Supplement for Graph Element Networks: adaptive, structured computation and memory

Ferran Alet [1] **Adarsh K. Jeewajee** [1] **Maria Bauza** [2]
Alberto Rodriguez [2] **Tomás Lozano-Pérez** [1] **Leslie Pack Kaelbling** [1]

A. Experimental details

Code can be found at https://github.com/FerranAlet/graph_element_networks. For the full paper with supplementary materials, please see https://arxiv.org/abs/1904.09019.

A.1. Details common to all experiments

Although the experiments are pretty diverse, most of the code and decisions are shared between them. We first talk about the details common to all experiments.

We explored two representation functions. The first assumes that nodes are placed in a regular grid and uses the latent values at the four nodes of the grid cell containing a point to compute its latent value. So the representation function $r(x, y) = v$; letting (i_1, i_2, i_3, i_4) be the node indices of the grid cell containing (x, y), and letting (x_{i_j}, y_{i_j}) be the location of node i_j, then $v_{i_j} = |x - x_{i_j}| \cdot |y - y_{i_j}|$ and $v_\ell = 0$ for all other indices ℓ. This is a generalization of barycentric coordinates to the rectangular case. The second is a "soft" nearest neighbor representation function $r(x, y) = \text{softmax}(D((x, y)))$ where $D((x, y))$ is a vector whose $i - th$ entry is $-\text{dist}((x, y), n_i)$. The grid-based representation has the advantage of being sparse while the soft nearest-neighbor representation can be applied independently of the placement or topology of the nodes. Since they did not have significantly different results in our experiments, we decided to use the "soft" nearest neighbor in all experiments of the Poisson experiments, since it also works for non-grid meshes and used the barycentric representation for the pushing and scene experiments, to show it can also work with sparse representations.

For our implementation we used PyTorch (?), we used ReLU non-linearities, trained with Adam (?) with learning rate 3e-3.

[1]CSAIL - MIT, Cambridge, MA, USA [2]Mechanical Engineering - MIT, Cambridge, MA, USA. Correspondence to: Ferran Alet <alet@mit.edu>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

We tried both training independent networks for each mesh and training a single model for all meshes, with no significant difference between both. We only put the results of a single model, since it reinforces our message of a single model being able to trade off computation and accuracy. We note that, as the density of the mesh increases, the number of computations and depth for GENs increases, but since we are reusing the same weights, the number of weights is always smaller than those of the baseline. The depth of the baseline is similar to a GEN of 4x4 nodes and we found that adding extra depth did not help much.

A.2. Poisson experiments on a square

The GNN within the GEN is specified by: latent space $\mathbb{L} = \mathbb{R}^{32}$; a set of k^2 nodes placed on a uniformly-spaced grid; edges generated by the Delaunay triangulation of the nodes; encoders e_1, e_2 and decoder d_1 are two-layer neural networks (with a hidden state of 48 and 32 units respectively); the GNN message size is 16, and both edge and node modules have one hidden layer of size 48 and 64 respectively; and the diameter of the graph is $T = 2(k - 1)$.

The test set results were averaged over 256 sample points per scenario, for 32 scenarios of 50 houses. We trained the baseline for 3000 epochs and GENs for 3000/6=500 epochs (since we trained on 6 mesh configurations, from 2x2 to 7x7).

A.3. Poisson experiments on a sphere

For the spheres, we generated our own dataset of pairs of Laplacians and PDE solutions. Similar to the square house experiments, we had 250 houses with 32 scenarios each; trained on 200 and tested on 50. We had 128 inputs and 128 queries. In contrast to the square experiments, since we did not have easy access to a PDE solver for a sphere, we instead specified the solution of the PDE and numerically computed the Laplacian. The solution of the PDE was of the form:

$$f(x) = \sum_{i=1}^{8} k_{i,s} \cdot (\boldsymbol{x} \cdot \boldsymbol{v}_i)^3$$

with k_i a random gaussian scalar, which also varied with scenario s and vector directions $\boldsymbol{v}_i$ which played the role of heater/cooler positions. Moreover, since the dot product on a sphere is symmetric and we raise it to the cube (an odd function), we know:

$$\int_{x\in\mathcal{S}^2}\left[(x\cdot\boldsymbol{v}_i)^3\right]=0$$

this in turn forces:

$$\int_{x\in\mathcal{S}^2} f(x)=\sum_{i=1}^{8}k_{i,s}\int_{x\in\mathcal{S}^2}\left[(x\cdot\boldsymbol{v}_i)^3\right]=\sum_{i=1}^{8}0=0$$

This is important because since the sphere has no boundary conditions, solutions to the Poisson equation are only defined up to a constant. We thus define that the solution is the *unique* function which satisfies the equation *and* has integral 0.

