1 Background

The problem being solved is the Poisson problem in one dimension:

$$-\nabla^2 \boldsymbol{u} = f$$

$$f(\boldsymbol{x}) = \boldsymbol{x}$$

Defined on the following domain with Dirichlet boundary conditions:

$$\Omega = [-1, 1]$$

$$\partial\Omega = 0$$

This is discretized using finite differences on a grid of $N = 31$ equispaced points and results in the following system:

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{x}$$

where

$$\boldsymbol{A} = \frac{1}{h^2}\text{tridiag}\left([-1, 2, -1]\right)$$

$$h = \frac{1}{N+1}$$

The above linear system is solved using a two-level V cycle multigrid method using the following steps:

1. Generate an initial random guess, $\boldsymbol{u}_0$.
2. Pre-smooth the guess using $\nu = 5$ iterations of Jacobi using weight ω.
3. Restrict the grid and residual using the operator $\boldsymbol{R} = \boldsymbol{P}^T$. The given interpolation operator $\boldsymbol{P}$ is constructed as the *ideal interpolation operator*.
4. Perform a linear solve on the restricted residual to obtain the coarse solution.
5. Interpolate the solution to the fine grid using operator $\boldsymbol{P}$.
6. Post-smooth using 5 iterations of Jacobi, with same weight ω.

The Jacobi smoothing weight that is used, ω, is the optimal weight that minimizes the overall factor of convergence for the method:

$$\omega = \arg\min_{\omega \in (0,1)} \lim_{k\to\infty} \frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}^*}{\boldsymbol{u}_k - \boldsymbol{u}^*}$$

where $\boldsymbol{u}^*$ is the "optimal" solution, precomputed by a dense linear solve. Computationally, ω is approximated using a brute-force sweep of values in $(0, 1)$.

As a side note, brief experimentation suggests the Jacobi weight as a function of convergence factor to be unimodal with the 1D Poisson problem (Fig 1). This brute-force sweep could perhaps be converted to a numerical optimization to speed up running time.

2 Generating Coarse Grids

To train the model, a set of 6016 C/F grids were randomly generated. A "reference" C/F grid was first generated such that every third grid point is a coarse point, and the rest are fine points. Educated readers may recognize this as a *coarsening by 3*.

This reference grid was randomly permuted such that each grid point had a random probability of being flipped to the opposite value. I.e. coarse point to fine, and fine point to coarse. Random trials of the following probabilities were used:

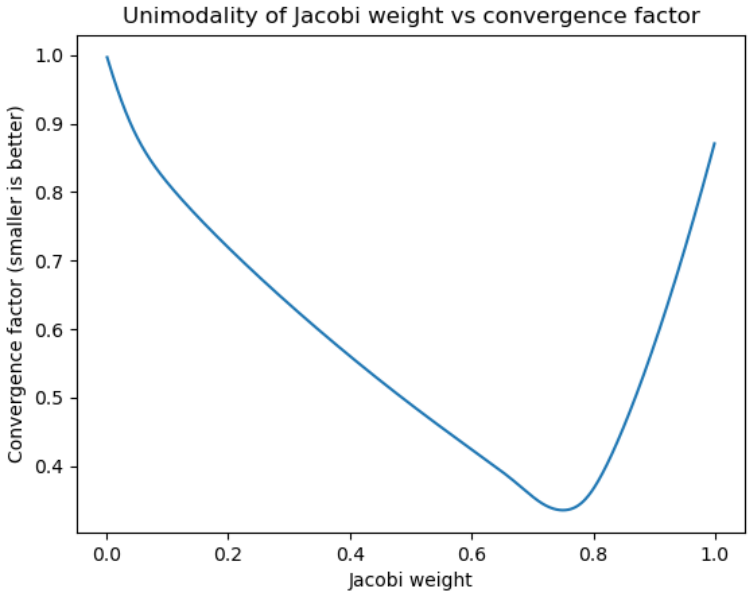

Figure 1: Graph showing the factor of convergence vs Jacobi weight for a grid with coarsening by 3

$$p = \{0.01, \quad 0.05, \quad 0.1, \quad 0.25, \quad 0.5, \quad 0.75\}$$

For each value of p, 1000 random grids were generated according to the above permutation strategy. Each random grid was then used to solve the defined Poisson problem, and the optimal Jacobi weight, ω, and convergence factor were recorded.

