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 direct 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 with a bracketed numerical optimization method, with the assumption that ω is unimodal with respect to the convergence factor. If ω is not unimodal, which could possibly be the case for more complicated differential equations, then a more sophisticated method would be required to find the global optimum.

2 Generating Coarse Grids

To train the model, a set of 96000 C/F grids were 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. The layers of this network were inspired by those used in Residual neural networks (ResNet) in which the output of a layer may "skip" over certain layers, thus speeding up training and allowing for deeper networks. A graphical description is given in Figure 1 and also an overview is described below:

1. 4 1-dimensional convolution layers with kernel size 7. Zero padding of size two is applied to the left and right to keep the output element the same shape — this is for the residual architecture so that output vectors have equal dimensionality. Each convolutional layer has a ReLU activation function. The first layer takes one input layer and emits 10 output layers, subsequent convolutions have 10 input and output layers. Even numbered convolution layers k take as input both the output of layers $k-1$ and $k-2$, while odd numbered layers only take the output from layer $k-1$.
2. 4 1-dimensional convolution layers with kernel size 5. Zero padding is applied. ReLU activation. 10 input, 10 output layers. Similar residual pushing strategy from above.
3. 4 1-dimensional convolution layers with kernel size 3. Zero padding is applied. ReLU activation. 10 input, 10 output layers. Similar residual pushing strategy from above.
4. Max pooling layer with kernel size 2 and stride 2.
5. The tensor is flattened from dimensions $N \times 5 \times 15$ to $N \times 1 \times 75$.
6. Fully connected layer with input size 75, output size 40. ReLU activation function.
7. Fully connected layer with input size 40, output size 10. ReLU activation function.
8. Fully connected layer with input size 10, output size 1. ReLU activation function.

3.1 Training

The previously generated set of 96,000 random grids was first filtered of any duplicate grids into a unique set of approximately 60,000 grids. This was then 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 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

Model	Metric	Dataset	Value
Jacobi Weight	MSE	Training	0.0015810804907232523
Jacobi Weight	MSE	Testing	0.0016168837901204824
Jacobi Weight	L1	Training	0.028497139936647067
Jacobi Weight	L1	Testing	0.02862278080380772
Convergence Factor	MSE	Training	0.0004870382254011929
Convergence Factor	MSE	Testing	0.00050189538160339
Convergence Factor	L1	Training	0.015667168480636044
Convergence Factor	L1	Testing	0.015805707197538342

Table 1: Final training MSE/L1 loss values for the two models. Lower values correspond to higher model accuracy.

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`.

The CNN was separately trained for 20 and 30 epochs for the Jacobi weights and convergence factor models, respectively. Training was done using a MSE (mean-square error) loss function with the *Adam* optimizer. In each iteration, minibatches of size 500 were used to train and backpropagate the model. Nothing special was performed for minibatch sizes under 500, in the case that the total number of grids was not divisible by 500. The MSE loss and L1 loss at each iteration is given by Figures 2, 3. 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 4 and 5, respectively. Final MSE/L1 loss values are given in Table 1.

3.3 Discrete Grid Optimization

In an attempt to find the most "convergent" grid, a *Basin-hopping* optimizer was used to obtain the grid corresponding to the minimum convergence factor, with the convergence factor being predicted by the neural network. Results of this minimization can be seen in Figure 6. The actual grid in the training/testing data that minimizes the convergence can be seen in Figure 7, though the results may be obvious.

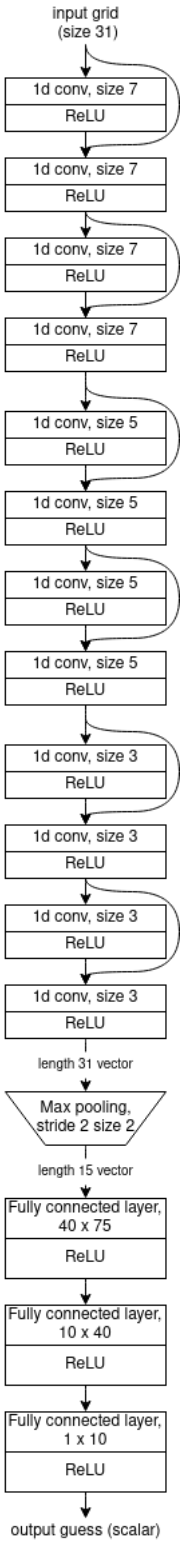


Figure 1: Architecture of the CNN with "skipping" akin to a ResNet. These skips may avoid vanishing gradients and can simplify training of the network.

