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, randomly containing somewhere between 25 and 225 points. In the unstructured case, random points are generated in $[0,1]^2$, then meshed with a Delaunay triangulation. However, due to computational costs, only 1/8 of the training set and 1/4 of the testing set are loaded during GA training.

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 $\boldsymbol{P} := \hat{\boldsymbol{P}}\text{Agg}$.

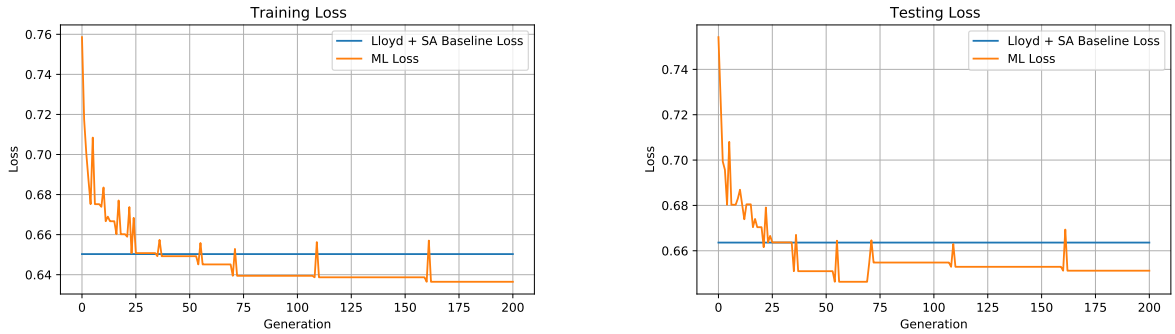
3 Genetic Training

Training of both networks is done at the same time with PyGAD[1], a genetic algorithm that can be used to train networks without any gradient information. 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 Results

The loss values for both the training and testing datasets are displayed in figure 1. The loss is the average convergence for the dataset: in both cases the average convergence for the ML method outperforms the average baseline loss after approximately 30 or so generations.

Convergence for all generated grids is displayed in Figure 2, with baseline convergence along the x-axis and ML convergence along the y-axis. The figure should be read by comparing points to their position relative to the diagonal. Samples where the ML and baseline method are equal will result in a position directly on the diagonal line, while those where the ML method is more convergent will give a position below the diagonal. A better baseline convergence will place samples above the diagonal line.



(a) Training loss

(b) Testing loss

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

Aggregate and interpolation data for a few structured grids are shown in Figures 3, 4, 5. Notice the lousy aggregates for the 15×15 case; in theory the ML method should have seen such a grid during training, yet it is unable to generalize.

Finally, the method is run on an unstructured, circular mesh (Fig 6). The ML method should not have seen anything like this during training, so it is somewhat encouraging that it was able to output a nice set of aggregates.

5 Future

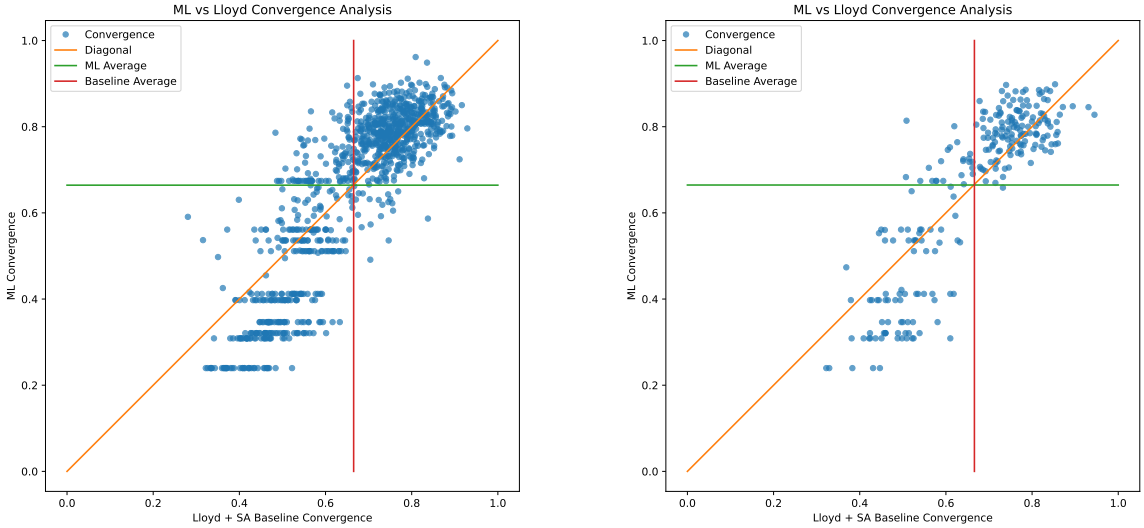
The actual training of the ML agent is horribly slow, taking about 13-14 hours just to reach 200 training generations. If we wanted to train on large datasets, this should probably be addressed. There are a few things to poke around with:

1. Fork the PyGAD library (or write our own GA package) and run the fitness calculation in parallel. This should be *embarassingly parallel*, as we are just doing some large number of AMG iterations to compute convergence.

2. Work out some continuous form of the agent, so we can use a gradient-based optimizer like Adam. Perhaps there are some tricks to be used to get this working?

References

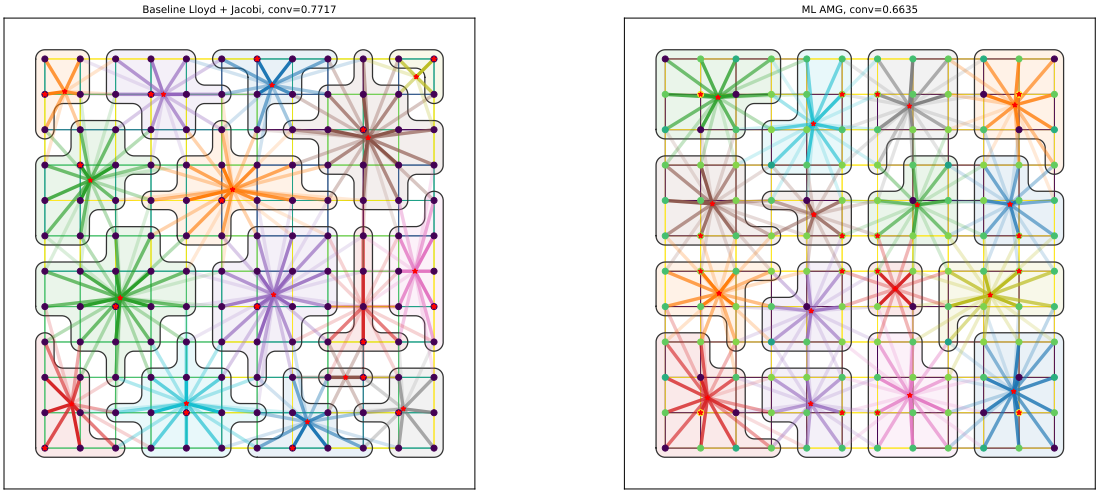
[1] A. F. Gad, *Pygad: An intuitive genetic algorithm python library*, 2021.



(a) Training convergence

(b) Testing convergence

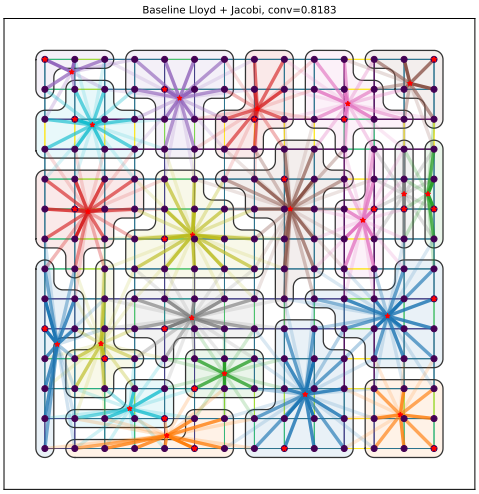
Figure 2: Convergence data for the ML AMG method vs a baseline Lloyd and Jacobi SA method. Values below the diagonal indicate a better convergence for the ML.