Then, in an attempt to train the model on a few "sane" grids, 16 grids with uniform spacing between coarse/fine points were generated. The coarsening factors used were:

$$r = \left\{ \frac{1}{9}, \quad \frac{1}{8}, \quad \frac{1}{7}, \quad \frac{1}{6}, \quad \frac{1}{5}, \quad \frac{1}{4}, \quad \frac{1}{3}, \quad \frac{1}{2}, \quad 2, \quad 3, \quad 4 \quad 5, \quad 6, \quad 7, \quad 8, \quad 9 \right\}$$

where values of $r < 1$ refer to grids where every $\frac{1}{r}$ points is a fine point and the rest coarse, and $r > 1$ refers to grids with every r points being a coarse point and the rest fine. These grids were then also used to solve the Poisson problem and their weights and convergence factors recorded.

The code used to generate these grids is given in `grids/gen_grids.py`. Note that it can take a decent amount of time to run, approximately 21 minutes on my machine to completely finish.

3 CNN Model

In an attempt to predict optimal Jacobi weight given an arbitrary grid a CNN of the following architecture was trained[1]:

1. 1-dimensional convolutional layer with kernel size 5. Input 1 layer, output 2 layers. ReLU activation function.
2. 1-dimensional convolutional layer with kernel size 5. Input 2 layers, output 8 layers. ReLU activation function.
3. Fully connected layer with input size 184, output size 92. ReLU activation function.

[1]I neither claim that this CNN is optimal nor was its architecture particularly well thought-out. I am sure that tweaking the layers will obtain better results

4. Fully connected layer with input size 92, output size 1. ReLU activation function.

3.1 Training

The previously generated set of 6016 random grids was split into training and testing sets, with a random 85% of entries going to the training set and the remaining 15% going into the testing set.

The grids were formed into a $N \times 1 \times n$ tensor, with $N = 6016$ being the number of grids and $n = 31$ being the grid size. The tensor representation was formed by means of assigning 1 to coarse points and -1 to fine points:

$$T_{i1k} = \begin{cases} 1 & \text{point } k \text{ in grid } i \text{ is coarse} \\ -1 & \text{point } k \text{ in grid } i \text{ is fine} \end{cases}$$

Of course, this assignment is arbitrary and swapping the values would produce similar results. The input weights were then normalized such that their range lies between $[0, 1]$. When displaying any output from the CNN, this transformation must be undone for the results to be sensible. The grid dataset class contains a `scale_output()` method to do this in `jacobi-cnn/model.py`.

This CNN was run through 200 iterations of a training loop using a MSE (mean-square error) loss function. In each iteration, minibatches of size 500 were used to train and backpropagate the model. Since the number of training samples is not evenly divisible by 500, these "extra" samples were arbitrarily discarded at each iteration. The MSE loss and L1 loss at each iteration is given by Figures 2, 3.

3.2 Results

Results of plotting the true vs predicted Jacobi weights can be seen in Figures 4 (Testing dataset) and 5 (Training dataset). For true weights > 0.7 the CNN tends to under-predict the weight, while < 0.7 it tends to over-predict. An interesting thing to note is the horizontal line of predictions at $y \approx 0.67$; this could indicate that the CNN was train for too few iterations. It is also interesting that there are no predicted values under this horizontal line, perhaps more training samples are needed.

