Reinforcement Learning in Graph-based Environments

Supervisor: Michal Valko

Marc Ducret **Florentin Guth**
École Normale Supérieure
45 rue d'Ulm, 75005 Paris, France
firstname.lastname@ens.fr

Abstract

The goal of this paper is to exploit the graph structure of environments to outperform standard reinforcement learning algorithms. We propose a new environment designed with this goal in mind, with several extensions to make the problem more difficult as the state of the art progresses, and an efficient GPU batched implementation. We conduct a review of graph-based convolutions and propose several neural network architectures suitable for deep reinforcement learning on graphs, and show experimentally that our novel DECOUPLED architecture outperforms classical architectures.

Introduction

Many reinforcement learning environments involve an underlying graph structure. Standard approaches usually let the algorithm learn the structure of the graph, by embedding it in a grid when it makes sense (as is most often the case). Such an embedding is however useful, as it allows to use convolutions (extensively), which made the success of most deep neural networks architecture. It is therefore reasonable to expect some progress to be made by designing architectures that take advantage of the graph structure and are inspired from euclidean convolutions.

Section 1 presents the environment and its several extensions. Section 2 conducts a review of ways to generalize convolutions to graphs, and Section 3 explores several ways to use those convolutions in neural networks architecture. Section 4 trains these architectures on the environment and compares their performance.

1 The environment

We create an environment to train a robot vacuum cleaner. We thus fix a graph on which the robot will move, based on a layout of rooms and corridors. The graph will remain fixed for all our experiments, we leave the study of generalization from one graph to another to future work. Our graph is a subset of the 2D euclidean grid, which makes sense since when a robot vacuum cleaner maps a continuous real-world area, we can chose to discretize it regularly (and the structure of houses and apartments is well-suited to an orthogonal grid). Besides, it allows us to strike a balance between a fully euclidean grid (which is well understood) and a totally general graph (where designing good-performing convolutions is notoriously hard). Designing good performing architectures in this setting would be a first step towards the long-term goal of convolutions for general graphs.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

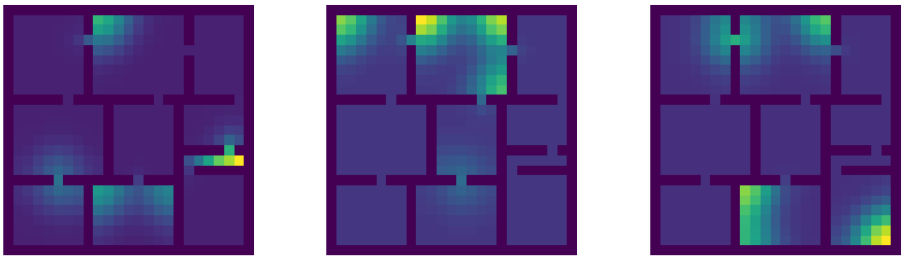

Figure 1: Examples of dust probabilities

The goal of the robot is to keep the environment clean. Based on the graph, we generate randomly dust spawning probabilities. Dust is more likely to spawn in corners, and the dust probability vary smoothly from one node to another (see figure 1). However, for the sake of simplicity, we fix the dust probabilities. This enables us to not have to deal with recurrent networks, which are harder to train, and focus on the design of the convolutions. We leave the design of a neural network architecture that would learn the dust spawning probabilities in an online way in each instance (but possibly after some offline training) to future work.

Since we fix the graph and the dust probabilities in all the following, the state of the environment is thus defined by the position of the robot on one node of the graph, and the presence or not of dust in each of the nodes. The observation we feed to the robot is composed of the structure of the graph (represented as a walkability mask over the regular grid) which never changes, its position on it, and the presence of dust or not in a small area surrounding the robot, defined as the k-hop neighborhood of the robot ($k = 6$ in our experiments, to relate with the size of the grid which is 25×25, including surrounding walls). Once again, we leave the task of mapping the area the robot is in to future work, as we focus here on non-recurrent architectures. We do not show the robot the dust spawning probabilities, as learning them is part of the challenge (but they remain fixed). Even with this, as the environment is partly observable, to be able to perform perfectly one would need memory (for instance, to remember what are the areas you just explored). To alleviate this need, we also provide the robot with the number of steps it last visited each node, albeit normalized and fed through a $\tanh$ to ensure it remains in $[-1, 1]$.