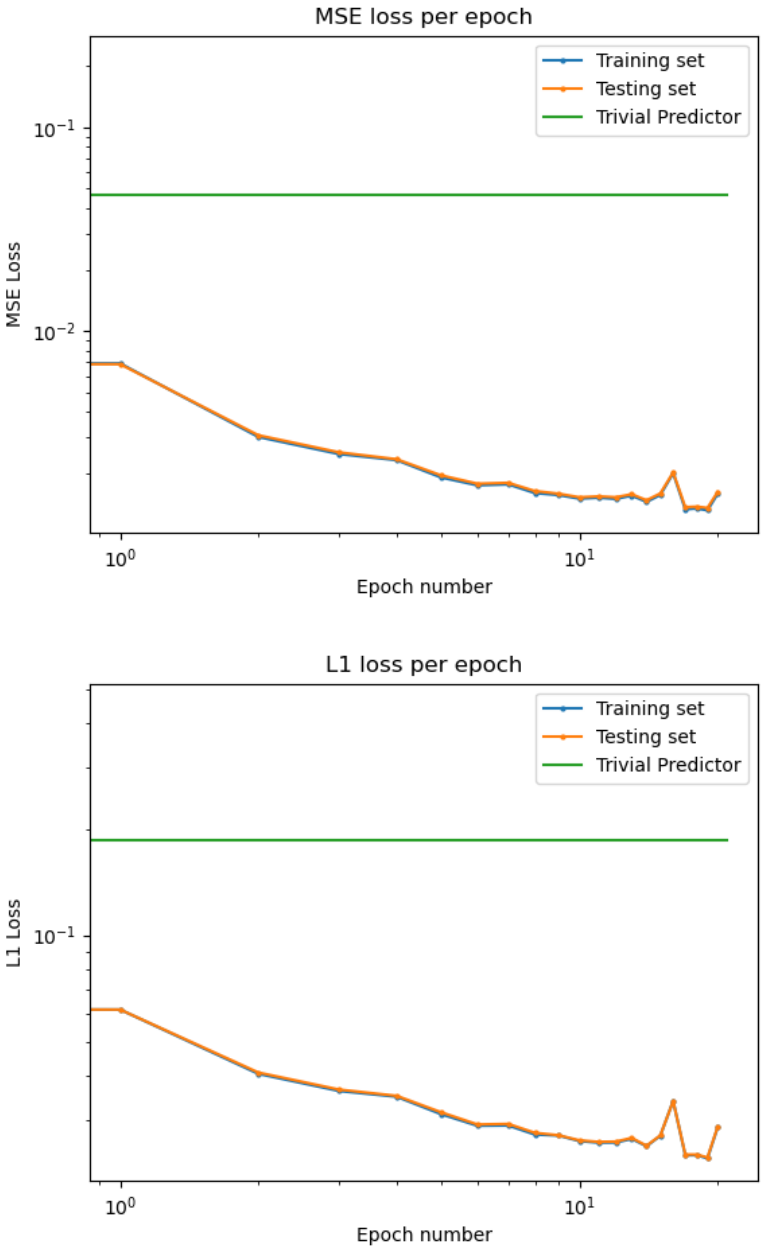


Figure 2: MSE, L1 loss per training iteration for Jacobi weights

