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 one iteration 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 one iteration 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 a bracked numerical optimization method, with the assumption that ω is unimodal with respect to the convergence factor.

2 Generating Coarse Grids

To train the model, a set of 96000 C/F grids were randomly generated via random perturbation. Several "reference" grids were generated according to varying coarsening factors, of which the following values were used:

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

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 used to solve the Poisson problem and their weights and convergence factors recorded.

Each 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:

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

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

In total, 16 convergence factors $\times$ 6 probability trials $\times$ 1000 random iterations = 96000 random grids were created. The code used to generate these grids is given in `grids/gen_grids.py`.

3 CNN Model

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

1. 1-dimensional convolutional layer with kernel size 7. Input 1 layer, output 20 layers. ReLU activation function.
2. Max pooling layer with kernel size 2 and stride 2. This takes the maximum element of each block of 2 elements, and effectively cuts the size of the input tensor in half.
3. 1-dimensional convolutional layer with kernel size 5. Input 20 layers, output 20 layers. ReLU activation function.
4. Max pooling layer with kernel size 2 and stride 2.
5. 1-dimensional convolutional layer with kernel size 3. Input 20 layers, output 20 layers. ReLU activation function.
6. Max pooling layer with kernel size 2 and stride 2.
7. Fully connected layer with input size 20, output size 15. ReLU activation function.
8. Fully connected layer with input size 15, output size 10. ReLU activation function.
9. Fully connected layer with input size 10, output size 1. ReLU activation function.

3.1 Training

The previously generated set of 96000 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 $N = 96000$ 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`.

[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

For the Jacobi model, the CNN was trained for 100 epochs while for the convergence factor model it was trained for 300 epochs. Training was done using a MSE (mean-square error) loss function with stochastic gradient descent. 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 1, 2. These are also compared against the loss of a "trivial predictor", where the predicted value is simply the average of all input values.

3.2 Results

Results of plotting the true vs predicted Jacobi weights and convergence factors can be seen in Figures 3 and 4, respectively. Interesting "peaks" in the predicted values occur when the CNN is more optimistic than the actual Jacobi or convergence factor allows. These results are interesting and the input grids could perhaps be extreme outliers. The convergence factor values still have quite a large relative error, but the results seem to be encouraging and suggest that a convergence factor could actually be learned. A grid corresponding to one of the dips is displayed in Figure 5.

As a fun experiment, the most "convergent" (or a local minimum) was found by numerically optimizing a random input tensor until the convergence factor was minimized. Results of this can be seen in Figure 6.