The robot can take four actions at each step, which are moving in any of the four cardinal directions. This is another argument in favor of our graphs being subsets of a grid: it is easier if the number of actions one can take remain constant at each step, which is not the case for general graphs where different nodes might have very different number of neighbors, and there is no clear identification (even partial) between the neighborhoods of different nodes. The `step` function is defined as follows: the robot moves to the selected neighbor (if the destination is not a wall), dusts spawn independently in each cell according to the dust spawning probabilities, and the robot cleans the cell it is on. The robot receives $+1$ reward if it cleans a dust, and -1 if it tries going into a wall.

There are two notable things about the implementation of the environment:

1. the environment is implemented in Pytorch, which means it can run on GPU,
2. the environment is batched, which means it runs simultaneously copies of itself, and the step function accepts the actions for every copy of the environment and returns all observations and all rewards.

These improvements, which are not common (they do not fit the OpenAI Gym framework), enables an easy interfacing with neural networks, which can do batched forwards efficiently, without any multithreading and CPU/GPU transfer required.

2 Convolutions

Convolutions are an important part of the success of deep learning in computer vision, and arguably so in reinforcement learning as well, with for instance AlphaGo [Silver et al., 2017]. We believe the success of convolutions comes from the following properties:

1. They have few parameters thanks to parameter-sharing, which means they are easier to train than fully-connected layers,
2. They are fast to compute, taking advantage of their locality,
3. They are translation-covariant, which means they exploit a powerful prior on the problem.

While convolutions are extremely adapted to data with a grid spatial structure such as images or Go boards, there are many cases where one does not have such a grid structure. However, the data is rarely totally devoid of structure: a frequent example being data on nodes of a graph, be it a social network, the rail system of France, or a 3D mesh. It remains an open research problem how to generalize convolutions to graph-based data in a way that keeps the 3 properties discussed above. We now conduct a small review of the state of the art in this domain, dubbed "Geometric Deep Learning" (see Bronstein et al. [2017] for a more comprehensive review), and derive a few architectures for our reinforcement learning task.

2.1 Spectral Convolutions

Spectral convolutions, first defined in Bruna et al. [2013], define general convolutions to general graphs, based on the convolution theorem, which states that $\widehat{f * g} = \widehat{f}\widehat{g}$, i.e., convolutions are diagonalized by the Fourier basis $(e^{i\omega \cdot})_\omega$. Thus, we need only find the equivalent of a Fourier basis in general graphs.

The Fourier basis is also the basis of eigenvectors of the (euclidean) Laplacian $\Delta = \sum_{i=1}^{d} \frac{\partial^2}{\partial {x_i}^2}$, since $-\Delta e^{i\omega \cdot} = \|\omega\|_2^2 e^{i\omega \cdot}$. However, we do have a definition of the Laplacian on general graphs $L = D - W$, where $w_{i,j}$ is the weight on the edge $i \leftrightarrow j$ and $D = \mathrm{diag}(W\mathbf{1})$. We do not consider a normalized Laplacian for the sake of simplicity. One should note that the Laplacian L defined here is non-negative, whereas its euclidean counterpart is non-positive. It is simply a matter of conventions, and we keep to them so as not to confuse the reader.

This parallel allows us to define spectral convolutions as matrices of the form $\Phi \Lambda \Phi^T$, where $\Phi = [\phi_1 \cdots \phi_n]$ is the Laplacian eigenbasis and Λ is any diagonal matrix. See figure 2 for the first eigenvectors of the Laplacian on our graph. As we can see, the eigenvectors capture meaningful patterns in the graphs, such as separate rooms, or linear gradients in each room.

How do spectral convolutions measure against our 3 properties? First, they have n parameters (n being the size of the graph), if we discard channels for the sake of simplicity. This is still more than $k \times k$ for euclidean convolutions, which do not depend on the input size (although because of their small receptive field, one should have $O(\log n)$ euclidean convolutional layers interspersed with pooling). A way to alleviate this problem is to only consider a fixed number of eigenvectors of the Laplacian, typically the ones associated with the lowest eigenvalues. Besides, it is clear that as λ grows, the eigenvectors become less meaningful and more irregular. Second, the complexity of a forward is $O(nd)$ if one considers only d eigenvectors, against $O(nk^2)$ for euclidean convolutions on a $\sqrt{n} \times \sqrt{n}$ grid. However, spectral convolutions are not translation covariant: though we cannot define translation in general graphs, it exists in some way in our setting, and although eigenvectors sometimes feature the same motif at different locations, nothing enforces the same coefficient for the same motifs in our architecture.