Computing the Laplacian on a sphere is not trivial to do analytically since the partial derivatives have to be taken on the sphere, not on Cartesian space. Since the Laplacian is related to the difference between $f(p)$ and the mean value of $f(x)$ in the epsilon ball around p, another interpretation is to say that the relevant neighbors are only those that lie on the sphere. We compute the Laplacian numerically by computing the tangent plane at $\boldsymbol{x}$ by using Gram-Schmidt to complete an ortonormal basis which includes $\boldsymbol{x}$: $\{\boldsymbol{x},\boldsymbol{a},\boldsymbol{b}\}$. Then, we numerically approximate both second derivatives by taking ϵ steps in $\boldsymbol{a},\boldsymbol{b}$, projecting back to the sphere and evaluating there:

$$\nabla^2 f(\boldsymbol{x})\approx\frac{1}{4\epsilon^2}\Big(f(\frac{\boldsymbol{x}+\epsilon\boldsymbol{a}}{|\boldsymbol{x}+\epsilon\boldsymbol{a}|})+f(\frac{\boldsymbol{x}-\epsilon\boldsymbol{a}}{|\boldsymbol{x}-\epsilon\boldsymbol{a}|})+ f(\frac{\boldsymbol{x}+\epsilon\boldsymbol{b}}{|\boldsymbol{x}+\epsilon\boldsymbol{b}|})+f(\frac{\boldsymbol{x}-\epsilon\boldsymbol{b}}{|\boldsymbol{x}-\epsilon\boldsymbol{b}|})-4f(\boldsymbol{x})\Big)$$

We used $\epsilon=3e-5$, but any $\epsilon\in[3e-6,3e-4]$ gave essentially the same results.

To make it as similar as possible to the experiments on a square, node positions also followed a grid in spherical coordinates:

$$x=\sin\theta\cos\phi$$
$$y=\sin\theta\sin\phi$$
$$z=\cos\theta$$

The mesh of order k $\theta\in\{0,\pi/(k-1),2\pi/(k-1),\dots,\pi\}$ and $\phi\in\{0,\pi/(k-1),2\pi/(k-1),\dots,2\pi\}$ and then removing points that were duplicates in Cartesian space; resulting in k longitudes of $1,2k-1,2k-1,\dots,2k-1,1$ nodes. Note that, for the sphere, the soft nearest neighbor distance function is not Euclidean, but the distance on the sphere surface (the arccosine of the dot product, since we are on the unit sphere). We trained both the baseline for 10000 epochs and GENs for and GENs for 10000/6=1666 (6 mesh configurations, with sizes randing from 2 to 7).

A.4. Optimizing node positions

We trained the GEN with optimizable positions for 10000/12=833 epochs (we trained with 12 mesh configurations). The optimization of the node positions used learning rate 3e-4, while the weights still used learning rate 3e-3. This came out of an informal search where we wanted nodes positions to not prematurely converge, yet also be able to go to any possible configuration. Smarter ways to perform this optimization (potentially including non-local optimization) are an interesting avenue for future work.

For each mesh size we initialize two random positions, which tend to converge to similar but different final positions, as seen in figure 2. The initial positions of the nodes were generated using the ghalton python library (**?**), which uses Generalized Halton Sequences to create points that are random but well spread out along a region. This is because uniformly random points tend to get clustered by pure chance.

The connectivity was the Delaunay triangulation, which has many desirable properties for our purposes: it has low graph diameter (**?**), it is very stable (small perturbations tend to produce little or no changes) and changes are local (moving a point can only affect edges nearby). Moreover it can be computed efficiently and it is the dual of Voronoi diagrams, which map each point to its nearest neighbor in space, thus being linked to a good representation function. After each gradient step, we recomputed the connectivity. Note that these changes are non-differentiable; we considered smoothing the connectivity by weighting each edge proportionally to the angle it represents in the Voronoi diagram, but performance was great without it and we preferred to keep it simple. Pure back-propagation would move some points outside the $[0,1]^2$ region; therefore, after back-propagation we clamp points back inside the region.

