1 Background

Here, we are attempting to train an ML agent to output both aggregates and interpolation for an AMG solver given the matrix $\boldsymbol{A}$ for a discretized PDE. We will be primarily be looking at the finite element discretization of the isotropic and anisotropic diffusion problems, i.e. PDEs of the form

$$-\nabla \cdot (\boldsymbol{D} \nabla u) = 0, \tag{1}$$

$$\boldsymbol{D} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & \\ & \varepsilon \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^T, \tag{2}$$

for some arbitrary θ and ε. In the isotropic case, $\theta = 0$ and $\varepsilon = 1$.

In both the isotropic and anisotropic cases, unstructured datasets of the following are generated:

1. A training set of 1000 unstructured grids
2. A testing set of 300 unstructured grids

Each of these grid meshes are fairly small, overall containing between around 15 and 400 points. To create a mesh, random points are generated in $[0, 1]^2$, a convex hull constructed, then meshed with gmsh. The training and testing sets are intentionally limited to a small number of grids to reduce training complexity.

2 Generating Aggregates and Interpolation

To compute the aggregates and final interpolation operator, we concurrently train three graph networks to give different outputs:

1. Θ_{agg}, outputting aggregate centers
2. Θ_{soc}, outputting a strength-of-connection matrix for Bellman-Ford
3. Θ_P, outputting a smoothing operator

The algorithm for computing the interpolation operator is roughly sketched below:

1. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be the system we are trying to solve and $\alpha \in (0, 1]$ be some parameter that determines the ratio of aggregates to vertices, i.e. we will be roughly coarsening the graph by $1/\alpha$. Define $k := \lceil \alpha n \rceil$, the number of aggregates we will be outputting.
2. Convolve the graph of $\boldsymbol{A}$ with Θ_{soc} to obtain $\boldsymbol{C}$, a matrix of weights that will be used for Bellman-Ford.
3. Convolve the graph of $\boldsymbol{A}$ concatenated with $\boldsymbol{C}$ using Θ_{agg} to obtain a new set of node values, restarting multiple times and inserting the tenative cluster centers as a node feature. We will use the output node values as a *scoring*. Define the aggregate centers as the indices of the largest k node scores.
4. Run Bellman-Ford on the graph with these aggregate centers to obtain a tentative aggregation operator, Agg.
5. Now, again convolve the graph of $\boldsymbol{A}$ but with Θ_{Interp} (with aggregate information) to obtain the aggregate smoother $\hat{\boldsymbol{P}}$. Form $\boldsymbol{P} := \hat{\boldsymbol{P}}\text{Agg}$.

These steps are formalized in Alg 1.