Bruna et al. [2013] propose a way to enforce some sort of translation covariance (and simultaneously, locality, as in the fact that the output at some node will mostly depend on the input at close nodes), through smoothness in the spectral domain. Indeed, locality in the space domain is closely related to smoothness in the Fourier domain in $\mathbb{R}^m$. However, smoothness in the spectral domain of general graphs is not well-defined, as one needs a distance between frequencies. The straightforward way to define one is to consider an embedding of frequencies in $\mathbb{R}$ by their respective eigenvalue. Note that this does not correspond to the euclidean setting: the distance between two frequencies ω_1 and ω_2 is there defined as $\left| \|\omega_1\|_2^2 - \|\omega_2\|_2^2 \right| = |\langle \omega_1 - \omega_2, \omega_1 + \omega_2 \rangle| = \|\omega_1 - \omega_2\|_2 \|\omega_1 + \omega_2\|_2 |\cos\theta(\omega_1 - \omega_2, \omega_1 + \omega_2)|$, whereas it should be $\|\omega_1 - \omega_2\|_2 = \sqrt{\langle \omega_1 - \omega_2, \omega_1 - \omega_2 \rangle}$. Whatever the chosen metric, the authors propose to enforce smoothness by subsampling the embedding, define the coefficients of Λ in this smaller, subsampled space, and interpolate the remaining ones with cubic splines. We did not implement this idea as it seems rather arbitrary, and thus did not enforce any smoothness, and kept the d eigenvectors with the lowest eigenvalues.

