Graph Networks in (Deep) Reinforcement Learning

Thibault Lahire[*]
Department of Mathematics
ENS Paris-Saclay / Master MVA
`thibault.lahire@student.isae-supaero.fr`

Abstract

This paper is the report of the project validating the MVA course "Graphs in Machine Learning" taught by Michal Valko. The aim of this paper is to show how graph networks work and review how these networks have been used in reinforcement learning. To do so, we first define what a graph network is, then we explain why there has been an increasing interest in this field. Finally, we provide an example of a reinforcement learning process where the agent learns directly from graphs.

1 Graph networks

1.1 Introduction

Definitions A *graph* G is defined by three elements : its nodes (or vertices) $V = \{\mathbf{v}_i\}_{i=1:N^v}$, its edges $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ and its global attribute $\mathbf{u}$. The r_k and s_k in the definition of E refer to the two nodes connected by the edge $\mathbf{e}_k$, with r_k the receiver node and s_k the sender node in the case of an oriented graph. Hence, a graph is denoted $G = (V, E, \mathbf{u})$.

A graph is a good way to represent interactions between objects. For instance, the n-bouncing balls problem can be seen as a graph. In this problem, there are n balls of fixed mass and diameter trapped in a box. The balls are submitted to the law of elastic collision and gravity, and some balls are connected by a spring, with a fixed spring constant k. To model this physical situation as a graph, the following model can be set : V represents the balls, with attributes for position, velocity, radius and mass. E represents the presence of springs between different balls, and their corresponding spring constants. $\mathbf{u}$ represents the total energy of the set of balls (other choices are also correct).

In this project, a key idea is to use graphs as input and output of functions. Such a function will be called a Graph Network (GN) block. The word *block* comes from the idea that this function can be part of a broader architecture where it plays the role of a simple entity. In the simple example above, we typically want to have a simulator that shows the behavior of the set of balls.

It is important to understand the two possibilities we have. On the one hand, we can build a function that predicts the state of the system at time $t + 1$ given the knowledge of the system at time t and all the rules it has to follow. On the other hand, we can build a function that predicts the state of the system at time $t + 1$ given (only) the trajectories from time 1 to time t.

In the first case, the Newton laws are manually coded inside the GN block. However, the relations between nodes are simple in this example, but it could be very tedious for a more complex problem. A solution to this issue is described in the second case, where the GN block observes the first t time steps and then predicts the future state of the system. When a GN block does so, it *trains* during the first t time steps, in other words, it learns by himself the Newton laws by observing the system. As a consequence, whereas the functions were manually coded in the first case, the second case falls

[*]Under the supervision of Omar Darwiche Domingues

Preprint. Under review.

into the deep neural network paradigm. Indeed, the functions are deep architectures whose goal is to approximate as best as possible the Newton laws. When GN blocks are used to infer relations and predicts future time steps, they are called *interaction networks* and studied in detail in [1].

The rest of this chapter explains the internal structure of a GN block, the relational inductive biases and ends with the issue of combinatorial generalization. In the next chapter, we illustrate the use of a GN block on the n-bouncing balls problem and discuss the two cases described above.

1.2 Internal structure of a GN block and computational steps

A GN block contains three "update" functions, ϕ^e, ϕ^v, ϕ^u, and three "aggregation" functions (also called message passing functions), $\rho^{e\rightarrow v}$, $\rho^{e\rightarrow u}$, and $\rho^{e\rightarrow v}$.

When the GN block is called, ϕ^e updates the edges, ϕ^v the nodes, and ϕ^u the global attribute of the graph. For each node $\mathbf{e}_k$, we have the update $\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$. The ρ functions each take a set as input, and reduce it to a single element which represents the aggregated information. Indeed, $\rho^{e\rightarrow v}$ is such that $\rho^{e\rightarrow v}(E^{'}_i) = \overline{\mathbf{e}'_i}$, where $E^{'}_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,k=1:N^e}$. All the information resulting from the edge update is contained in $\overline{\mathbf{e}'_i}$.