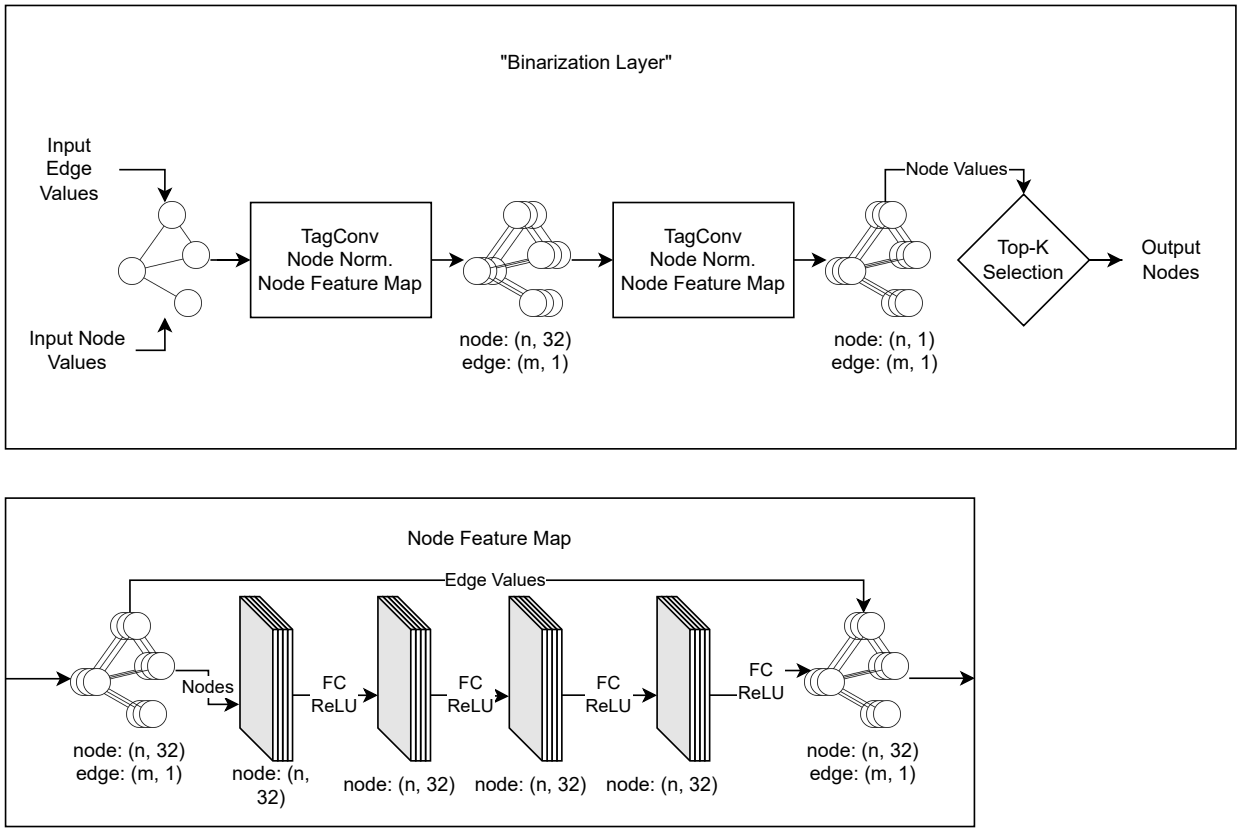


Figure 1: Intermediate binarization layer and node feature map that is used by the aggregate-picking network. This consists of 2 intermediate TAGConv convolutions, interspersed with fully connected layers on the node values. Each TAGConv layer convolves with a three-hop radius around each node. Afterwards, a binary vector assigning 1 to the $k := \lceil \alpha n \rceil$ largest nodes is returned.

3 Genetic Training

Training of both networks is done at the same time with a custom-written genetic algorithm that takes advantage of the easily parallelizable fitness calculation. A genetic algorithm is used because it does not require any gradient information, which benefits us because the algorithm used to output the aggregates is not easily differentiable.

A basic overview of the training algorithm is that the method is seeded with some number of randomly generated networks. A subset of the best performing (most fit) networks are selected and *bred* with one another (crossing weights/traits) and *mutations* inserted (random perturbations to weights) to create another population of networks. This is then repeated for many *generations* until a set of hopefully trained networks is obtained, from which we can pick the best fit as our final network.

3.1 Folded Crossover

So-called "folded crossover", as explored in [1], is a genetic crossover technique that groups together consistent regions of the population's chromosomes. When training neural networks, it is entirely possible that specific modules (layers) can contain several hundred or even several thousand weight entries, meaning traditional crossover that naïvely slice arbitrary points of the weight space can partially replace weight values for modules and produce suboptimal offspring.

By "folding" regions of the weight space, we ensure that whole modules are copied to offspring. We further modify the crossover procedure by allowing the two children of a pair of parents to randomly pick which parent they receive each module from. This is an additional level of flexibility compared to, e.g., single-point