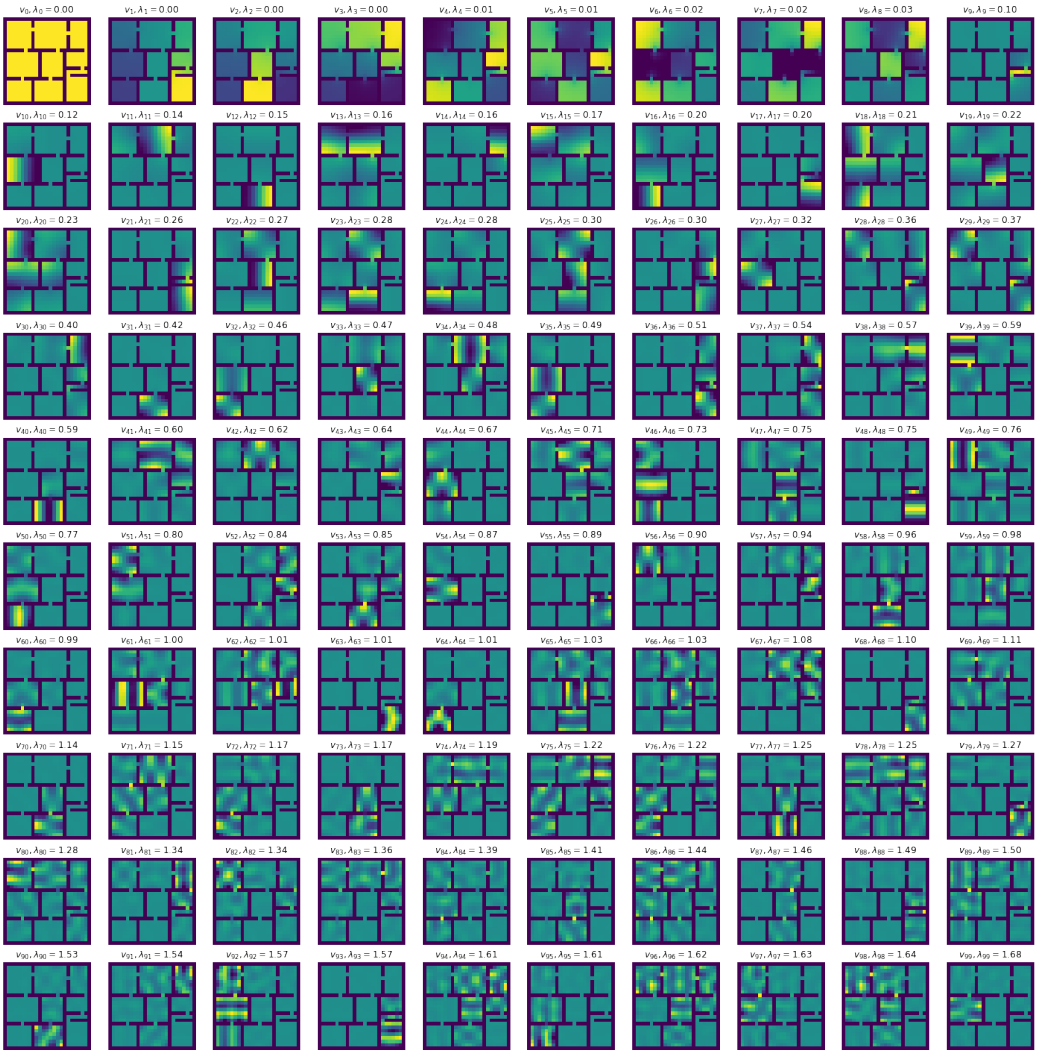

Figure 2: The first 100 (out of 444) eigenvectors of L (renormalized), ordered from lowest to highest eigenvalue

2.2 Heat Kernels

Another interesting construction one can make on graphs is heat kernels. They arise when solving heat diffusion equations:

$$\begin{cases} \frac{\partial f}{\partial t}(x,t) = -\Delta_x f(x,t) \\ f(x,0) = f_0(x) \end{cases} \tag{1}$$

which gives

$$f(x,t) = e^{-t\Delta} f_0(x) = \int f_0(y) \underbrace{\sum_i e^{-t\lambda_i} \phi_i(x)\phi_i(y)}_{h_t(x,y)} \, \mathrm{d}y \tag{2}$$

$h_t(x,\cdot)$ is the heat kernel at time t in x, and is the solution of equation (1) when $f_0 = \delta_x$. It can be interpreted as the heat flowing from x to y (or y to x, as it is symmetric) in time t. In $\mathbb{R}^m$, $h_t(x,y) = h_t(x-y) = \frac{1}{(4\pi t)^{\frac{m}{2}}} e^{-\frac{\|x-y\|_2^2}{4t}}$, a translation invariant gaussian. See figure 3 for some heat kernels in our graph.

