1 Background

Here, we are attempting to train an ML agent to output both aggregates and interpolation for an AMG solver given some matrix $\boldsymbol{A}$. We will explore how well such an agent can learn to generalize on an *isotropic diffusion problem*, by generating:

1. A training set of 300 structured grids and 700 unstructured grids.

2. A testing set of 75 structured grids and 175 unstructured grids.

Each of these grid meshes are fairly small, overall containing between around 15 and 400 points. In the unstructured case, random points are generated in $[0,1]^2$, a convex hull constructed, then meshed with gmsh. However, due to computational costs, only 1/4 of the training and testing sets are loaded to reduce overall training time.

2 Generating Aggregates and Interpolation

To compute the aggregates and final interpolation operator, two individual networks are trained concurrently to perform each action, which we will label Θ_{Agg} and Θ_{Interp}, both taking some weighted graph and returning a new graph with same connectivity, but different weight values. The algorithm for finding the aggregates and interpolation 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_{Agg} to obtain a new set of node values and edge values. We will use the node values as a *scoring* and the edge values as path weights. Define the aggregate centers as the indices of the largest k node scores. Then, run Bellman-Ford on the graph with these aggregate centers and new edge weights to obtain a tentative aggregation operator, Agg.

3. Now, again convolve the graph of $\boldsymbol{A}$ but with Θ_{Interp} (with aggregate information) to obtain the aggregate smoother $\hat{\boldsymbol{P}}$. Form $\boldsymbol{P} := \hat{\boldsymbol{P}}\text{Agg}$.

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.

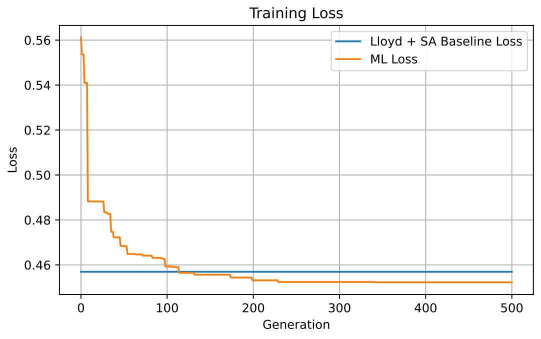
A basic overview of the training algorithm is that the method is seeded with some number of randomly generated networks. A subset of the best performing (most fit) networks are selected and *bred* with one another (crossing weights/traits) and *mutations* inserted (random perturbations to weights) to 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.

4 Discussion

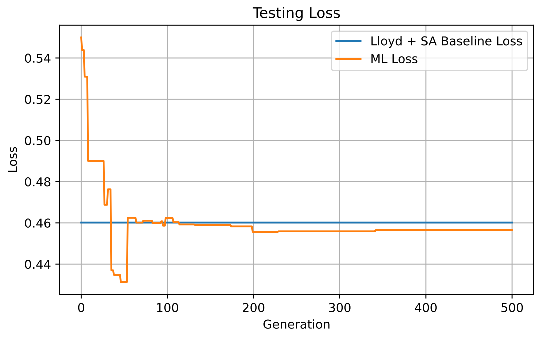
I think an obvious step for next week would be to make the distribution of problem sizes be more uniform and measuring the network performance then. It seems that the training skewed towards smaller problems, perhaps for a few reasons:

1. There are more of these smaller problems in the data sets

2. These problems will naturally have a lower convergence, probably meaning that it skews the overall average convergence



(a) Training loss



(b) Testing loss

Figure 1: Loss plots per generation for training and testing datasets, respectively.