Once the edges have been updated, the nodes are actualized : $\mathbf{v}'_i = \phi^v(\mathbf{v}_i, \overline{\mathbf{e}'_i}, \mathbf{u})$. Then, $\rho^{v\rightarrow u}$ aggregates all the information on the updated nodes : $\overline{\mathbf{v}'} = \rho^{v\rightarrow u}(V')$ where $V' = \{\mathbf{v}'_i\}_{i=1:N^v}$, the set of updated nodes.

The last component to update is the global attribute of the graph $\mathbf{u}$. To do so, we define another aggregate element : $\overline{\mathbf{e}'} = \rho^{e\rightarrow u}(E')$ where $E' = \cup_i E^{'}_i$. While $E^{'}_i$ is the set of updated edges directed to the node $\mathbf{v}_i$, E' is the set of all updated edges. E' can also be written : $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$.

Eventually, $\mathbf{u}$ can be updated : $\mathbf{u}' = \phi^u(\overline{\mathbf{e}'}, \overline{\mathbf{v}'}, \mathbf{u})$.

Then, the GN block is able to return an updated graph $G' = (E', V', \mathbf{u}')$.

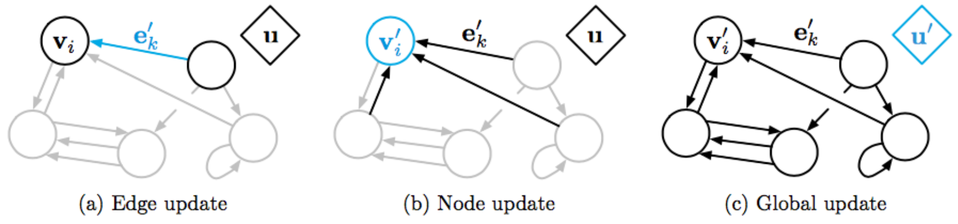

Figure 1: Updates in a GN block : the edges are updated first whereas the global attribute is updated last

In this description (inspired by [6]), a choice was made. The edges are the first to be updated, then the nodes, and then the global attribute. This is not the only way to proceed. In particular, the nodes can be updated before the edges, as is the case in [7]. In the appendix of this report, you will find a summary of the algorithm and the functions involved.

1.3 Relational inductive biases in graph networks

Definition An inductive bias is a set of assumptions of a learning algorithm that leads it to choose one hypothesis over another regardless of the observed data.

For instance, these assumptions may be encoded in the prior of a Bayesian model [8], or instantiated via architectural assumptions in a neural network. In the case of a convolutional neural network, there is an inductive bias of translational invariance that some authors call a "spatial inductive bias" because it builds in specific assumptions about the spatial structure of the world.

Definition Similarly, a relational inductive bias builds in specific assumptions about the relational structure of the world.

In probabilistic models, especially those based on the Bayesian framework, many relational inductive biases can be naturally found. On the contrary, the majority of deep learning architectures are not based on strong assumptions. To be convinced, we can think of a feed-forward neural network whose objective is to say if a mail is a spam or not. This network can be viewed as a (black-box) function that takes a mail as input and return 0 or 1 to indicate the nature of the mail. The only bias of such a function is its training set, i.e. a huge quantity of mails previously classified as 0 or 1.

As well as a deep learning architecture, a GN block (or, more precisely, the functions ϕ and ρ inside the block) is often built from a training set. Details will be provided later, but let's consider for now that the functions ϕ and ρ are based on feed-forward neural networks, as is the case in [5]. The key difference between a GN block and a more common deep learning architecture is the structure of the input data. In the first case, it is a graph, which means that the user specifies the number of objects involved (the nodes) and the interactions between them (the edges). In the second case, the input data are just objects with no specified structure, such as a mail that must be classified as spam or not. Forcing the network to use graphs is a relational inductive bias.