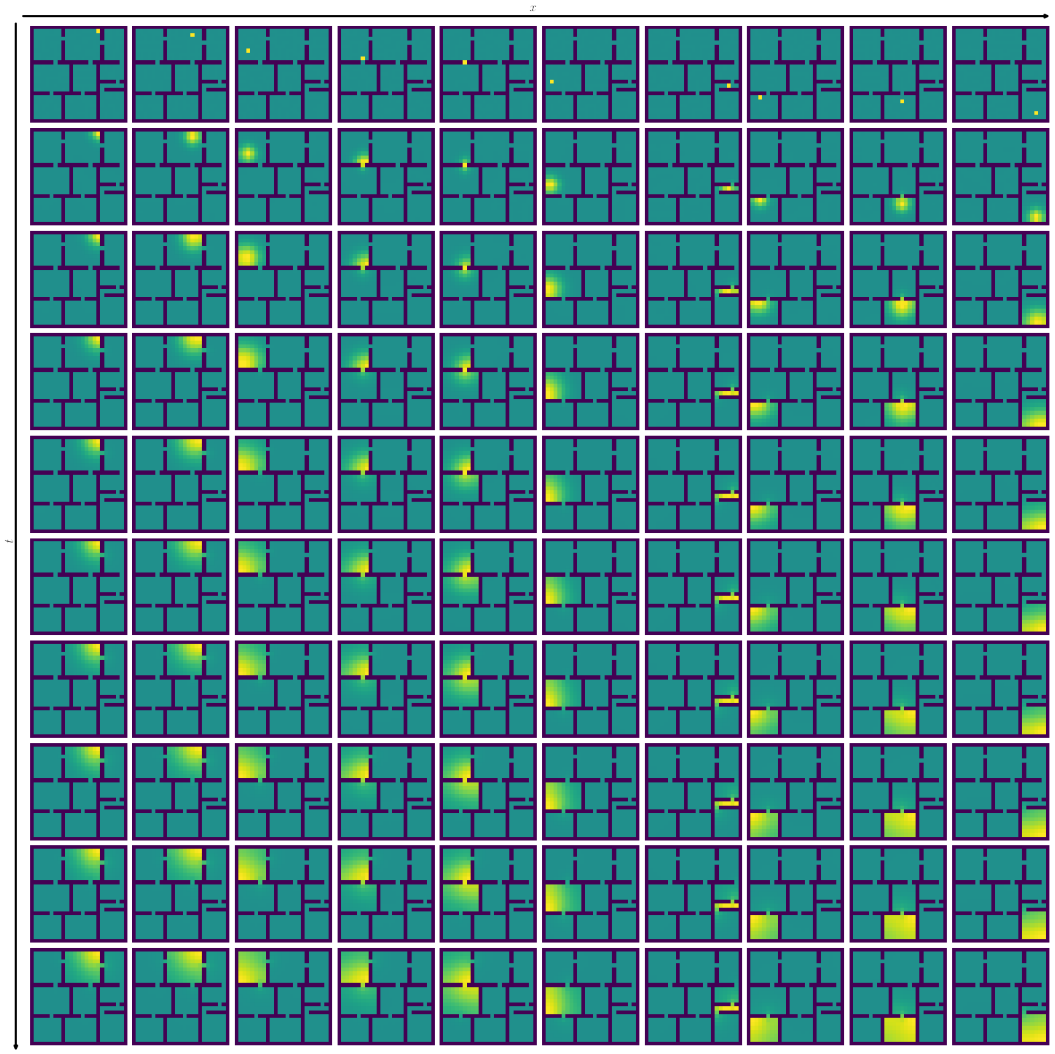

Figure 3: Heat kernels of L (renormalized)

Heat kernels can be used as a way to perform diffusion in the graph. The input is treated as the initial condition f_0 of equation (1), and the output is simply $f(\cdot, t)$ for some fixed t, which is akin to the kernel size of an euclidean convolution as it sets the receptive field. Heat kernels diffusion are fast to compute, if one code them in a sparse way, and have some sort of translation-covariance. They also have the added advantage over spectral eigenvectors that they are more stable to perturbations (according to Bronstein et al. [2017]) and uniquely defined. However, it is unclear how to use them to create parametric convolution layers suitable for learning.

A way around this problem is to parametrize the Laplacian itself, creating an anisotropic version of it [Boscaini et al., 2016]. The idea is that one has $L = -\mathrm{div}\nabla$, with $(\nabla f)_{i\to j} = f_j - f_i$ and $(\mathrm{div}F)_i = \sum_{i\to j} w_{i\to j} F_{i\to j}$. Then, one can define an anisotropic Laplacian $L_A = -\mathrm{div}A\nabla$ with A a matrix operating on edges of the graph. Since our edges have some structure here, as they are a subset of $\{1,\dots,n\} \times \{\leftarrow, \rightarrow, \uparrow, \downarrow\}$ (and even $\{1,\dots,n\} \times \{\leftrightarrow, \updownarrow\}$ if we consider them undirected), it is possible to consider A acting the same on each node, having thus a size of 4×4 or 2×2. This defines a diffusion layer parametrized by very few parameters (see figure 4 for examples of anisotropic heat kernels in our graph). This has several problems though:

- one has to perform an eigenvalue decomposition at each update of A to recompute the heat kernels,

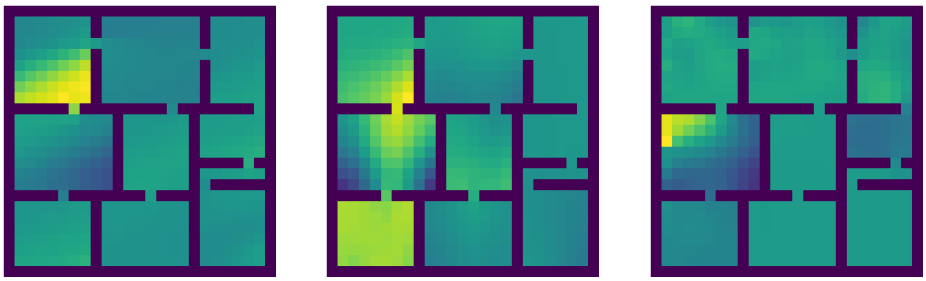

Figure 4: Examples of anisotropic heat kernels

- the Laplacian L_A is not necessarily symmetric nor non-negative, which yields complex eigenvectors,
- `torch.eig` is less numerically stable than its symmetric counterpart `torch.symeig`,
- in order to do optimization in A, one has to backpropagate gradients through the eigenvector decomposition, which sounds not very stable and is not implemented in pytorch.

Because of these drawbacks, we chose to keep the heat kernels as parameter-free diffusion layers.

2.3 Translations

Another angle to look at to generalize convolutions is simply their expression: $(f * g)(x) = \int f(y)g(y-x)\,\mathrm{d}y$. If one manages to define translations, then one can compute convolutions between two functions on the nodes of the graph. Shuman et al. [2013] propose to define them by a spectral convolution with a Dirac:

$$\tau_x f = \Phi\left(\Phi^T f \odot \Phi^T \delta_x\right) \tag{3}$$

where $\odot$ is the Hadamard/pointwise product $(A \odot B)_{i,j} = A_{i,j}B_{i,j}$.