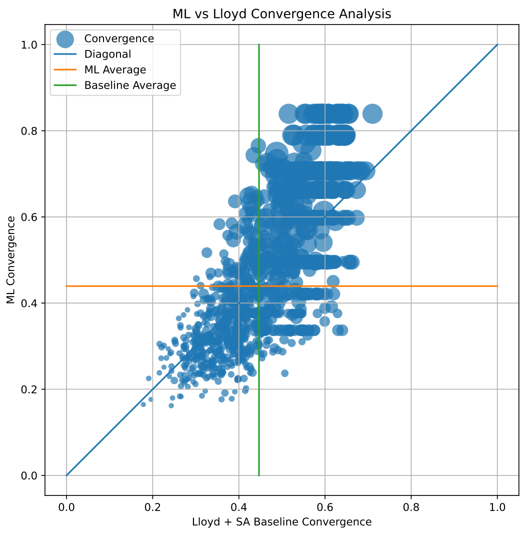
Currently the loss is computed as

$$\ell := \frac{1}{n} \sum_{i}^{n} c_{\mathrm{ML}}\left(i\right), \tag{1}$$

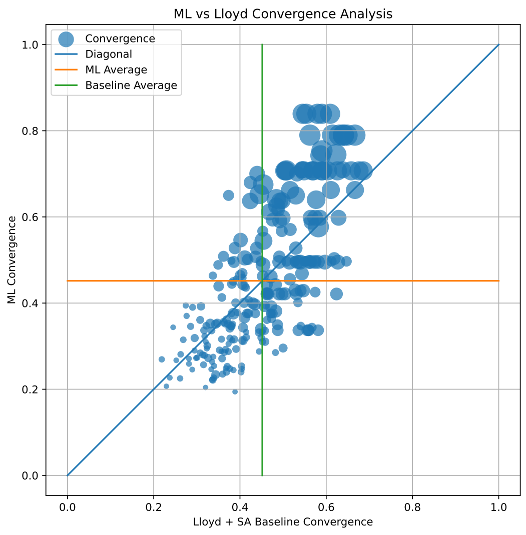
where $c_{\mathrm{ML}}\left(i\right)$ is the ML convergence of the i'th problem. Perhaps it be worth computing the loss as instead

$$\ell := \frac{1}{n} \sum_{i}^{n} \frac{c_{\mathrm{ML}}\left(i\right)}{c_{\mathrm{Lloyd}}\left(i\right)}, \tag{2}$$

where each problem is instead normalized by the baseline performance. Each term inside the summation can be thought of as the *relative performance* of the ML method versus Lloyd, with $c < 1$ indicating a lower convergence and $c > 1$ indicating higher performance.

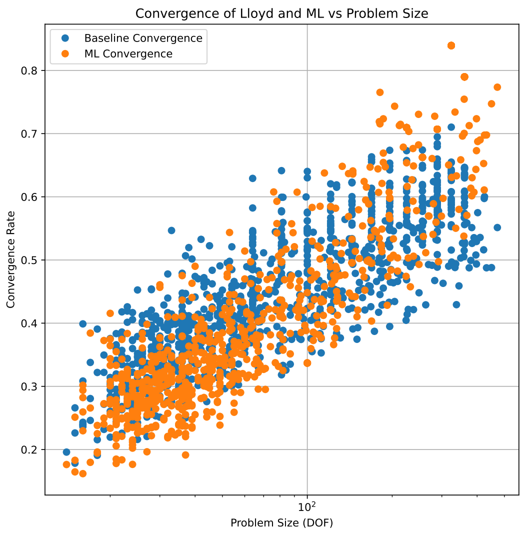


(a) Training convergence

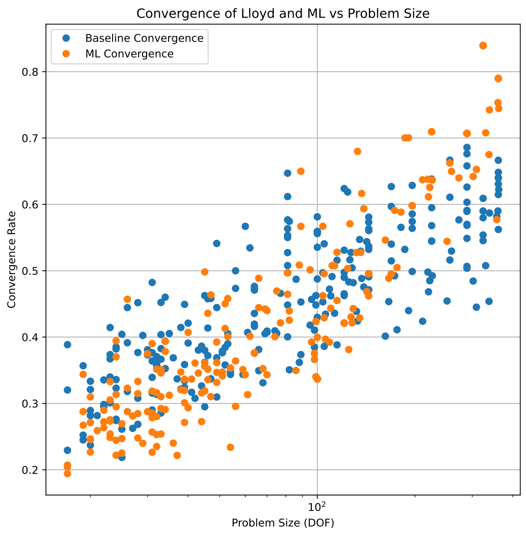


(b) Testing convergence

Figure 2: 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.