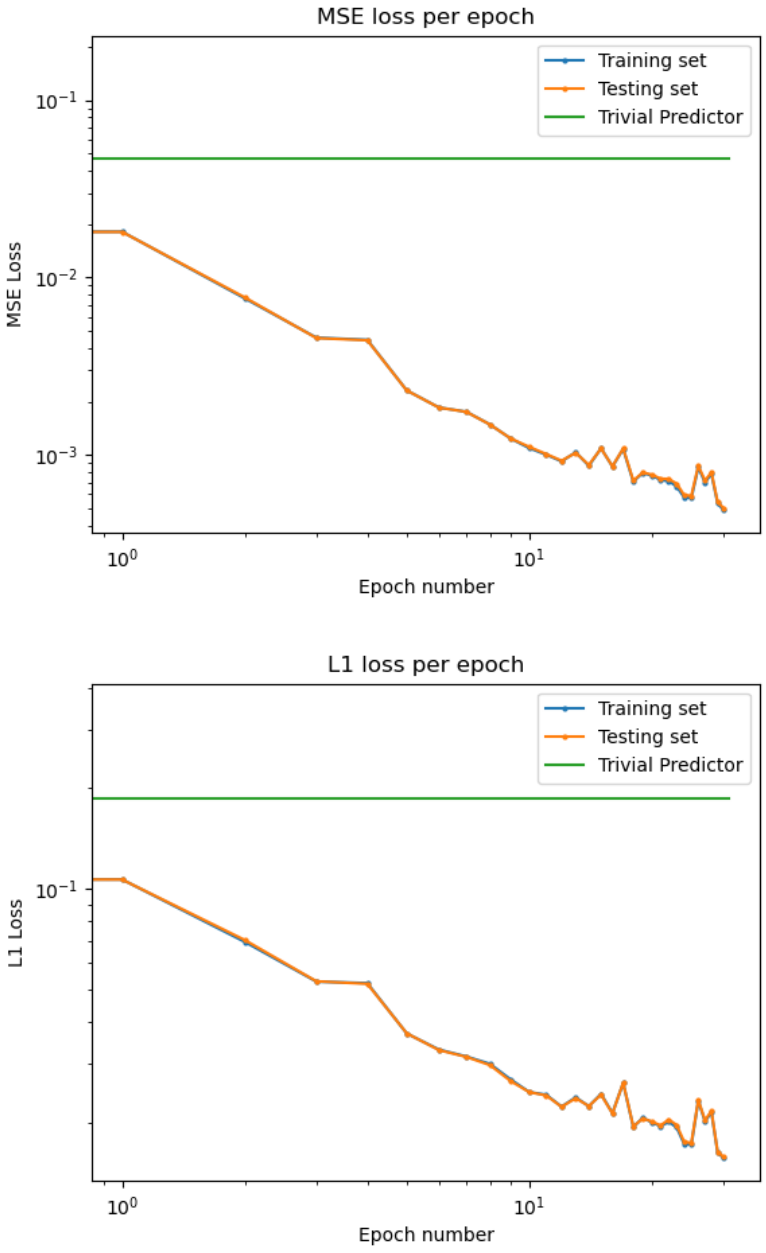
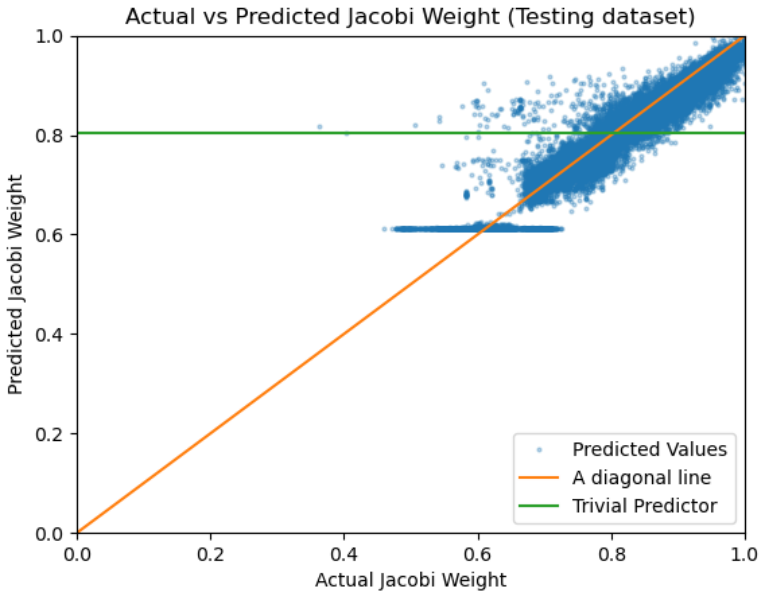