A.5. Scene representation experiments

Our GEN model treats the (x,y) coordinates of the camera pose as the space $\mathbb{X}$ and the image and camera roll and pitch angles as the input space $\mathbb{I}$. Because we have an image classification task, it is possible to interpret the output space $\mathbb{O}$ as an output distribution over the candidate images. This is not entirely correct, however, because the set of candidate images varies during training, so this is not precisely a fixed output space. The encoder of our GEN model is the same as the one used by the original NP: it uses a CNN to map input images and camera poses into a 256-dimensional latent space. Once an (image, pose) pair is encoded, it is stored into the latent state of the nodes in the GEN using the representation function. Given a query camera pose, the latent representation z is determined based on the (x,y) coordinates. This $z\in\mathbb{L}$ is concatenated with the camera pose and decoded to an embedding space $\mathbb{Z}'$. The candidate images are also mapped into $\mathbb{Z}'$, allowing us to compute a vector of squared Euclidean distances betweeen the query output and the candidate images in $\mathbb{Z}'$. This vector is fed through a softmax to generate the final vector of output values. For the GEN structure, the edge set is composed of the all the bi-directional edges between pairs of adjacent nodes on the grid described in 4.3. The rectangular generalization of barycentric coordinates, described in A.1 was used to compute the latent value of any query position (x, y). When obtaining the encoding of input images (with their camera pose coordinates) with the GEN, the GNN messages used have 256 dimensions, and both the node and edge neural networks are 2-layer feedforward networks with a hidden layer of 512 units. To compensate for the these GNN computations, the NP capacity is increased by always concatenating 32 copies of input query coordinates to their corresponding embeddings. For the GEN we only provide 8 copies, and we experimentally observed that these copies slightly benefit the GEN and strongly benefit the neural process. We believe that, without copies, the signal from the query pose is drowned by the embedding, which has many more dimensions. We did not notice further benefits beyond 32 copies, which makes sense since $32 \cdot 7$ is already of the same order of magnitude as 512. For both the GEN and baseline experiments the encodings are post-processed through feed-forward networks, which have depth 2 and width 256 for the GEN and depth 4 and width 512 for the NP, respectively.

When running the scene experiments for many epochs, without batch normalization (not added in other experiments) the performance of both GENs and the baseline smoothly increased until 80% accuracy on 1x1, but then weights exploded and performance plummeted for both models. Batch normalization solved this issue for both models.

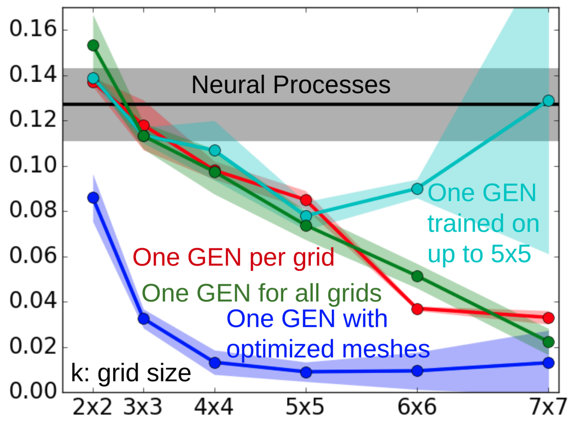

Figure 1. Comparison betweeen original GENs and two other reasonable approaches. We see that a GEN using the same weights for all mesh sizes (green) can do as well as using a custom GEN for each mesh size. However, the same weights do not generalize to much bigger meshes (cyan, trained only on 2x2..5x5). Contrary to other GNN-based systems that generalize to bigger input sizes than those seen at test time, we do increase the number of propagation steps and graph size increases quadratically; both things can make propagation unstable. This is an interesting point to address in future work.