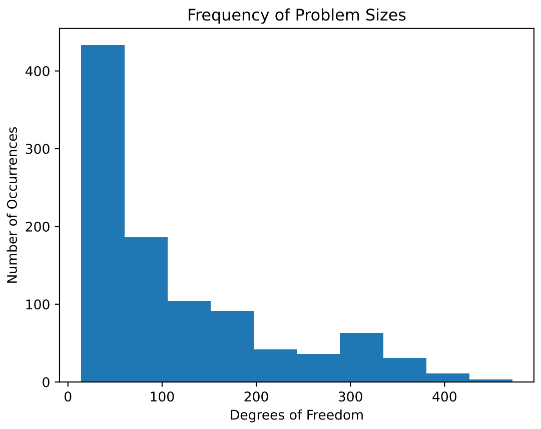


(a) Training convergence

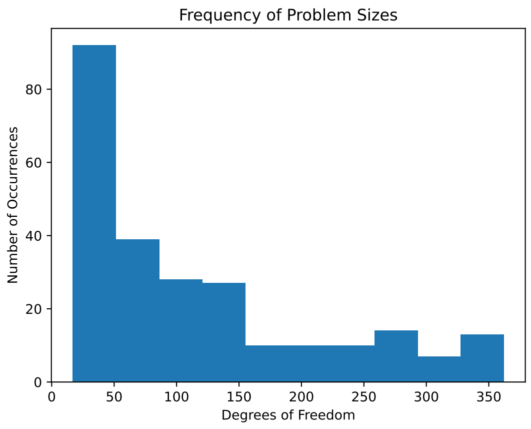


(b) Testing convergence

Figure 3: 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.

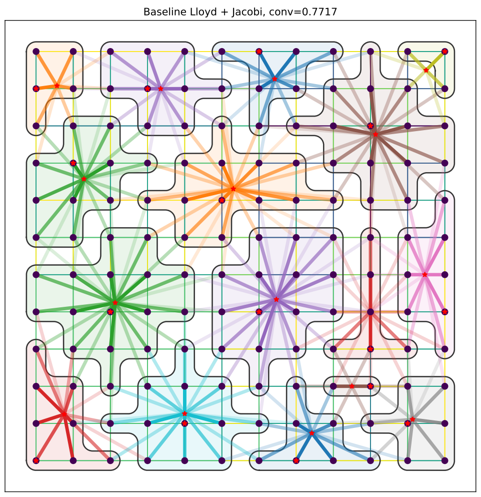


(a) Frequency of problem sizes in training set

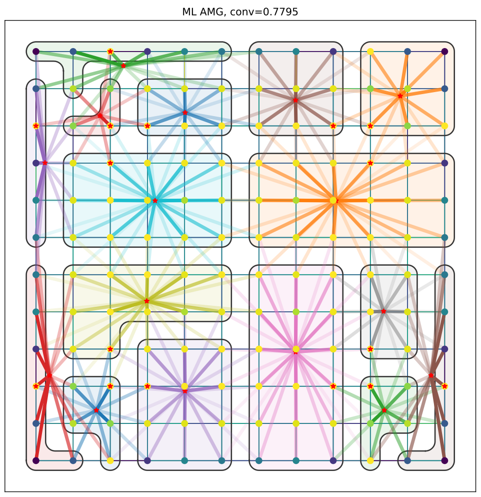


(b) Frequency of problem sizes in testing set

Figure 4: Frequency of problem sizes in each dataset. These skew towards small problems, which may explain the poor performance on the medium-to-larger problems.

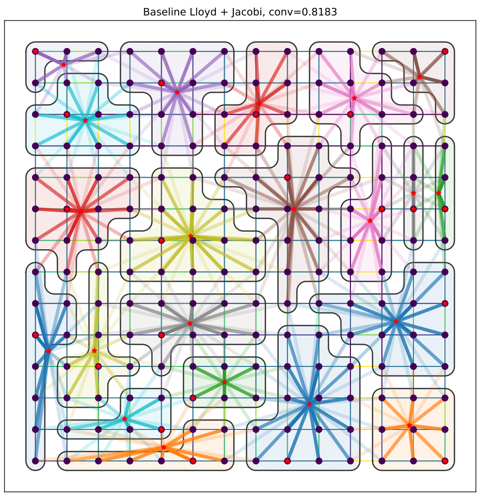


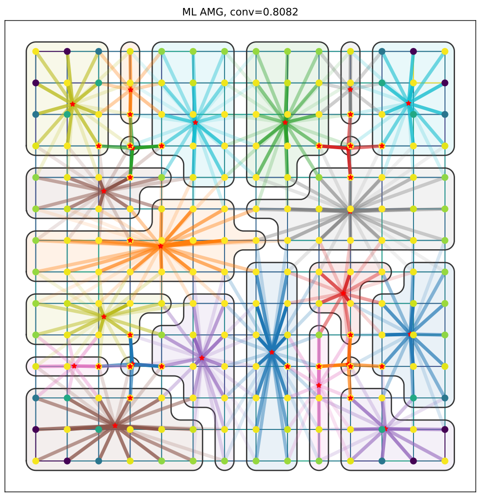
(a) Lloyd + Jacobi



(b) ML

Figure 5: Aggregate and interpolation data for a 12×12 structured grid.