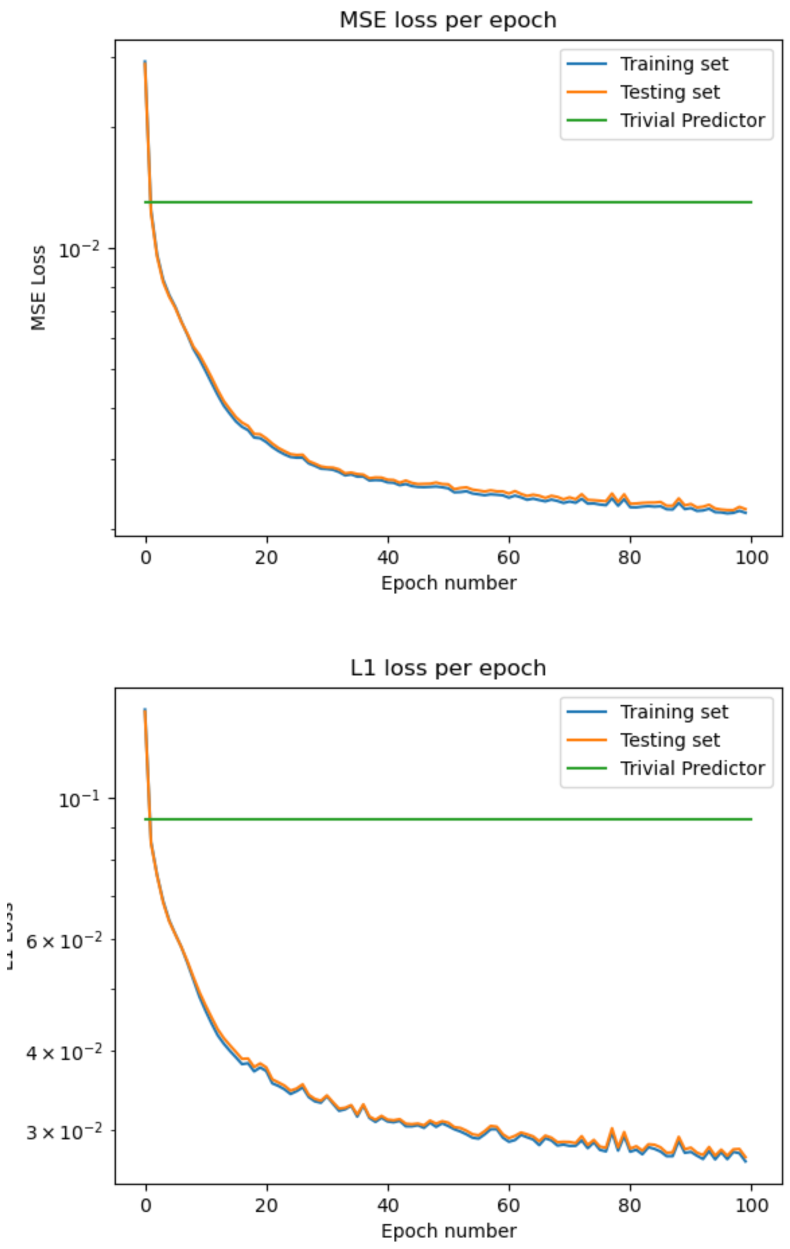
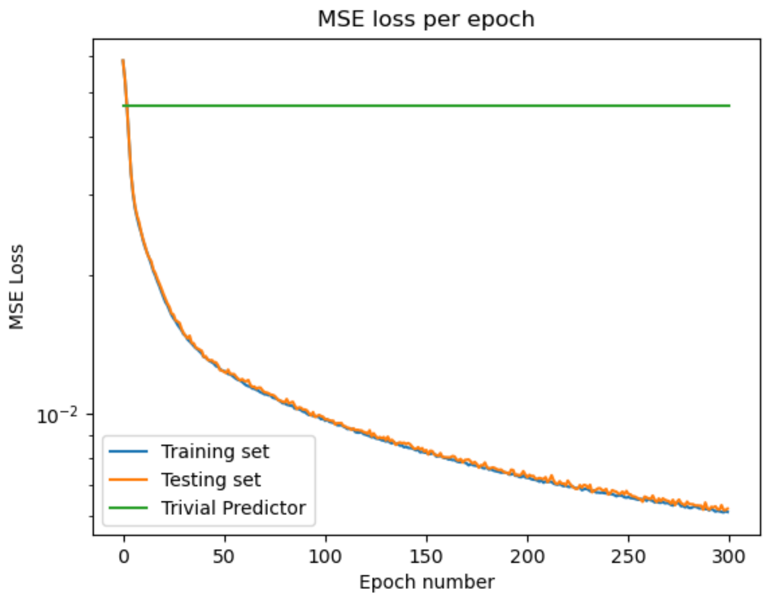


Figure 1: MSE, L1 loss per training iteration for Jacobi weights



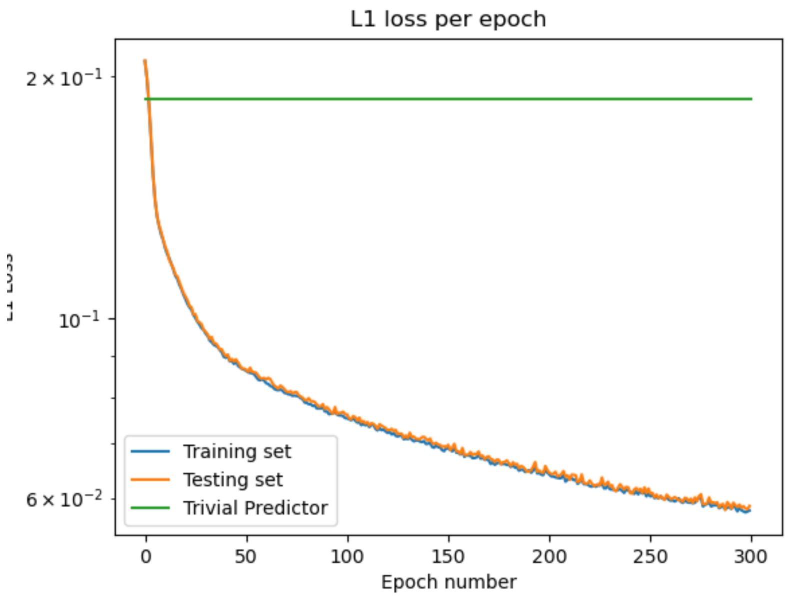
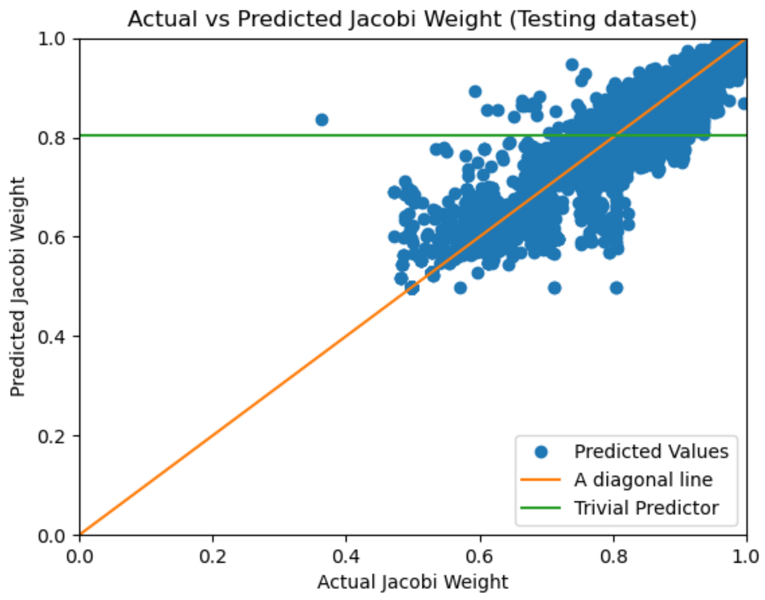