Figure 3: MSE, L1 loss per training iteration for convergence factor



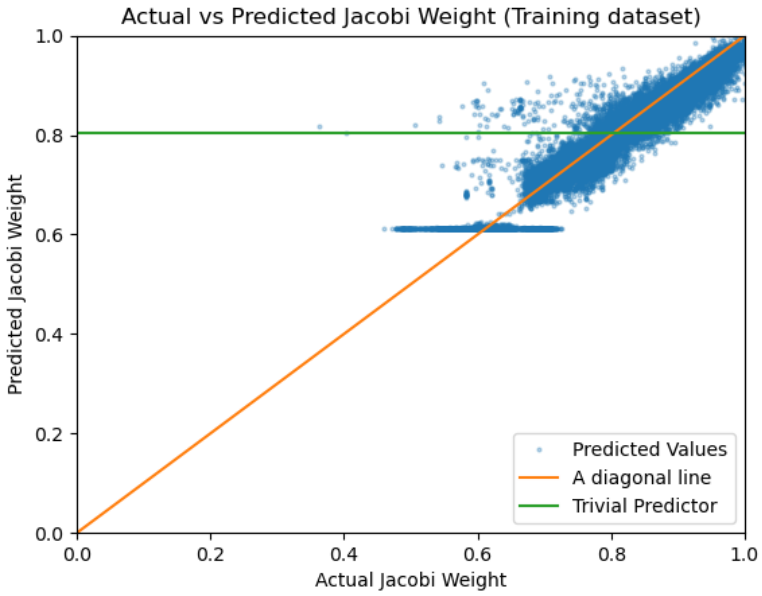
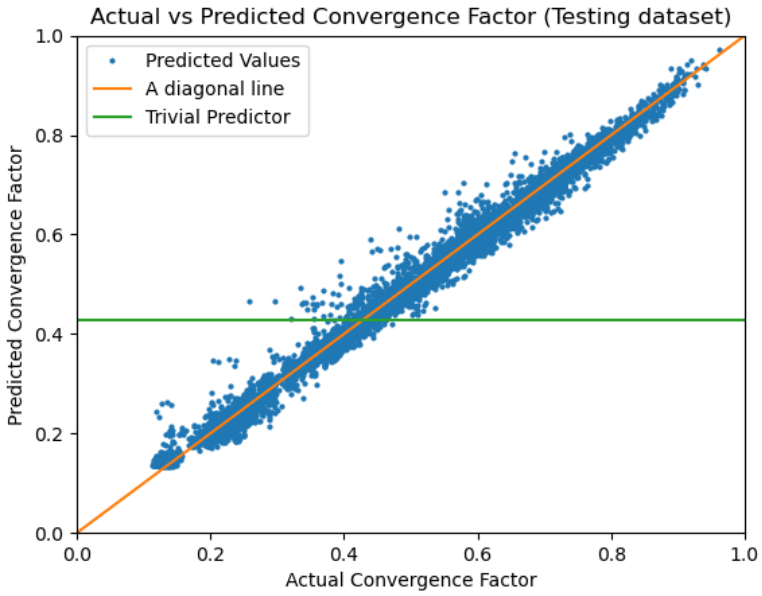


