples: PGN-triplets



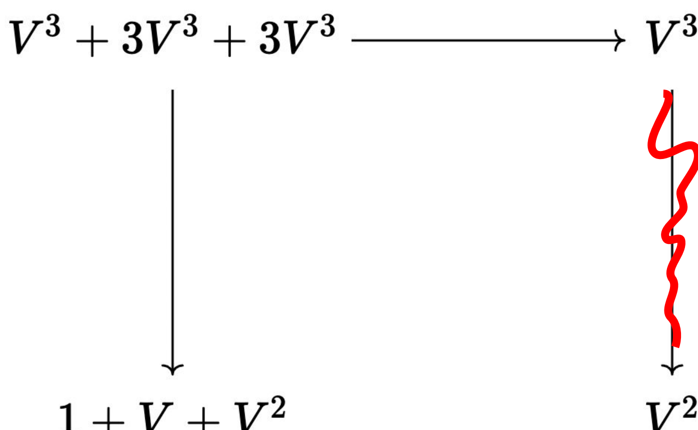
```
return (
    jnp.expand_dims(tri_1, axis=(2, 3))      +
    jnp.expand_dims(tri_2, axis=(1, 3))      +
    jnp.expand_dims(tri_3, axis=(1, 2))      +
    jnp.expand_dims(tri_e_1, axis=3)         +
    jnp.expand_dims(tri_e_2, axis=2)         +
    jnp.expand_dims(tri_e_3, axis=1)         +
    jnp.expand_dims(tri_g, axis=(1, 2, 3))
)
```



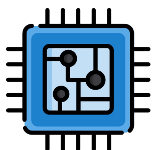
Code Examples: PGN-triplets



```
o3 = hk.Linear(self.out_size)
tri_msgs = o3(jnp.max(triplets, axis=1))
```



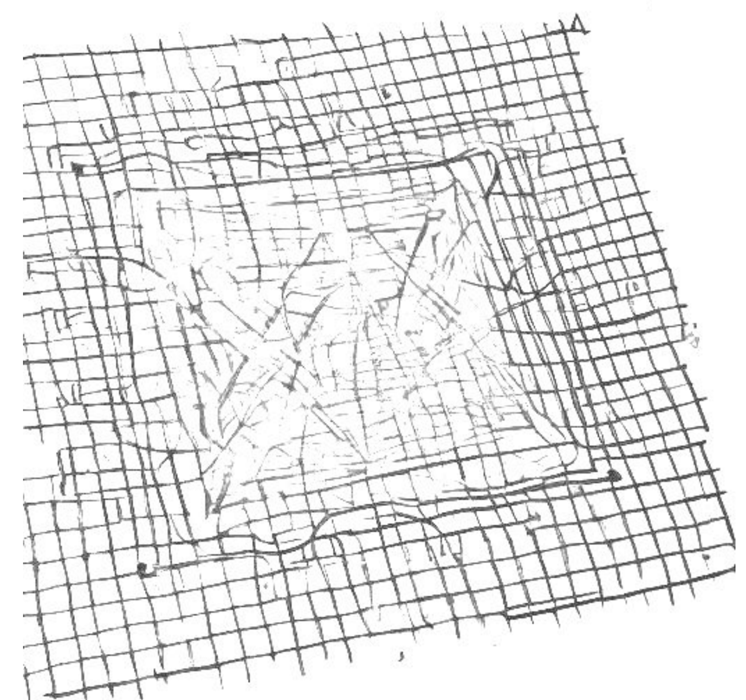
Type-checking GNNs



The triplet architecture came from the realization that the usual method for producing edge features in GNNs doesn't pass our polynomial "type checker".