Figure 2: MSE, L1 loss per training iteration for convergence factor



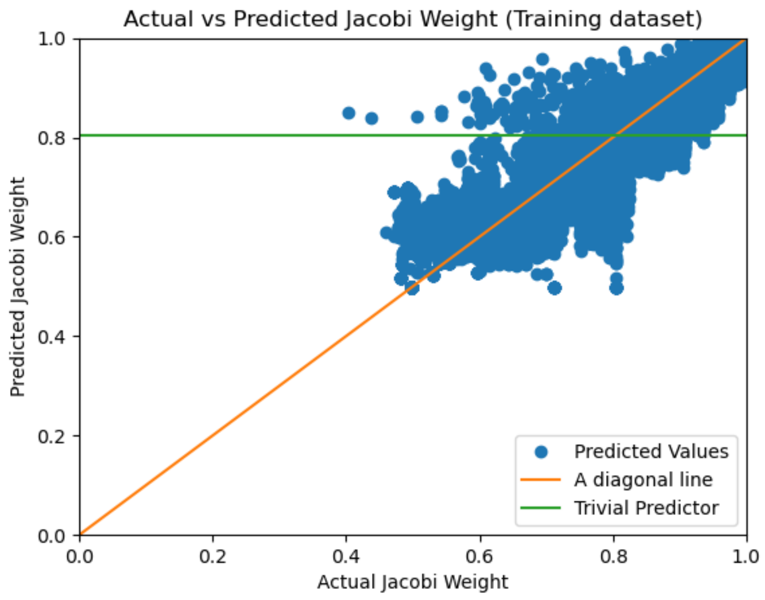
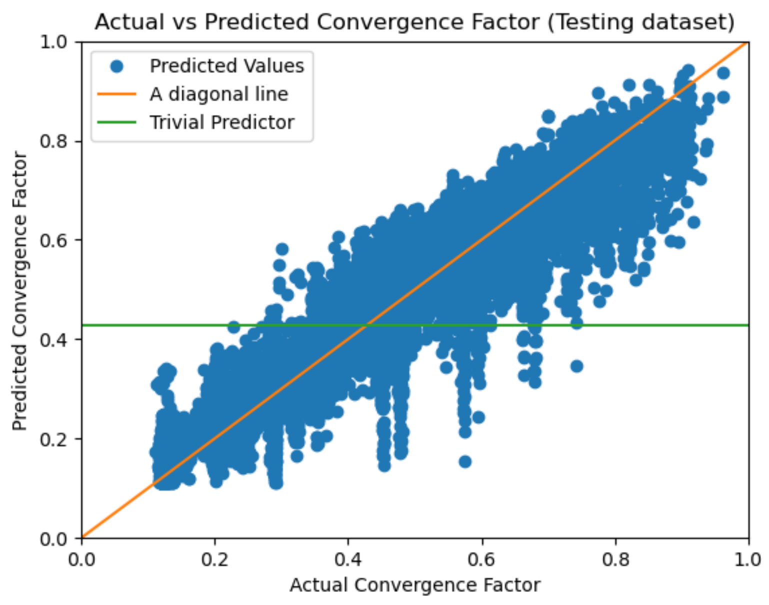