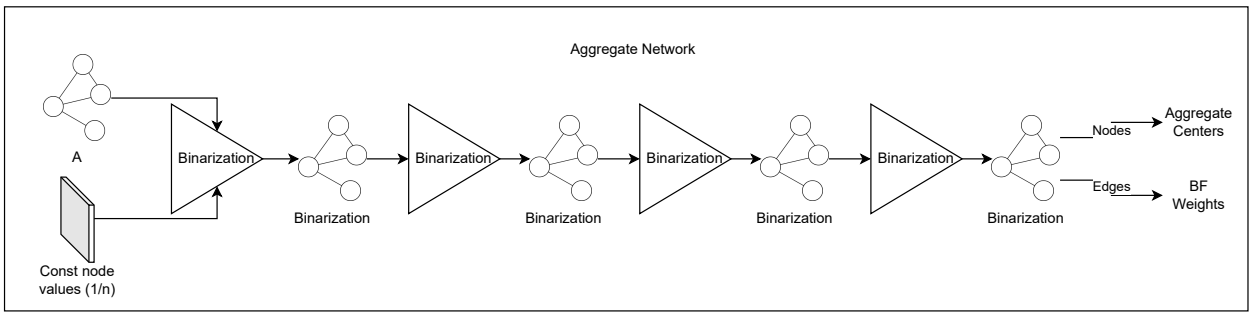


Figure 2: Architecture for the aggregate-picking network. This consists of 4 binarization layers (Fig 1).

Algorithm 1 Outputting Aggregates and Interpolation

procedure FULLAGGNET($\boldsymbol{A} \in \mathbb{R}^{n \times n}, \alpha$)
  $k := \lceil \alpha n \rceil$ ▷ Number of aggregates to choose
  $\boldsymbol{C} := \Theta_{\text{soc}}(\boldsymbol{A})$ ▷ Obtain strength-of-connection matrix

  $\boldsymbol{s} := \Theta_{\text{agg}}(\boldsymbol{A}, \boldsymbol{C}, k)$ ▷ Get aggregate scores for each node
  $\boldsymbol{r} :=$ TOP-K$(\boldsymbol{s}, k)$ ▷ Get indices of k largest nodes to be centers
  Agg := BELLMAN-FORD$(\boldsymbol{C}, \boldsymbol{r})$ ▷ Get tentative aggregate assignment matrix

  $\hat{\boldsymbol{P}} := \Theta_{\text{Interp}}(\boldsymbol{A}, \text{Agg})$ ▷ Aggregate smoother
  return $\hat{\boldsymbol{P}}$Agg ▷ Smooth aggregate assignment
end procedure

or two-point crossover which divide the weight-space into two or three regions, respectively.

Initial testing of the folded crossover provides subjective speedup to the rate at which networks learn, but I haven't really timed it or anything.

3.2 Stochastic Batching

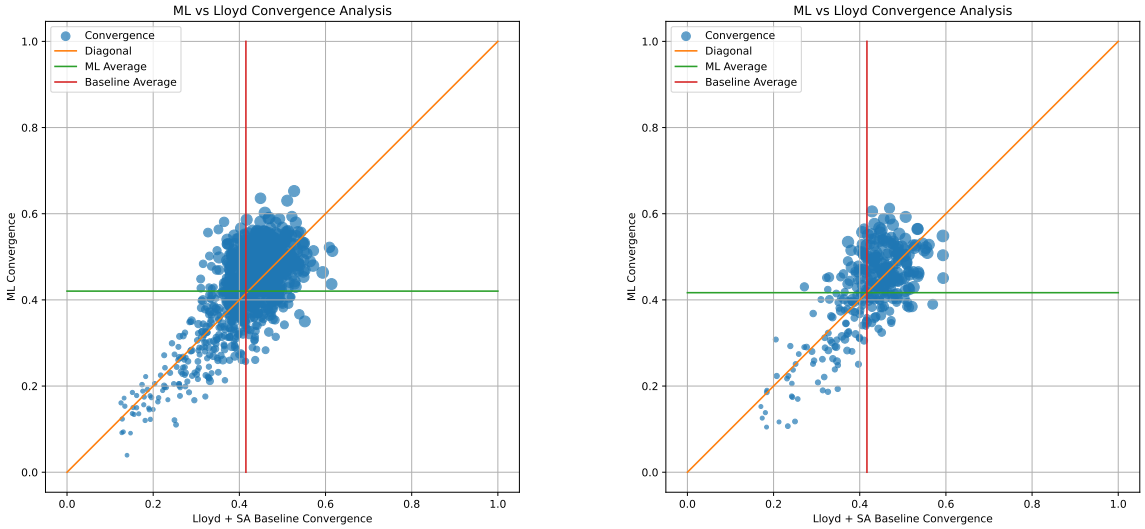
As mentioned previously, the most expensive part of training the networks is the computation of the fitness values, as we must compute an average convergence rate over every mesh in our training set, for every network in our population. Reducing the number of meshes used at each generation can drastically speed-up the training routine, at perhaps a reduction in training effectivity. This also eliminates any notion of caching fitness values for regions of the population that do not change between generations, as we have a different batch each generation and must totally re-compute fitness to have any sort of reasonable comparison.