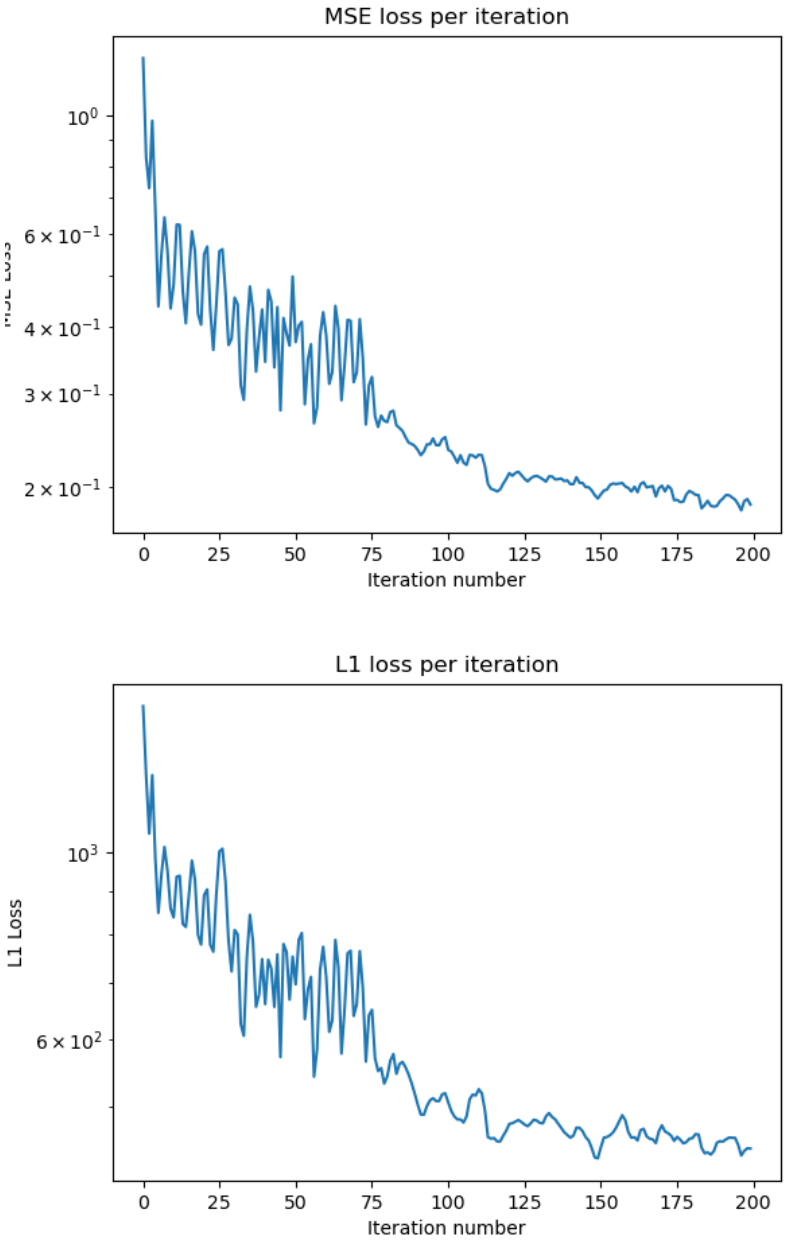

Figure 2: MSE, L1 loss per training iteration

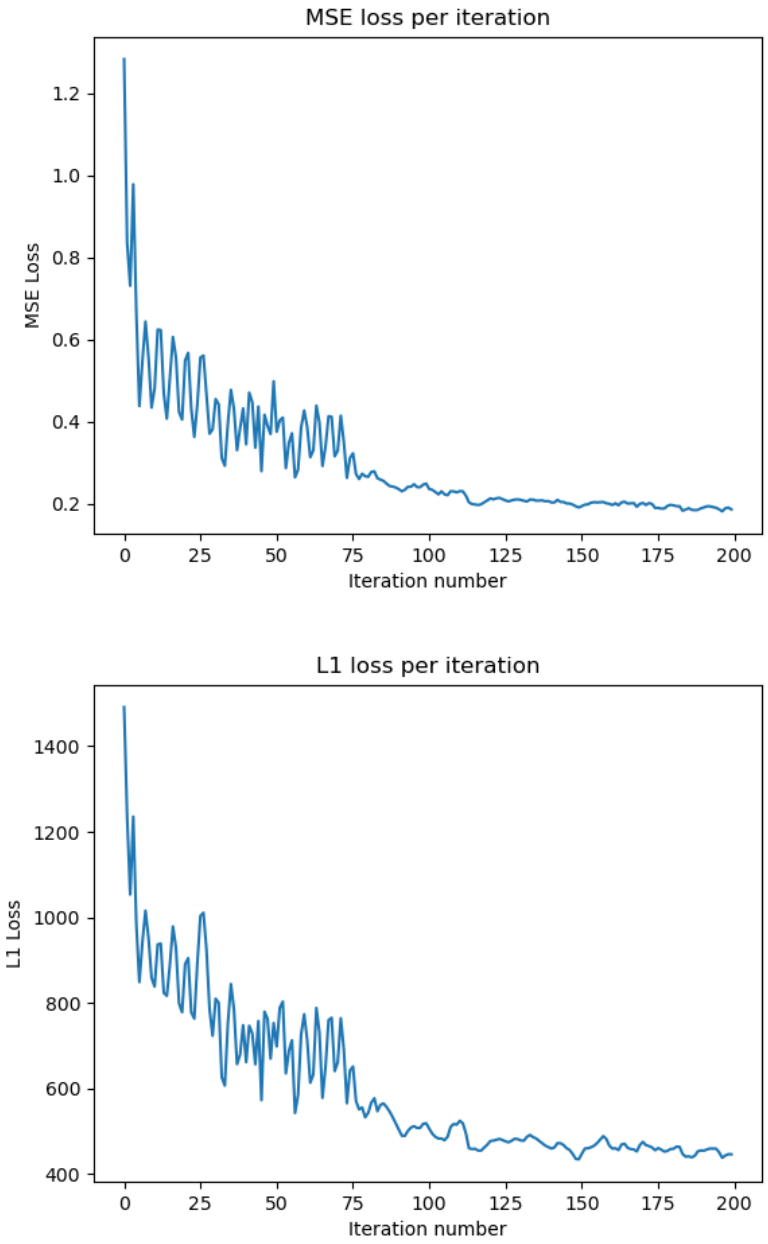

Figure 3: MSE, L1 loss per training iteration. Linear y-axis, in the event that someone complained about the log scaling.