Although generalizing translations to any graph sounds appealing, we chose not to use them in our architectures because we felt they did not correspond to intuition. Indeed, the neutral element of convolutions, defined as e such that $e * f = f\ \forall f$, is the vector with only ones in the spectral basis, $e = \Phi\mathbf{1} = \sum_i \phi_i$. In the euclidean case, one has $e = \delta_0$, with $\hat{e}(\omega) = 1$. Thus one expects e to be close to a Dirac. It is not the case at all (see figure 5). Besides, if we look at the translation of something as simple as a Dirac $\tau_y\delta_x$, then the result do not look like a Dirac for all values of x and y (see figure 6 for an example).

2.4 Others

This review is not meant to be comprehensive. There are other ways to define convolutions on general graphs, for instance based on polynomials of the Laplacian. Graph Neural Networks [Scarselli et al., 2009] are a generalization of such types of convolutions. We did not implement those for lack of time, as they looked less promising than other methods.

3 Architectures

3.1 Hourglass

Since our graph is naturally embedded in the 2D euclidean grid, we can consider classical architectures used in computer vision, such as for image classification. These typically are composed of blocks of several euclidean convolutions interspersed by a non-linearity, and pooling. This allows to gradually reduce the input down to 1×1 spatial size, effectively aggregating the information globally. They work very well, and require only $O(\log n)$ parameters, as each block divides the input size by a constant.

However, we require the network to predict one value per node in the graph, as to integrate better with the other architectures (see section 4 to see the meaning of the values). This can be achieved

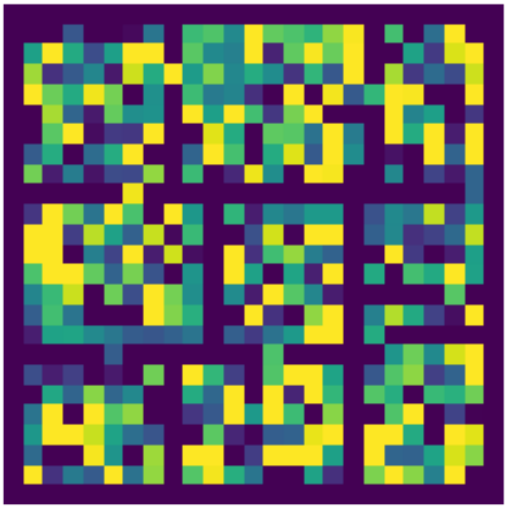

Figure 5: The neutral element of translation-based convolutions

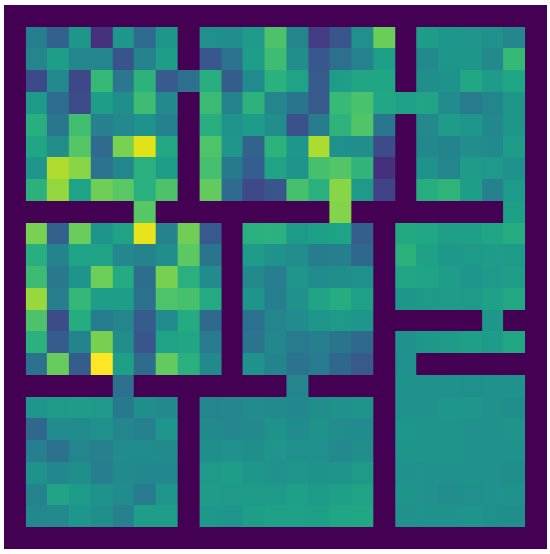

Figure 6: The translation of a Dirac

by using padded convolutions, in order to keep a fixed size. However, this would require a lot of convolutions to achieve a receptive field covering the whole grid ($O(\frac{\sqrt{n}}{k})$ layers, which means $O(\sqrt{n}k)$ parameters disregarding channels). Deep architectures are notoriously hard to train, and having a lot of parameters increase the risk of overfitting.

An efficient way of aggregating information is to use pooling. Since convolutions and pooling reduces the input size, one needs to use transposed convolutions and upsampling to go back to the original size, yielding an encoder-decoder architecture we call HOURGLASS (after Newell et al. [2016]). We also add skip connections to help the reconstruction, as some information is lost during pooling. With c intermediate channels and an input size of $\sqrt{n} \times \sqrt{n}$, the architecture has $O(c^2 \log n)$ parameters and the forward complexity is $O(c^2 n)$.