1.4 Combinatorial generalization and learning transfer

A problem is said *combinatorial* when solving it becomes difficult when the number of objects involved grows large. The common deep learning architectures often struggle when faced with structured, combinatorial problems. Indeed, most of these architectures are used to assign an input to a class (classification problem) or to approximate a function (regression problem). When such an architecture meets an object it has never seen before (i.e. it has not been trained on examples with a similar structure), it performs badly.

On the contrary, a graph network is more adaptable to combinatorial generalization, since the new covariates can be seen as a new graph, just like the others met before (i.e. during the training phase). As long as the new example can be implemented as a graph, the architecture can operate on.

In the work done in [2], the authors have used a graph network on different objects (represented by graphs) such as a swimmer or a cheetah. Swimmer and cheetah (see Fig. [2]) are two well-known study cases in the reinforcement learning community provided by the *OpenAI Gym* library (Python) [4]. The aim was to train their physical systems (cheetah, swimmer) to make them achieve a certain goal (reaching a certain position as fast as possible for example). In one experiment, they trained a certain physical system (for example the swimmer), and apply the GN model to another physical system (for example cheetah). Even if the *learning transfer* was not as good as a dedicated training, the movements of the physical systems were not incoherent.

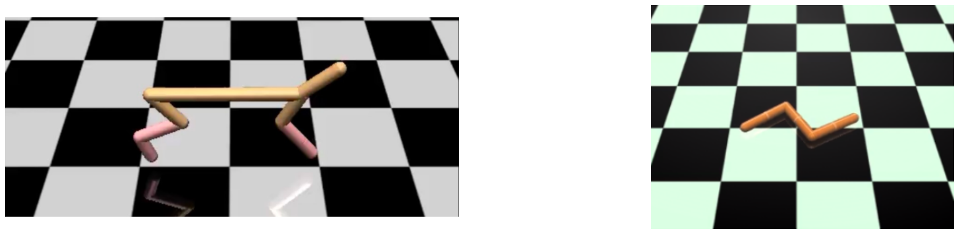

Figure 2: Cheetah (right) and swimmer (left) provided by the OpenAI Gym / Mujoco environment. These two objects can be seen as graphs where nodes are articulations between two body members. They are able to move toward the grid by modifying the angle between body members.

This learning generalization is a hope for a better understanding of human intelligence. Indeed, human reasoning is rooted in a rich system of knowledge about **objects** and **relations** [9] which can be composed to support powerful forms of combinatorial generalization. Von Humboldt wrote in 1836 that a characteristic of human reasoning was the ability to make "infinite use of finite means", as is the case for the language, where a small set of elements (such as words) can be productively composed in limitless ways (such as into new sentences). We now understand why graphs are an hopeful approach, since they are themselves composed of **objects** (nodes) and **relations** (edges).

Another advantage of graphs is the representation of entities and relations as sets (set of nodes, set of edges), which means GNs are invariant to permutations, i.e. invariant to the order of elements [6], which is often desirable. For example, the objects in a scene do not have a natural ordering.

2 Illustration

In the first sections, we define graphs and show how to proceed on graphs. It is now time to give examples of architectures processing graphs. We also wrote that the functions ϕ and ρ involved in the GN blocks are often complex architectures such as feed-forward neural networks. The n-bouncing balls problem is reused to show that GN blocks (and more generally graphs) are a very bad answer when the problem is "simple", but they are a very powerful paradigm for inferring relations on complex data.

2.1 Examples where input data are naturally graphs

A simple example We now detail the n-bouncing balls problem presented in the introduction of this paper. Imagine that v_i represents the i-th ball, with attributes for position, velocity, and mass. E represents the presence of springs between different balls, and their corresponding spring constants. E also encodes the forces that exist between nodes. **u** represents the total energy.

Everything is known at time t, we apply the GN block to this graph to know the graph at time $t + 1$.

- 1. ϕ^e is applied per edge, which means that the forces or potential energies between two connected balls are updated.
- 2. $\rho^{e \to v}$ is applied, which means summing all the forces or potential energies acting on the i-th ball.
- 3. ϕ^v is applied on each node i to update its attribute, typically the position and velocity of each ball.
- 4. $\rho^{e \to u}$ is applied to compute the summed forces or the spring potential energy.
- 5. $\rho^{v \to u}$ computes the total kinetic energy of the system.
- 6. ϕ^u is applied on the whole graph to compute the total energy of the physical system.

