1 Background

The problem being solved is the Poisson problem in one dimension with variable coefficients:

$$-\nabla \cdot \left(k\left(\boldsymbol{x}\right)\nabla \boldsymbol{u}\right) = f$$

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

Where $k\left(\boldsymbol{x}\right)$ is some function $k\,:\,\mathbb{R} \to \mathbb{R}$. For constant k this is equivalent to:

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

The choice of $f\left(\boldsymbol{x}\right)$ is arbitrary and was chosen to produce interesting solution plots, but any $f\left(\boldsymbol{x}\right)$ should show similar convergence properties. This differential equation is defined on the following domain with Dirichlet boundary conditions:

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

$$\partial\Omega = 0$$

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

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

where the i'th entry of $\boldsymbol{A}\boldsymbol{u}$, $\left(\boldsymbol{A}\boldsymbol{u}\right)_i$ is given by:

$$\left(\boldsymbol{A}\boldsymbol{u}\right)_i = \frac{1}{h^2}\left(-k_{i-\frac{1}{2}}u_{i-1} + \left(k_{i-\frac{1}{2}} + k_{i+\frac{1}{2}}\right)u_i - k_{i+\frac{1}{2}}u_{i+1}\right)$$

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

Note that $k\left(x\right)$ is discretized on the *midpoints* of the grid nodes instead of at the gridpoints. This preserves the SPD structure of the system.

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

1. Generate an initial guess of 0, $\boldsymbol{u}_0 = \boldsymbol{0}$.
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*, $\boldsymbol{P} = -\boldsymbol{A}_{ff}^{-1}\boldsymbol{A}_{fc}$.
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 grids of two types were generated via random perturbation. These were generated for different problems, and their types are:

- poisson problem with a constant coefficient of 1 ($\Delta u = f$)
- variable coefficient problems ($\nabla \cdot (k(x) \nabla u) = f$)

2.1 Poisson with unitary constant coefficient

For the constant-coefficient poisson problem, 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.

For this case, 16 convergence factors $\times$ 6 probability trials $\times$ 1000 random iterations = 96000 random grids were created.

2.2 Variable-Coefficient Poisson

The process of generating the variable coefficient grids is very similar to the constant case. Again, several "reference" grids were generated according to different coarsening factors:

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

For the sake of time, only integer values of r were used to create the grids. Using the same p values as above, 3000 random grids were permuted for each value of p. For each grid, a random function for k is selected from the set:

$$k(x) = \begin{cases} \alpha & 0 < \alpha < 10 \\ \text{rand}()\,(\alpha + 1) & 0 < \alpha < 10 \\ \alpha \cos(\pi x \beta) + \gamma & 0 < \alpha < 10, 0 < \beta < 10, 0 < \gamma < 10 \\ \left(\sum_{i=1}^{5} \alpha_i x^i\right) + 0.01 & -10 < \alpha < 10 \end{cases}$$

where the various coefficient values are randomly chosen so that the function of $k(x)$ is strictly positive. A total of 144,000 of these variable coefficient grids were generated, which equates to a total grid count of 240,000. The code used to generate these grids is given in `grids/gen_grids.py`, and with slight modification can produce the unitary coefficient poisson grids as well.

3 CNN Model

In an attempt to predict both optimal Jacobi weight and factor of convergence given an arbitrary grid, a CNN of the architecture described below 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. An overview is described below:

1. 6 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 two input layers 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. 6 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. 6 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 $N \times 10 \times 15$ to $N \times 1 \times 150$.
6. Fully connected layer with input size 150, output size 132. ReLU activation function.
7. Fully connected layer with input size 132, output size 113. ReLU activation function.
8. Fully connected layer with input size 113, output size 95. ReLU activation function.
9. Fully connected layer with input size 95, output size 76. ReLU activation function.
10. Fully connected layer with input size 76, output size 57. ReLU activation function.
11. Fully connected layer with input size 57, output size 39. ReLU activation function.
12. Fully connected layer with input size 39, output size 20. ReLU activation function.
13. Fully connected layer with input size 20, output size 1. ReLU activation function.