In practice, this gives a healthy speed-up for minibatch sizes of only 16. I had to decrease the maximum perturbation during mutation to $|m| = 0.1$ to get it to do anything sensible, but perhaps it was too high in the first place?

3.3 Greedy Selection

I've experimentally found that using steady-state selection will train the network up to some point, then completely fail to make any more progress. At this point in the training, I've switched to what I call a greedy selection, where the population at generation $i + 1$ will totally consist of the one fittest individual from i. Then, these will be mutated as usual.

I kind of think of this as a sort of greedy approximation to gradient descent: we have one network and at every step we attempt to take the most optimal step towards a better network.



(a) Training convergence

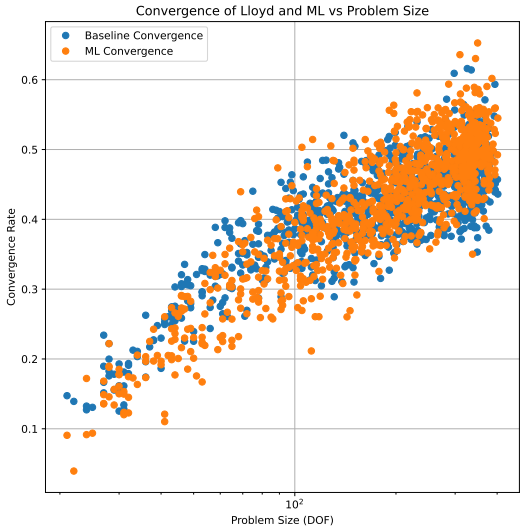
(b) Testing convergence

Figure 3: Convergence data for the ML AMG method vs a baseline Lloyd and Jacobi SA method. Values below the diagonal indicate a better convergence for the ML. Markers are scaled by problem size.

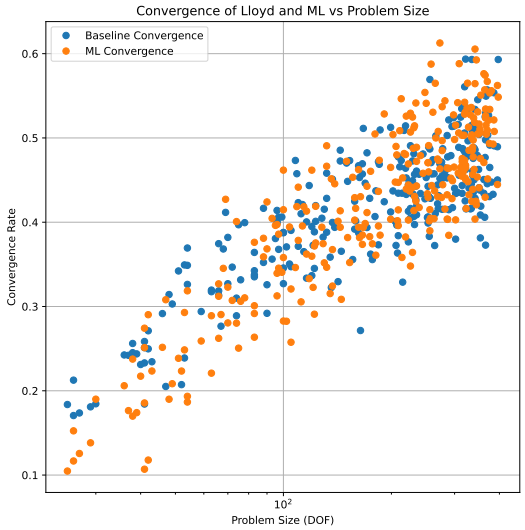
4 Results

References

[1] P. Esfahanian and M. Akhavan, *GACNN: training deep convolutional neural networks with genetic algorithm*, CoRR, abs/1909.13354 (2019).