The six functions (ρ and ϕ) implied in this algorithm are deterministic and perfectly known by the user. Indeed, it is simple (but it can be long...) to write all the equations involving a specific ball (or node) and governing its movement. Such a dynamical system, at this scale, typically follows Newton laws, which include derivatives of functions. As long as you can write $f(t + \Delta t) \simeq f(t) + \Delta t f'(t)$, you can predict the state of the system at time $t + \Delta t$ if you know everything at time t.

2.2 Remarks : graphs are not always the best answer !

The simple example described above was an illustration of a physical system that can be represented as a graph. More specifically, we saw how this graph can be seen as a function input and output (the GN block).

However, even if it is easy to represent complex physical systems as graphs, we may wonder why such a representation is necessary if all the relations are explicitly known. Indeed, considering a graph when dealing with physical systems implicitly means that we are computing a *time-driven* simulation.

When the system is *simple*, this is clearly not the best way to proceed. Consider the n-bouncing ball case, where the balls are trapped in a box, and there is no spring between them and no gravity. The ball movement is thus rectilinear and the balls are submitted to the laws of elastic collisions. Computing a *time-driven* simulation is an approach that suffers of two main drawbacks :

- if you consider n balls, a one second simulation takes a time proportional to $n^2/\Delta t$ (checking for collisions is quadratic) which is not suitable for large n.
- if Δt is too big, you may miss collisions.

However, there exists another approach: The idea is to use a simulation guided by events, particularly collision events, because it is easy to update the balls positions between two collisions (recall that the trajectory of the balls is rectilinear in absence of collisions). The main loop of an *event-driven* simulation will be the following:

- 1. find the next collision event e to process (i.e. the one that will occur first among all possible collision events)
- 2. update the balls positions to the time of event e and update the state of the balls concerned by the collision
- 3. go back to step 1 until you reach a fixed time limit

The bottleneck of the algorithm is clearly step 1 in which all the possible collisions are computed. We can use an efficient data structure to store the events, in order to be able to efficiently retrieve the earliest event and also to efficiently store new events. For instance, a priority queue can be used. If we implement this priority queue using a binary heap, we reduce the complexity of the n-bouncing balls problem, since the operations on a binary heap are executed in $O(log(n))$.

To conclude, on *simple* physical systems (especially when the six functions ρ and ϕ are perfectly known), such as the n-bouncing balls problem, some computational techniques are more efficient than considering a whole graph. However, when the functions ρ and ϕ are learned from a database, and when the input of the GN block is not always the same graph (e.g. a different number of nodes and edges from an input to another), the GN paradigm remains a powerful approach both for prediction and combinatorial generalization. We now describe this point on several examples.

2.3 How to learn ρ and ϕ ?

The n-bouncing balls problem can also be learnt. As written in the introduction, the functions ρ and ϕ have to learn Newton laws. In this case, ρ and ϕ can be MLPs (Multi-Layer Perceptron) or any other deep architecture that can play the role of a universal function approximator. We note θ_1 the parameters of the network ρ and θ_2 the parameters of the network ϕ. The GN block learns by observing how the physical system evolves. After the training, it is able to infer the future time steps.

The user gives to the GN block the true trajectories and a loss is defined for both approximators. During the training phase, the ρ-approximator and the ϕ-approximator compare their predictions to the true trajectories. If the predictions are totally false, then the loss is high. If they are near to the truth, the loss is low. The aim of the training phase is to minimize the loss by updating θ_1 and θ_2 at each time step (or over epochs).

Using deep architectures on the n-bouncing balls problem may seem unnecessarily heavy. To conclude this chapter, we give an example which reveals the power of deep architectures. Kipf et al. [10] use a Variational Auto Encoder [11] (very different from the interaction network of Battaglia et al. [1]) and try to predict the future movements of 4 basketball players in the specific pick and roll configuration. The encoder creates graphs with 5 nodes (the ball and the 4 players) and the decoder predicts the future time steps.