Because these architectures are still rather deep, and for the lack of time, we did not implement this architecture.

3.2 STANDARD

The STANDARD architecture is defined as a sequence of any kind of size-preserving convolutions, interspersed with non-linearities. We can still use regular euclidean convolutions here, as our graph is naturally embedded in a 2D grid, using half-padding. This allows to fix the number of layers l depending on the receptive field of the underlying convolutions. The architecture has $O(lc^2)$ parameters (assuming the convolutions used have $O(1)$ parameters), and the forward complexity is $O(lc^2 n)$ (assuming each convolution is $O(1)$ to compute). These assumptions hold for euclidean convolutions with a fixed kernel size and spectral convolutions with a fixed number of eigenvectors.

3.3 DECOUPLED

The DECOUPLED architecture is a sequence of l blocks, where each block is composed as a 1×1 convolution, a diffusion layer and a non-linearity. The diffusion layer is replicated along channels (each channel diffuses independently). The standard diffusion layer uses heat kernels at a fixed time t, but we explore several other diffusion layers in our experiments. The number of parameters is $O(lc^2)$ if the diffusion used has no parameters, and the complexity of a forward is $O(lc^2 n)$ assuming the diffusion can be done in $O(1)$ time, which is true for heat kernels diffusion if t is fixed and heat kernels are stored in a sparse way (the support of h_t in $\mathbb{R}^m$ is of size $O(t^{m/2})$). The decoupling in channels and spatial passes allow for very few parameters, without sacrificing the receptive field thanks to diffusion layers.

4 Experiments

4.1 Training

In all our experiments, we set $\gamma = 0.98$ and train the networks on simultaneous infinite episodes. We are not interested in the starting period, where there is no dust spawned yet, but we look at the stationary regime. All our code is available at `https://github.com/Kegnarok/GraphRL`.

We mainly experimented with two reinforcement learning algorithms, REINFORCE and DEEP Q LEARNING (DQN). We were not able to achieve any significant result with REINFORCE and we also were not able to design a satisfying network that would use the graph structure for it. Therefore, we will focus on our DQN results.

DQN usually requires a network to approximate $Q(s, a)$. We approximate this value in a slightly original manner. A network maps a global observation of the grid (walls, nearby dusts and duration since last visit) to a value per cell. Then $Q(s, a)$ is the value of the neighbour cell of our current position in direction a. This seems well adapted to our learning needs and is convenient for our various architectures as both spectral and conventional convolutions can achieve such mappings. Figure 7 is an example of resulting values, the rooms on the right are less valued as they were visited recently. Note that the robot is unaware of the presence of dusts in cell further than $k = 6$ hops.

Our training is slightly different from usual DQN, instead of doing multiple steps in between policy updates and using a replay buffer, we take advantage of the batching capabilities of our environment implementation to use a large batch size $(4,096)$ which makes policy updates every step without