Figure 3: Predicted vs Actual Jacobi Weights



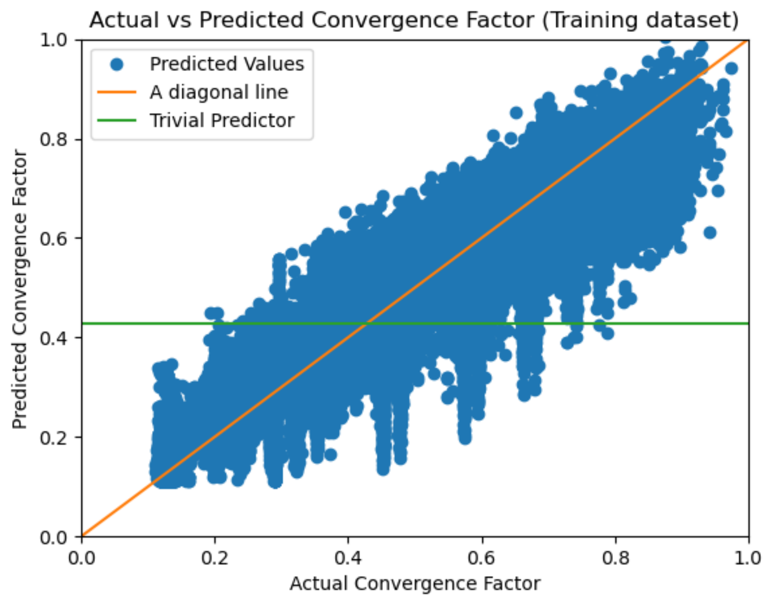


Figure 4: Predicted vs Actual Convergence Factors

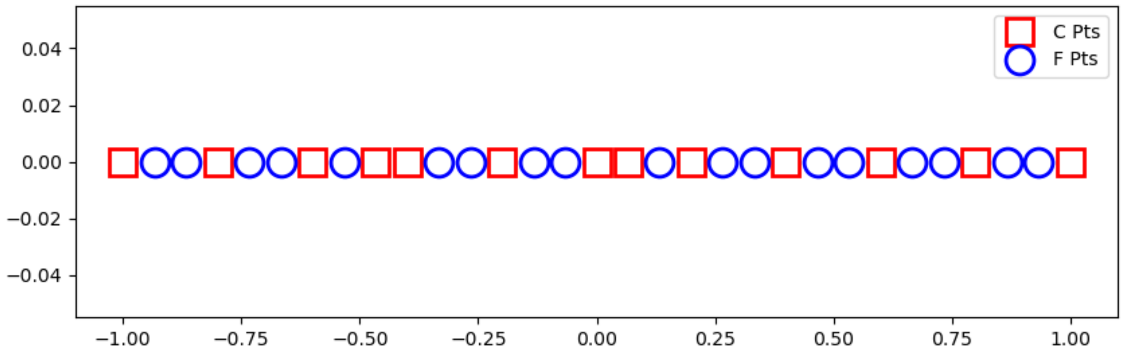


Figure 5: Grid corresponding to worst prediction error. Actual value of $\omega = 0.6799$, predicted $\hat{\omega} = 0.2950$. Absolute error 0.384, relative error $0.566 = 56.6\%$.

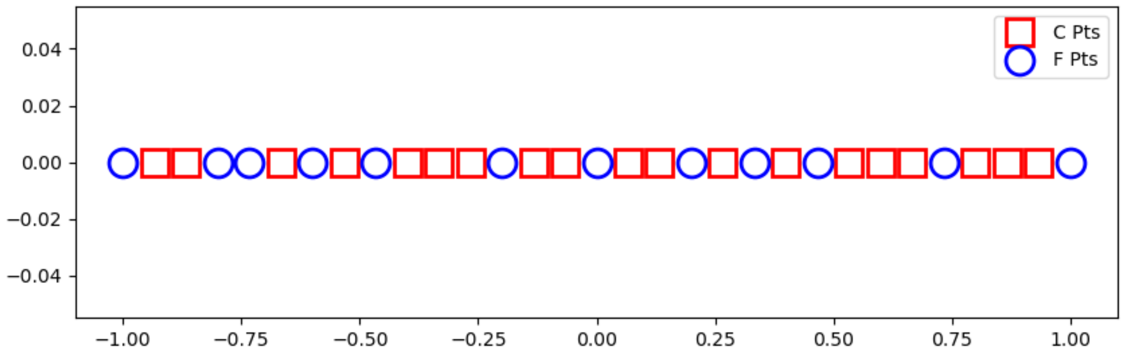


Figure 6: Grid obtained by numerically optimizing CNN output. The neural net was run "in reverse", by running SGD on the input tensor instead of the network weights. Values > 0 were interpreted as coarse points, and negative values as fine points. True convergence factor: 0.2327, predicted factor: 0.2179.