In this case, the meaning of the edges constructed by their model is much more difficult to understand. For the n-bouncing balls, the edges are the springs. But what relations are there between 2 players or between the ball and a player ? Actually, there is no physical meaning of the created edges. The network has learnt a *latent representation* that is useful to model the future movements of the 5 nodes (and it works well!), but there is no concrete meaning. As for the n-bouncing balls, the training of the ρ-approximator and the ϕ-approximator was done by observing the scenes a few seconds.

In the next chapter, we explain another study case where ρ and ϕ are deep networks. Contrarily to the previous examples, ρ and ϕ results from of a reinforcement learning process, in other words and to remain general, ρ and ϕ have been learnt through a trial-and-error procedure where the goal was to maximize a cumulative sum of rewards.

3 Reinforcement learning on graphs

One example of a complex problem based on a graph is studied by Jessica B. Hamrick et al. in [3]. The GN here constructed receives a positive reward if the GN takes the proper decision, and receives a negative reward if the GN takes the wrong decision. Hence, the framework of this problem is a reinforcement learning approach.

3.1 Description of the problem

In this problem, the participants (or the machines) were given a 2D tower built from rectangular blocks. In the first phase, gravity is not applied on the tower, and the goal of the game is to apply glue between blocks to ensure the stability of the tower when the gravity will be applied (second phase).

All the towers presented to the participants are such that at least one block moves when gravity is applied. Three examples of performing the task are shown in Fig. [3]. Green blocks in the gravity phase indicate stable blocks. Top: no glue is used, and only one block remains standing (+1 points). Middle row: one glue is used (-1 points), resulting in three blocks standing (+3 points). Bottom row: two glues are used (-2 points), resulting in a stable tower (+6 points); this is the minimal amount of glue to keep the tower stable (+10 points). See `https://goo.gl/f7Ecw8` for a video demonstrating the task.

The goal of this experiment is to compare the performances of humans, common deep learning architectures (such as a multi-layer perceptron (MLP) which belongs to the class of feed-forward neural network), and graph networks (GN).

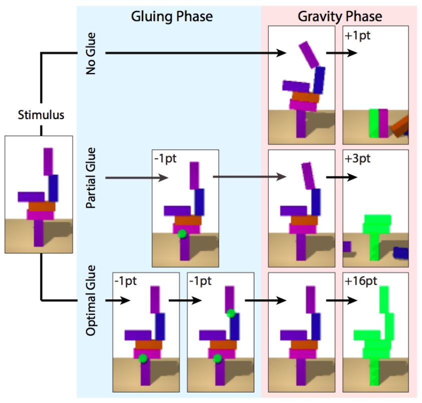

Figure 3: The gluing task and its rewards

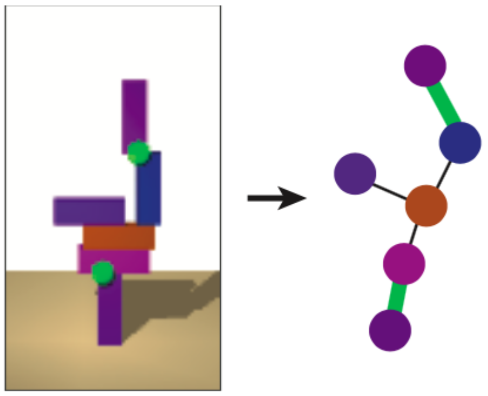

Figure 4: The problem seen as a graph

3.2 Modeling as a graph

As shown in Fig. [4], the tower can be seen as a graph, where the positions and orientations of the blocks are encoded as nodes, the link between blocks is encoded with the presence of edges, and the edge attribute indicates whether there is glue (1) or not (0). In input of a block (MLP or GN), there is no glue, so all the edges attributes are set to 0. The goal of the block is to decide where 0's have to be turned into 1's.