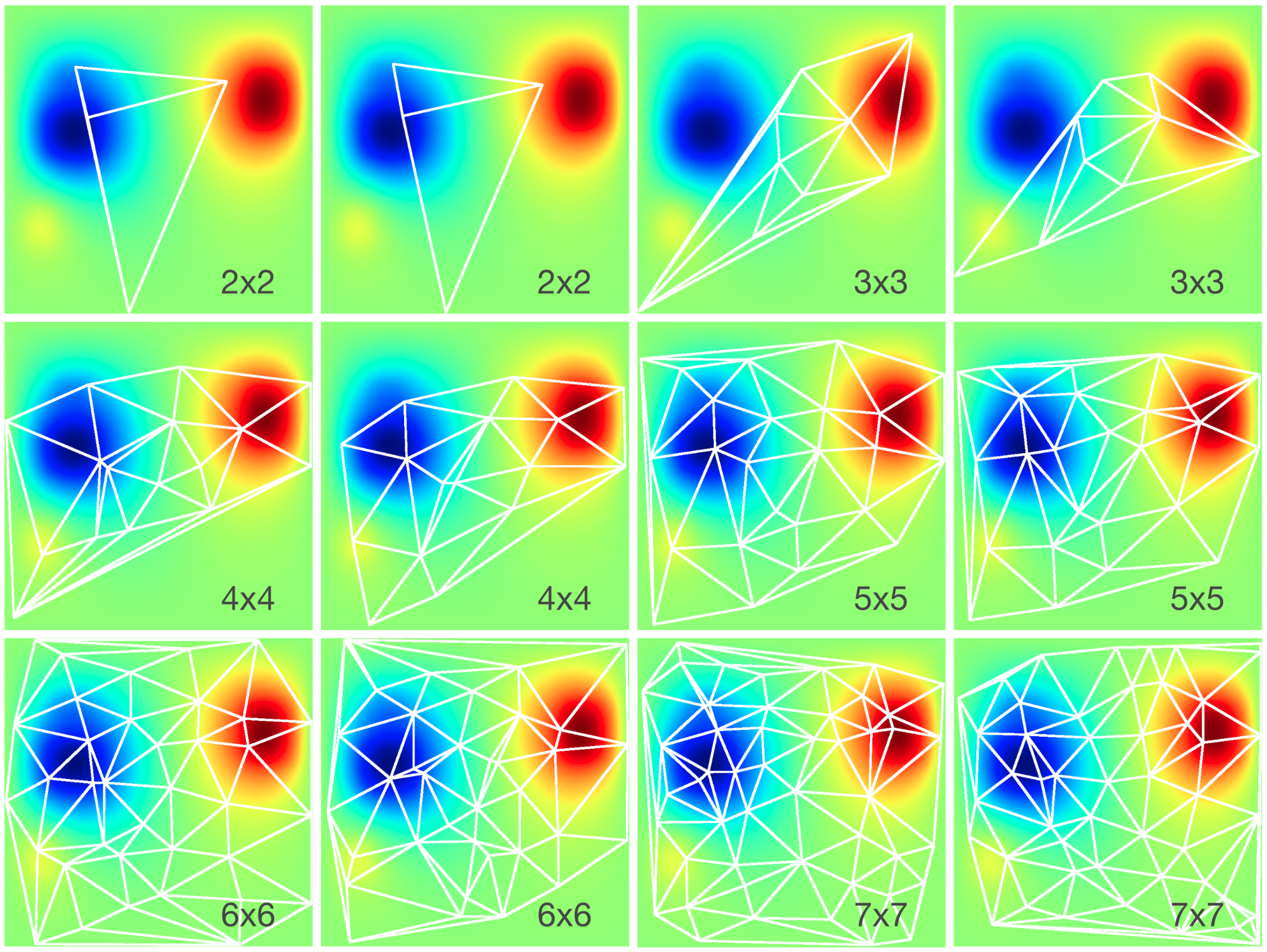

Figure 2. **One room with all the optimized meshes**. We notice that each size makes the most of its amount of nodes by focusing on the most complex parts of the space. For each size, the pair of meshes resembles each other, but (except for the 2x2) are qualitatively different.

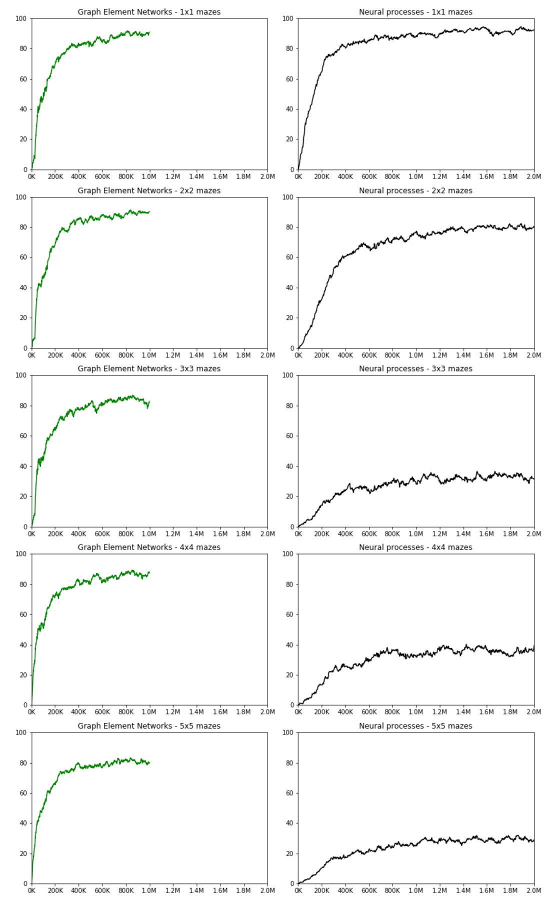

Figure 3. Test accuracy plotted against number of mini-batch gradient descent steps, for Graph Element Networks experiments (left) and the baseline Neural Processes experiments (right). The graphs from top to bottom show performance on supermazes of increasing sizes (1x1 to 5x5).