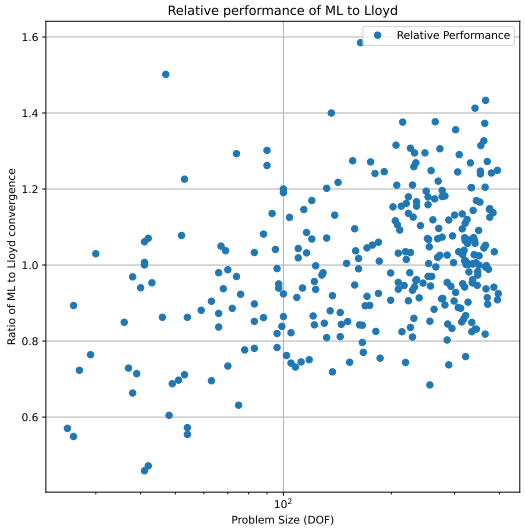
(a) Training convergence



(b) Testing convergence

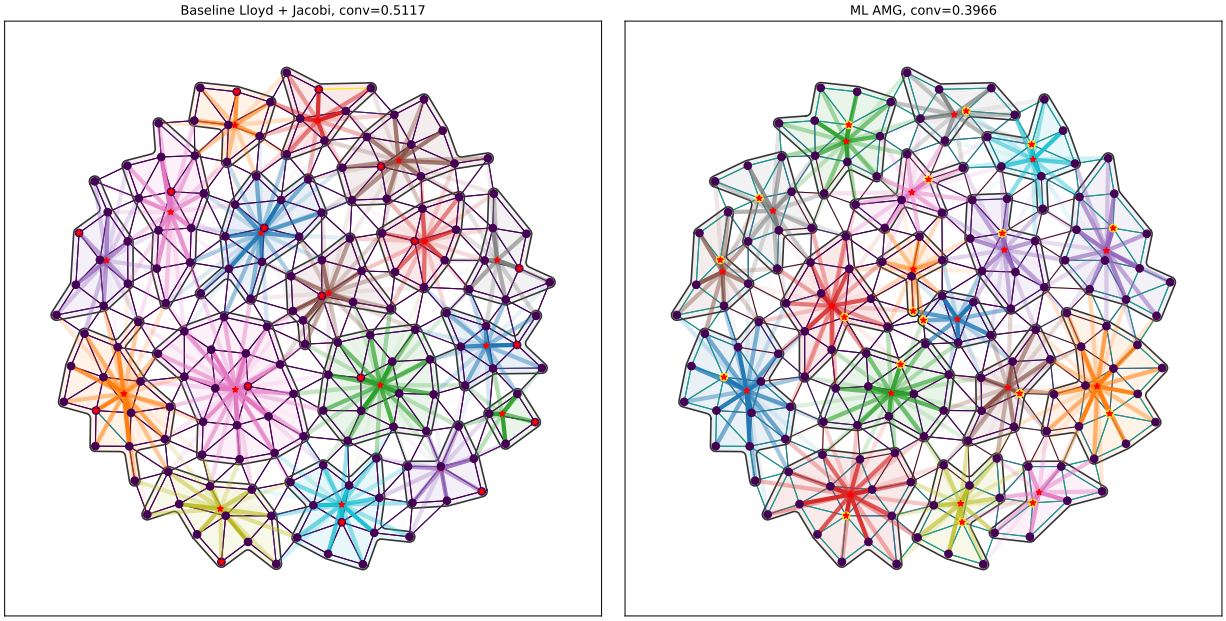
Figure 4: Convergence data for the two methods plotted against problem size (DOF). The ML method seems biased towards better performance on the smaller problems, and does not do as well on the larger problems.