Figure 4: Predicted vs Actual Jacobi Weights



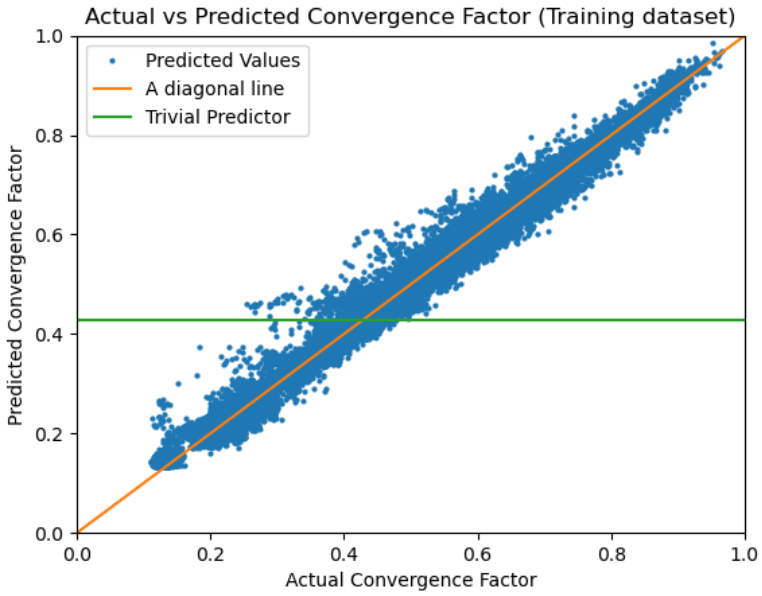


Figure 5: Predicted vs Actual Convergence Factors

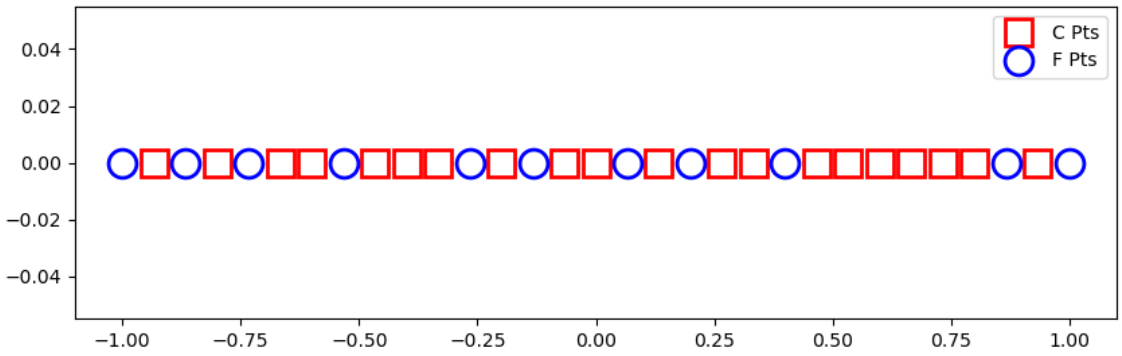


Figure 6: Grid obtained by optimizing CNN output. This was computed using SciPy's Basin-hopping optimizer. True convergence factor: 0.1464, predicted factor: 0.1571.

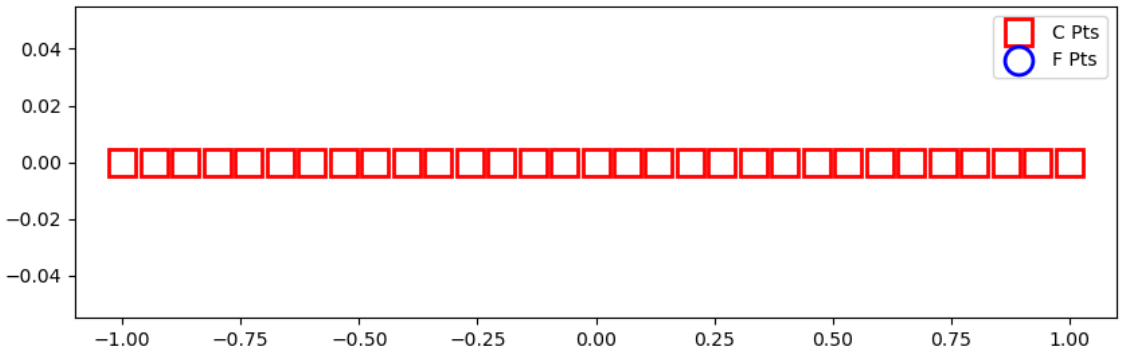


Figure 7: Grid in the testing/training data that minimizes convergence factor. This is just a grid of all coarse points, which reduces to a direct solve.