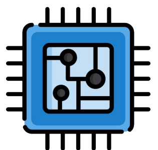
In the diagram on the right, o is not a function! Even though the architecture can be implemented in practice, this view makes it easy to tell that the edge representation is overloaded; we are performing an "illegal" copy.

$$\begin{array}{ccc} V^2 + (V^2 + V^2) + V^2 & \xrightarrow{\quad p \quad} & V^2 \\ \Big\downarrow i & & \Big\downarrow o \\ 1 + V + V^2 & & V + V^2 \end{array}$$



Monads,
Mo’ Problems

Monads



QUANTALES AND HYPERSTRUCTURES

Monads, Mo' Problems

by

ANDREW JOSEPH DUDZIK

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Mathematics

in the

Graduate Division

of the

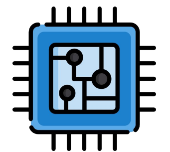
University of California, Berkeley

INTRODUCTION

We cannot understand what something is without grasping what, under certain conditions, it can become.

— Roberto Mangabeira Unger,
The Singular Universe and the Reality of Time

The list monad



$$\texttt{list} : \texttt{Set} \to \texttt{Set}$$

$$\eta_X : X \to \texttt{list}(X)$$

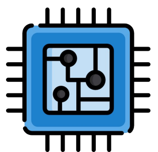
$$\mu_X : \texttt{list}(\texttt{list}(X)) \to \texttt{list}(X)$$

There is probably no type constructor more common in programming than the *list* functor.

Given a type X, we have a new type list(X) of the lists with entries from X.

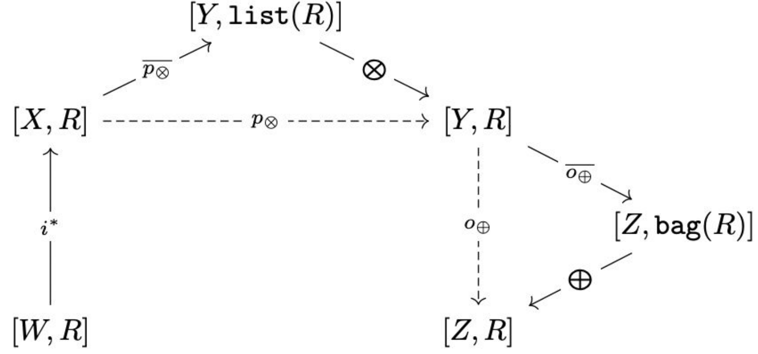
We also have a way to wrap instances of X into instances of list(X), and concatenation allows us to turn lists of lists into lists.

Message passing with monads

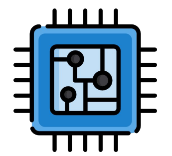


All of these operations work for bags as well! A *bag* or *multiset* is like a list, but without an ordering.

We use the list and bag monads to break down message passing; the messages are aggregated from lists of arguments, and the outputs are aggregated from bags of messages.



Monads

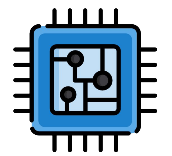


But there's more! These two monads have a *distributive law:*

$$\lambda : \mathtt{list}(\mathtt{bag}) \to \mathtt{bag}(\mathtt{list})$$

$$\lambda([B_0, B_1, \ldots, B_n]) = \langle [b_0, b_1, \ldots, b_n] \mid b_i \in B_i \rangle$$

Monads



This law leads us to the conclusion we stated earlier: our feature space should have the structure of a semiring.

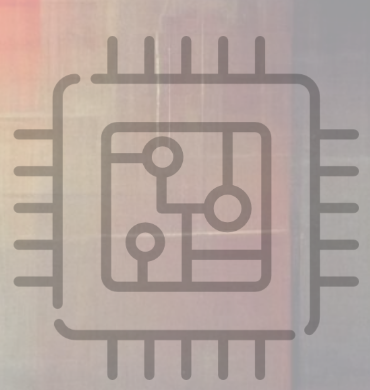
$$\lambda : \mathtt{list}(\mathtt{bag}) \rightarrow \mathtt{bag}(\mathtt{list})$$

$$\lambda([B_0, B_1, \ldots, B_n]) = \langle [b_0, b_1, \ldots, b_n] \mid b_i \in B_i \rangle$$

(see cats.for.ai for more details)

Thank you!

Questions?



adudzik@deepmind.com