(a) Relative training performance



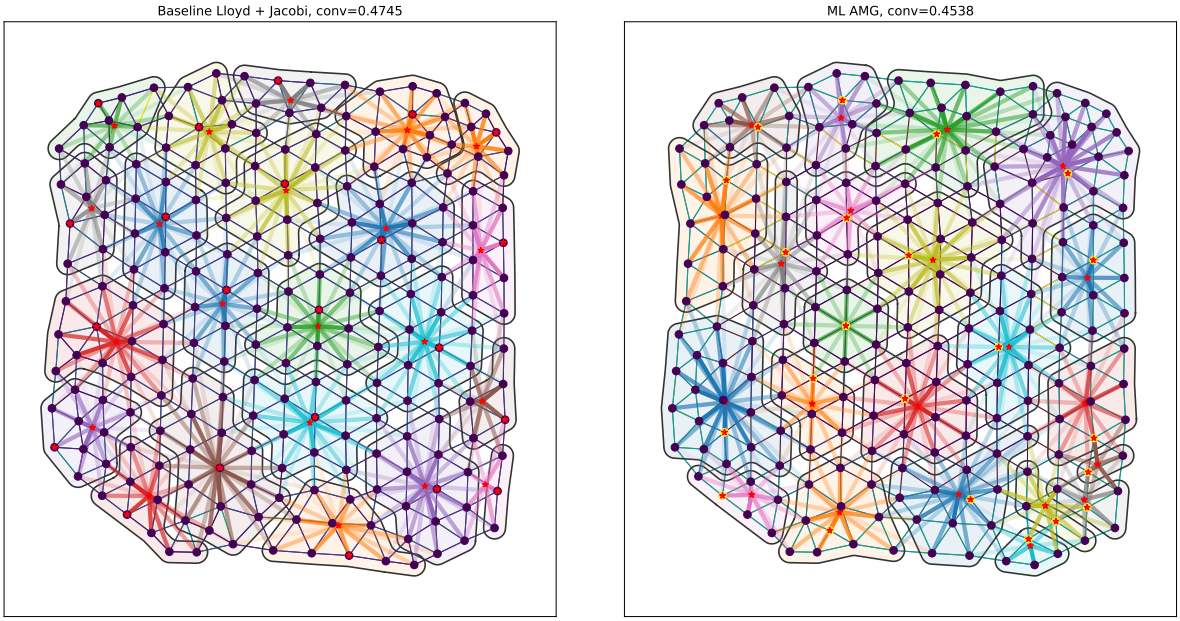
(b) Relative testing performance

Figure 5: Relative performance of the ML to the Lloyd method, plotted against problem size. Relative performance is obtained by dividing the ML convergence by the Lloyd convergence for each problem. Values below 1 indicate better ML performance, while values above 1 indicate better baseline performance.



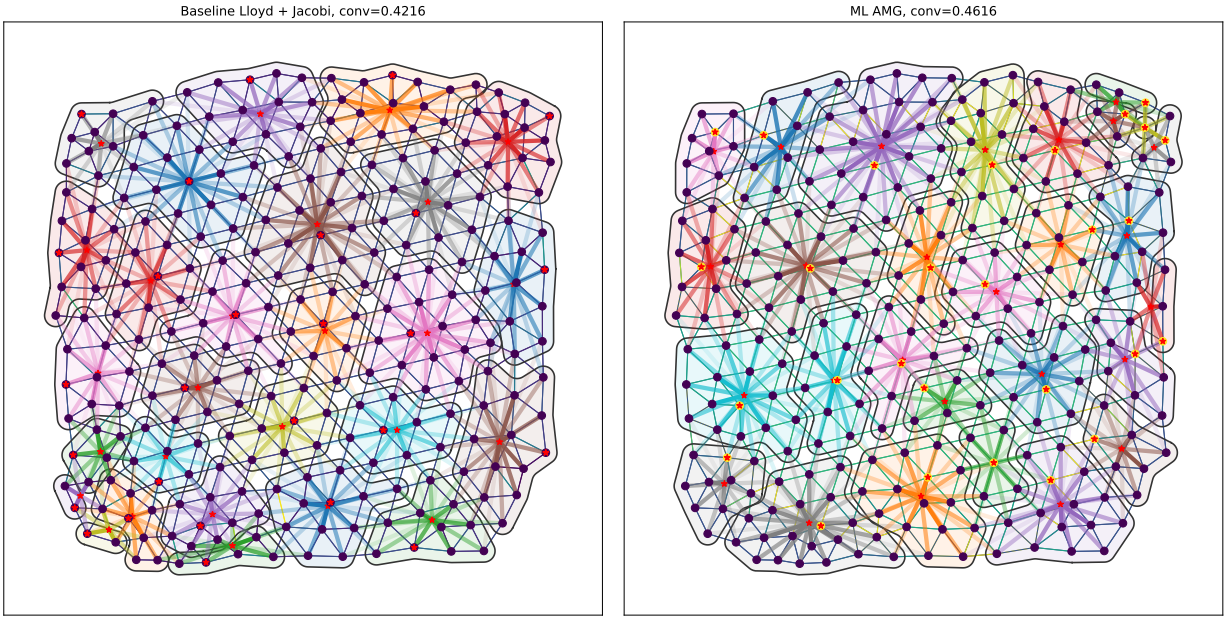
(a) Lloyd + Jacobi (b) ML

Figure 6: Aggregate and interpolation data for a circular unstructured mesh. This particular mesh has 173 DoF.