3.1 Training

The previously generated set of $240,000$ 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 $N \times 2 \times n$ tensor, with N being the number of grids and $n = 31$ being the grid size. The dimension of 2 in the second index allows the CNN to receive both the *grid* and the *coefficient values* as input. The tensor representation was formed by means of assigning 1 to coarse points and -1 to fine points:

$$T_{i1j} = \begin{cases} 1 & \text{point } j \text{ in grid } i \text{ is coarse} \\ -1 & \text{point } j \text{ in grid } i \text{ is fine} \end{cases}$$

$$T_{i2j} = k\left(\frac{j}{n}\right)$$

Where $k(x)$ is the coefficient function. Note that even though this is discretized on the midpoints when the system is created, the CNN still receives the values of the function evaluated on the grid points. This is to ensure that the input grid and the coefficients have the same shape.

Model	Metric	Dataset	Value
Jacobi Weight	MSE	Training	0.0018331280443817377
Jacobi Weight	MSE	Testing	0.0018395978258922696
Jacobi Weight	L1	Training	0.029253767424938727
Jacobi Weight	L1	Testing	0.029271099756634424
Convergence Factor	MSE	Training	0.001483894418925047
Convergence Factor	MSE	Testing	0.0015170895494520664
Convergence Factor	L1	Training	0.023908496504070377
Convergence Factor	L1	Testing	0.023865242609902033

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

The input weights and convergence factors 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`.

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 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. Final MSE/L1 loss values are given in Table 1.

4 Discrete Grid Optimization

In an attempt to find the most "convergent" grid, an implementation of the *simulated annealing* optimizer was used to obtain the grid that minimizes both the convergence rate and C/F ratio (number of points on the coarsest level divided by the total number of grid points). To minimize both metrics, each iteration of the simulated annealing algorithm alternates between optimizing the convergence rate, or the C/F ratio. Thus, each pass steps either towards a more optimal C/F ratio, or more optimal convergence rate. An overview is given in Algorithm 1. The algorithm itself and helper methods are adapted from those in [1].

As a sanity check, the optimizer was first run on *only* the convergence factor (Fig 5) and the C/F ratio (Fig 6). An example of optimizing both can be seen in Figure 7, though the final output is wildly dependant on the random input guess. More tweaking to the parameters and the algorithm itself is likely needed.

References

[1] D. Bertsimas and J. Tsitsiklis, *Simulated annealing*, Statistical Science, 8 (1993), pp. 10–15.

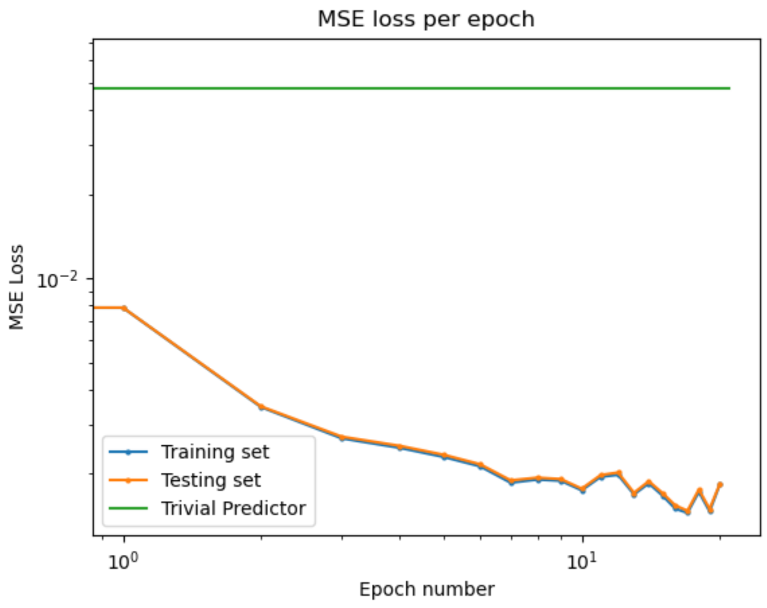
Algorithm 1 Simulated annealing

```
function ANNEAL
    $\boldsymbol{G} \leftarrow$ generate_random_grid()          ▷ Generate random grid and evaluate obj. function
    $\boldsymbol{\phi}_G \leftarrow \phi(\boldsymbol{G})$
    for $k \leftarrow 1 \ldots k_{\max}$ do
        $t := T(k/k_{\max})$          ▷ Compute current temperature value
        $\boldsymbol{S} :=$ random_perturb($\boldsymbol{G}$)          ▷ Generate a random next step, and evaluate obj. function
        $\boldsymbol{\phi}_S := \phi(\boldsymbol{S})$
        $i := k \mod 2$          ▷ Alternate minimizing each metric
        if $P(\boldsymbol{\phi}_G, \boldsymbol{\phi}_S, t, i) \geq$ random(0,1) then
            $\boldsymbol{G} \leftarrow \boldsymbol{S}$          ▷ Randomly take the next step according to the temperature
            $\boldsymbol{\phi}_G \leftarrow \boldsymbol{\phi}_S$
        end if
    end for
end function

function $\phi(\boldsymbol{G})$          ▷ Objective function to minimize
    $c :=$ CNN_conv($\boldsymbol{G}$)
    $w := |\text{C points in } \boldsymbol{G}| \,/\, |\text{total points in } \boldsymbol{G}|$
    return $\begin{bmatrix} c & w \end{bmatrix}^T$
end function

function T($r$)          ▷ Computes temperature of current iteration
    return $\frac{1}{10\log(r)}$
end function

function P($e, e', T, i$)          ▷ Computes probability of switching grids
    return $\exp\left(-\left(\boldsymbol{e}'_i - \boldsymbol{e}_i\right)/T\right)$
end function
```