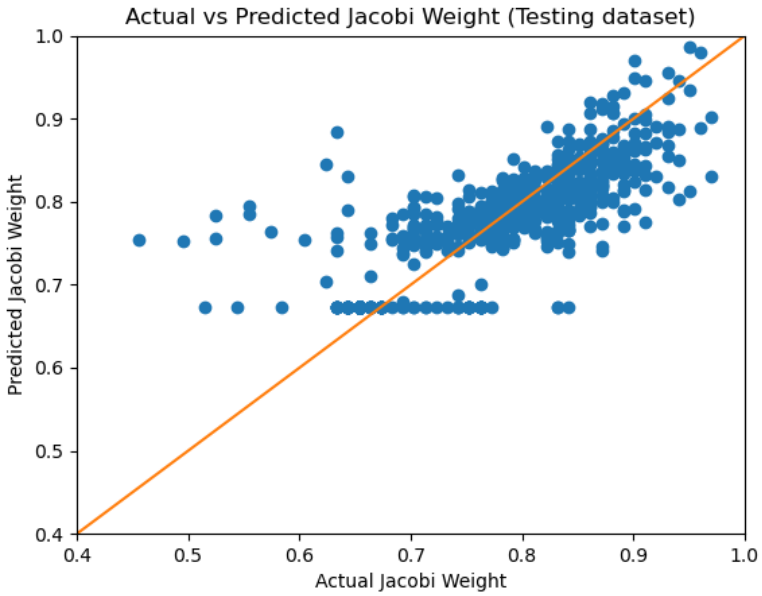

Figure 4: Predicted vs Actual Jacobi Weights (Testing Dataset)

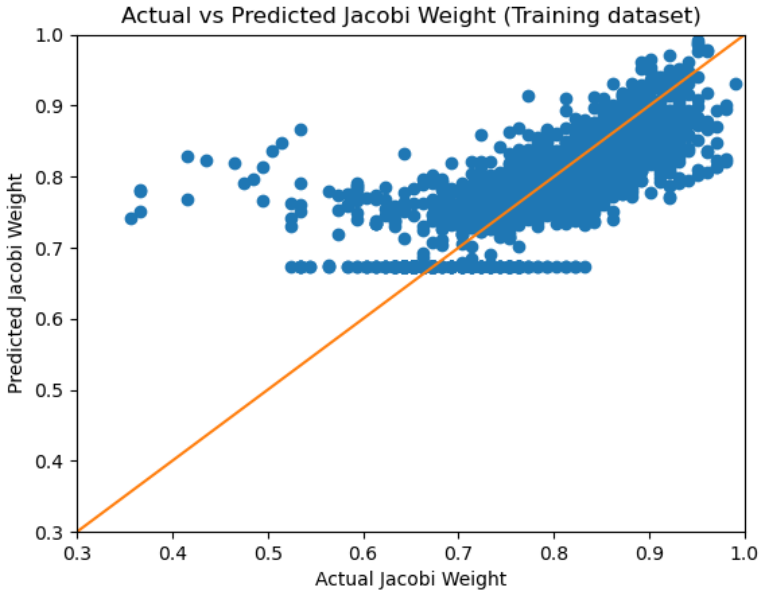

Figure 5: Predicted vs Actual Jacobi Weights (Training Dataset)