In terms of programming, the inputs of the MLP agent consists of the concatenated positions and orientations of all objects in the scene, as well as a one-hot vector of size $E_{fc} = N(N-1)/2$ encoding each pair of blocks and the floor. There are $E_{fc}+1$ outputs in the final layer (there is an additional output for the “stop” action).

The inputs to the GN agent also include the positions and orientations of all objects in the scene, but the “glue” vector instead has size $E_{sparse} \simeq N$ (where E_{sparse} is the number of pairs of blocks in contact). It means that the GN agent is told which blocks are in contact ! There are $E_{sparse}+1$ outputs in the final layer.

The GN agent has to choose where to put glue among blocks that are already in contact, whereas the MLP agent may put glue between blocks that are not in contact. This information is a relational inductive bias for the GN agent; it avoids the GN to choose an obviously wrong glue position.

As for humans, the input is the view of the tower...

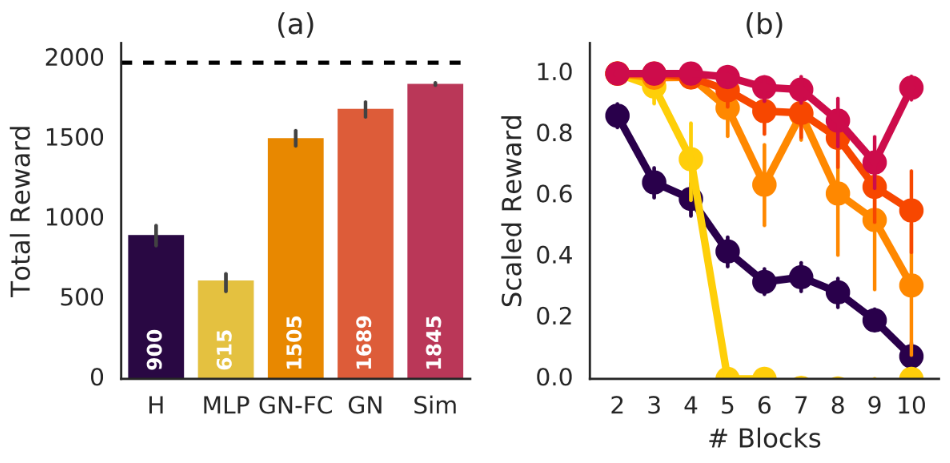

Figure 5: Results of the comparison between the 5 agents

3.3 Results

Fig. [5] presents the results of the experiments, where the performance of 5 agents are compared : H for humans, MLP for Multi-Layer Perceptron, GN-FC for Graph Network on a Fully-Connected graph (which means that this agent has the same inputs as the MLP, but the "glue" input vector is transformed into a fully-connected graph whereas it remains a vector for the MLP), GN for Graph Network operating over a sparse graph and Sim for Simulation (i.e. an agent that knows the forces that will be present when gravity will be applied).

The left subfigure presents the (average) reward obtained over experiments for each agent. The MLP agent provides very bad results compared to the GN-FC agent, which has exactly the same inputs but a different processing. This comparison shows that an internal graph structure is a better structure for problems where graphs are in inputs. The GN agent is even better than the GN-FC agent. It is not surprising, since this agent has a strong relational inductive bias that tells him which blocks can (or cannot) be glued. Without any surprise, the simulation agent performs best, since it is able to compute the effects of the forces that will be applied on the towers.

The right subfigure is linked to the combinatorial generalization explained earlier in this report. On the y-axis the rewards have been rescaled : 0 corresponds to the reward when no action was taken, while 1 corresponds to the optimal reward. On the x-axis the number of blocks in the tower is found. As we see, the performance of every agents decreases when the number of blocks in the tower increases. Indeed, the problem is more difficult when the number of blocks is large. However, some agents generalize better than others. We do not study the case of the simulation agent, because it has access to all the forces.

During the training, no one has seen towers with 7 to 10 blocks. The MLP agent has a very bad result and does not know what to do when the number of blocks is larger than 5. The humans are almost unable to build stable towers when there are 10 blocks. The GN agent has a quite good generalization and is able to construct stable 10-blocks towers approximately 50% of the time.