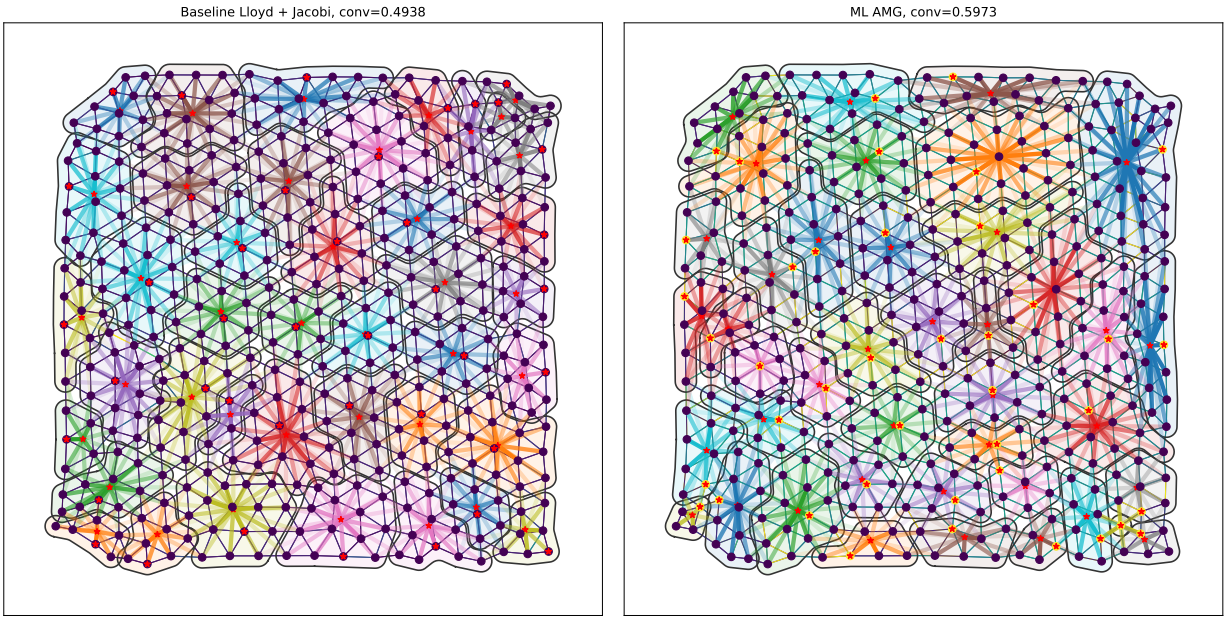
(a) Lloyd + Jacobi (b) ML

Figure 7: Aggregate and interpolation data for a random unstructured mesh. This particular mesh has 220 DoF.



(a) Lloyd + Jacobi (b) ML

Figure 8: Aggregate and interpolation data for a random unstructured mesh. This particular mesh has 273 DoF.



(a) Lloyd + Jacobi (b) ML

Figure 9: Aggregate and interpolation data for the largest mesh in the *training set*. This particular mesh has 410 DoF.

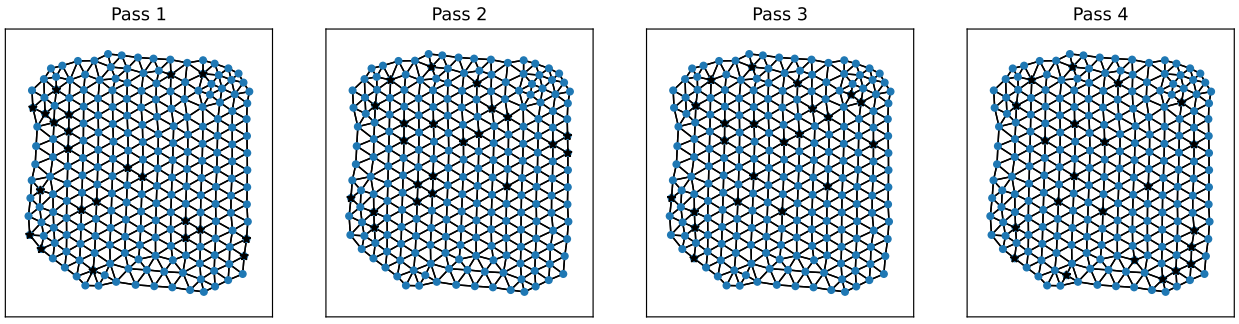


Figure 10: The tentative set of aggregate centers after each convolution "pass". This is the same mesh as in Fig 7.