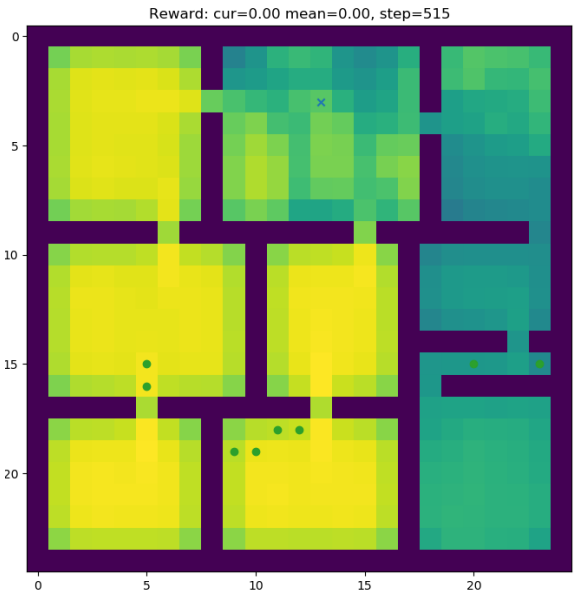

Figure 7: Values computed by policy

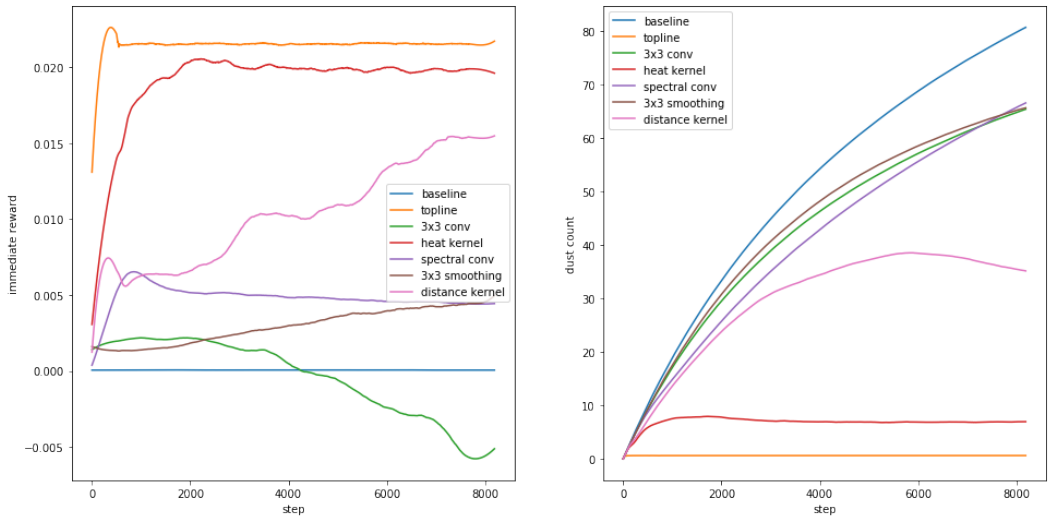

Figure 8: Mean reward and dust counts with trained policies

replay buffer stable enough. Moreover, with relatively small networks, forwards are much more efficient with a large batch size.

Exploration is handled with an off-policy ε-greedy approach, ε starting at 1 and decaying by 0.9998 at each step. Policies were trained for $50,000$ steps.

4.2 Testing

Each curve in figure 8 represents the evaluation of an architecture once the associated policy is trained. The metrics are averages over $16,384$ runs of immediate reward (left) and dust count (right), showing their evolution over time steps.

All architectures uses $\tanh$ as non-linearities, 8 intermediate channels and 2 layers.

- *topline* is a fixed policy which observes the whole graph and goes to the nearest dust at each step.
- *baseline* is a fixed policy which remains on the same two cells (as it is not possible to not move).
- 3×3 *conv* is a STANDARD architecture with euclidian convolutions with 3×3 kernel.
- *heat kernel* is a DECOUPLED architecture with heat kernel diffusion.
- *spectral conv* is a STANDARD architecture with spectral convolutions on the 10 first eigenvectors.
- 3×3 *smoothing* is a DECOUPLED architecture with 10 passes of neighbour smoothing as diffusion.
- *distance kernel* is a DECOUPLED architecture with distance kernel diffusion: $e^{-\mathrm{dist}}$ with dist being the graph distance.

3×3 *smoothing* is about 20 times slower to compute than other methods but we suspect it could be made similar by precomputing a kernel. Other architectures are very fast to compute (about 25 forwards per second with $16,384$-sized batches).

Our experiments suggest that with the right diffusion, DECOUPLED can lead to satisfying results. Other approaches were mostly unsuccessful. We suspect classical architecture could be trained in this setting but would have far more parameters and training would be a lot harder.

Conclusion

In this paper, we proposed a new reinforcement learning environment designed to measure progress in our ability to exploit graph structure in deep neural network architecture. We described several extensions to this environment to make the problem more difficult, and provides an efficient GPU batched implementation. We reviewed the majority of attempts to define general convolutions on graphs, discussing their advantages and drawbacks, and proposed several modular neural network architectures making use of those convolutions. We then compared them experimentally and showed that our novel DECOUPLED architecture outperforms classical architectures.

In the future, it would be interesting to study: generalization to unseen grids, learning dust probabilities to survey some areas more than others and adapting this to more realistic environments where programming a good policy would be hard.

References

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, L Robert Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.

Davide Boscaini, Jonathan Masci, Emanuele Rodolà, and Michael Bronstein. Learning shape correspondence with anisotropic convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 3189–3197, 2016.

David I. Shuman, Benjamin Ricaud, and Pierre Vandergheynst. Vertex-frequency analysis on graphs. *CoRR*, abs/1307.5708, 2013.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, Jan 2009. ISSN 1045-9227. doi: 10.1109/TNN.2008.2005605.

Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation. In *European Conference on Computer Vision*, pages 483–499. Springer, 2016.