To conclude on this set of experiences, this study proves that reinforcement learning combined to graphs seems to be an hopeful approach to deal with combinatorial generalization.

Conclusion

This review project is an overview of the recent uses of graphs in complex architectures to tackle the problem of combinatorial generalization, i.e. an issue closely linked to human brain modeling. From the modeling point of view, it appears that graphs are a powerful structure which seems to work similarly to our brain. From the programming point of view, we now have the computational abilities to deal with graphs in complex architectures. In particular, an hopeful approach to solve the "problem of intelligence" seems to be the reinforcement learning paradigm using graph networks.

Acknowledgments

I thank my tutor Omar Darwiche Domingues for his availability and his advice.

References

[1] Battaglia, P. et al. 'Interaction Networks for Learning about Objects, Relations and Physics' (https://arxiv.org/abs/1612.00222)

[2] Sanchez-Gonzalez, A. et al. 'Graph networks as learnable physics engines for inference and control' (https://arxiv.org/pdf/1806.01242.pdf)

[3] B. Hamrick, J. et al. 'Relational inductive bias for physical construction in humans and machines' (https://arxiv.org/abs/1806.01203)

[4] `https://gym.openai.com/envs/#classic_control`

[5] Scarselli, F., Gori, M., Tsoi, A., Hagenbuchner, M. and Monfardini, G. 2009, 'The graph neural network model', IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61-80 : `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1015.7227&rep=rep1&type=pdf`

[6] Battaglia, P. et al. 'Relational inductive biases, deep learning, and graph networks' `https://arxiv.org/pdf/1806.01261.pdf`

[7] Kansky, K., et al. (2017). Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In Proceedings of the International Conference on Machine Learning (ICML).

[8] Griffiths, T. L., Chater, N., Kemp, C., Perfors, A., and Tenenbaum, J. B. (2010). Probabilistic models of cognition: exploring representations and inductive biases. TiCS, 14, 357–364.

[9] Spelke, E. S., and Kinzler, K. D. (2007). Core knowledge. Develop- mental Science, 10, 89–96.

[10] Kipf T., et al., 'Neural Relational Inference for Interacting Systems' `https://arxiv.org/abs/1802.04687`

[11] Kingma D., et al., 'Auto-Encoding Variational Bayes' `https://arxiv.org/abs/1312.6114`

Appendix

Here is a short summary of a GN block structure, as presented in [6].

$$
\begin{aligned}
\mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \qquad \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\
\mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \qquad \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right) \\
\mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & \qquad \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right)
\end{aligned}
$$

Figure 6: Summary of the functions involved in the GN block

Algorithm 1 Steps of computation in a full GN block.

```
function GraphNetwork($E$, $V$, $\mathbf{u}$)
    for $k \in \{1 \ldots N^e\}$ do
        $\mathbf{e}'_k \leftarrow \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right)$        ▷ 1. Compute updated edge attributes
    end for
    for $i \in \{1 \ldots N^n\}$ do
        let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\; k=1:N^e}$
        $\bar{\mathbf{e}}'_i \leftarrow \rho^{e \rightarrow v}\left(E'_i\right)$        ▷ 2. Aggregate edge attributes per node
        $\mathbf{v}'_i \leftarrow \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right)$        ▷ 3. Compute updated node attributes
    end for
    let $V' = \{\mathbf{v}'\}_{i=1:N^v}$
    let $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
    $\bar{\mathbf{e}}' \leftarrow \rho^{e \rightarrow u}\left(E'\right)$        ▷ 4. Aggregate edge attributes globally
    $\bar{\mathbf{v}}' \leftarrow \rho^{v \rightarrow u}\left(V'\right)$        ▷ 5. Aggregate node attributes globally
    $\mathbf{u}' \leftarrow \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right)$        ▷ 6. Compute updated global attribute
    return $(E', V', \mathbf{u}')$
end function
```

Figure 7: Update algorithm of a GN block