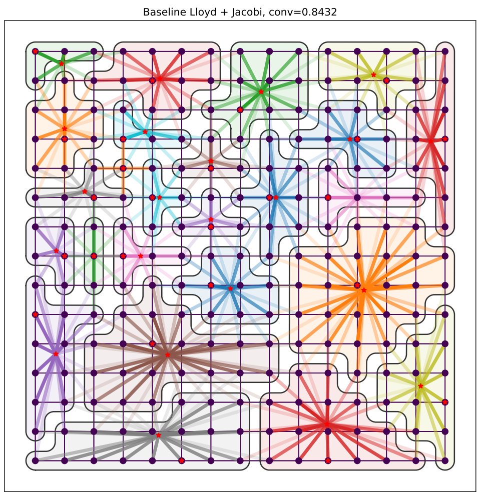


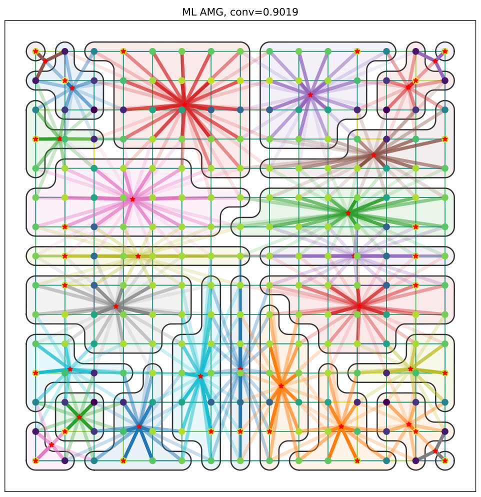


(a) Lloyd + Jacobi

(b) ML

Figure 6: Aggregate and interpolation data for a 14×14 structured grid.

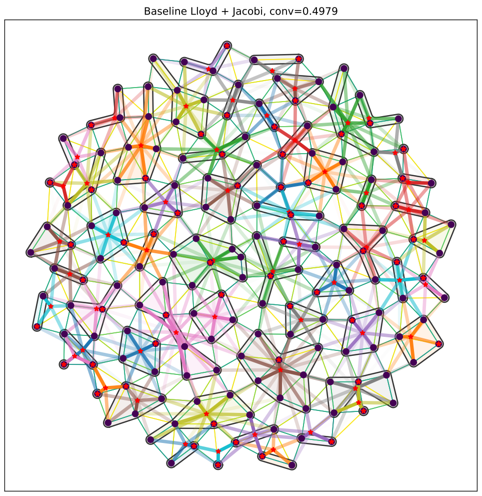




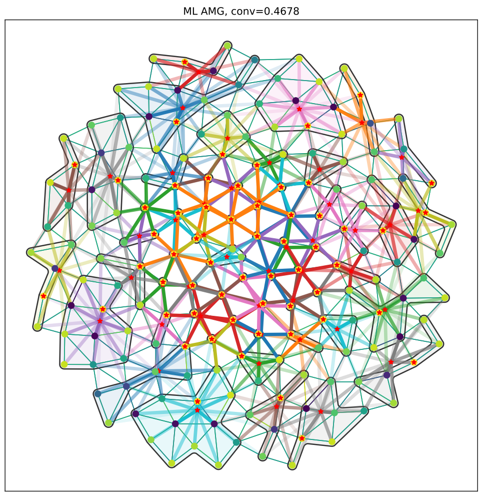
(a) Lloyd + Jacobi

(b) ML

Figure 7: Aggregate and interpolation data for a 15×15 structured grid.



(a) Lloyd + Jacobi



(b) ML

Figure 8: Aggregate and interpolation data for a circular unstructured mesh.