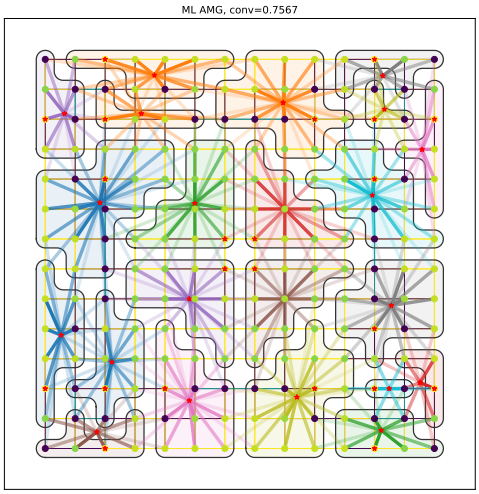
(a) Lloyd + Jacobi

(b) ML

Figure 3: Aggregate and interpolation data for a 12×12 structured grid.

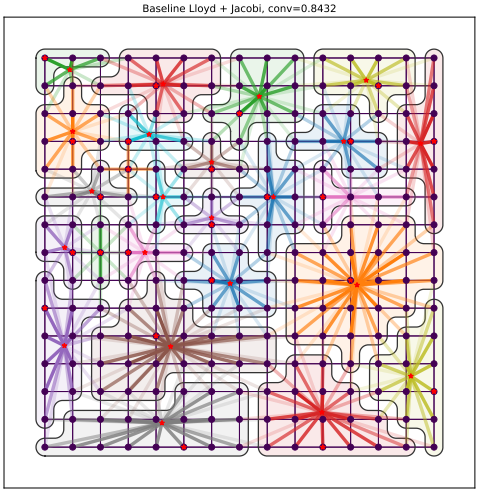


(a) Lloyd + Jacobi

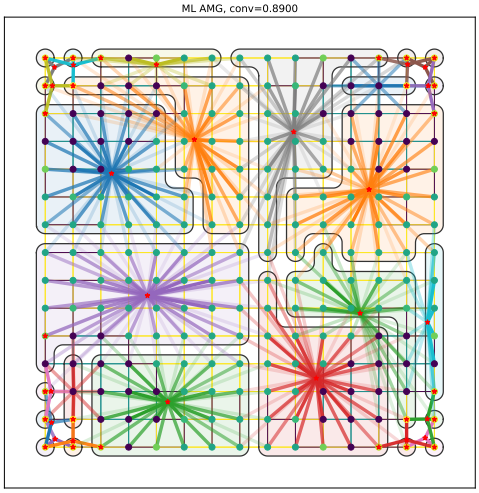


(b) ML

Figure 4: Aggregate and interpolation data for a 14×14 structured grid.

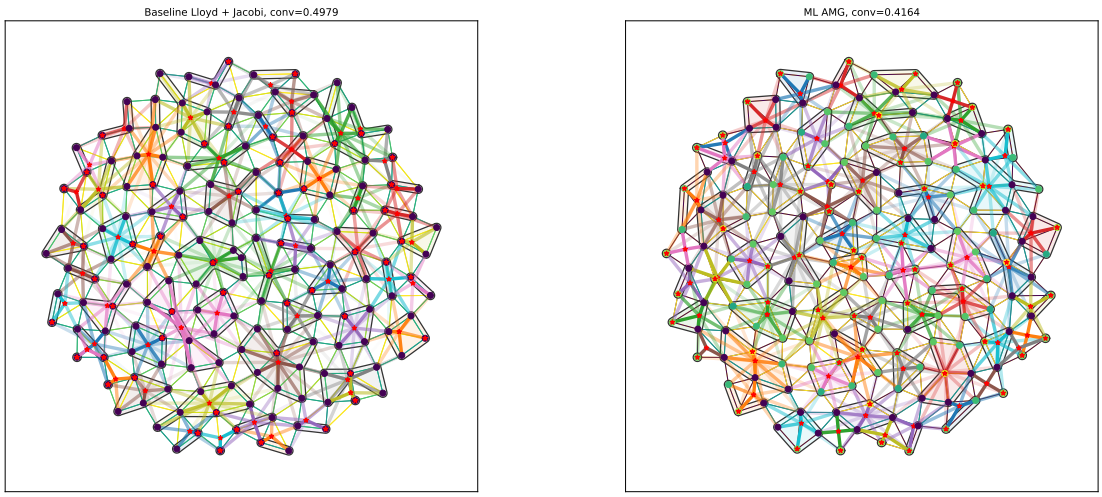


(a) Lloyd + Jacobi



(b) ML

Figure 5: Aggregate and interpolation data for a 15×15 structured grid.



(a) Lloyd + Jacobi (b) ML

Figure 6: Aggregate and interpolation data for a circular unstructured mesh.