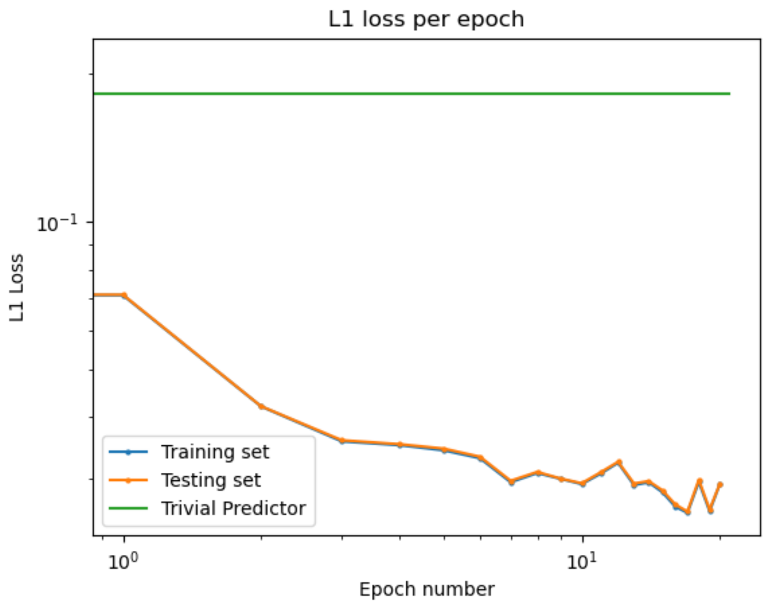
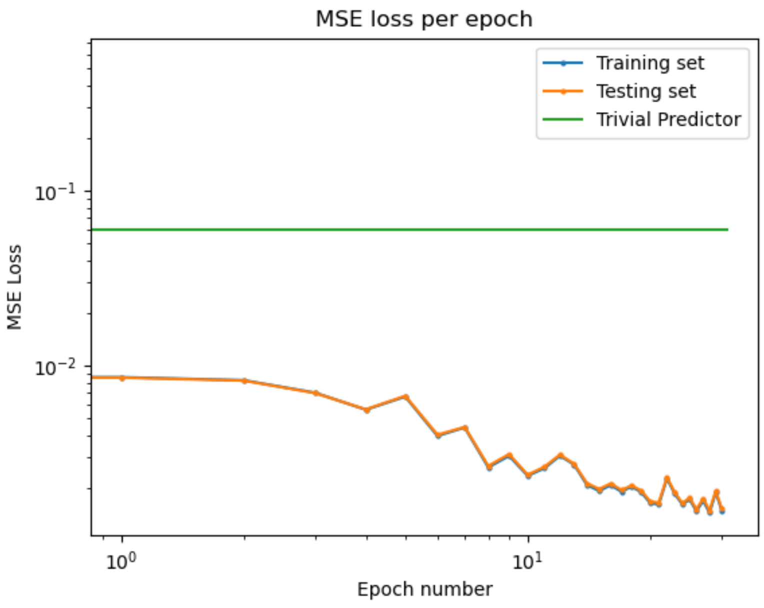


Figure 1: MSE, L1 loss per training iteration for Jacobi weights



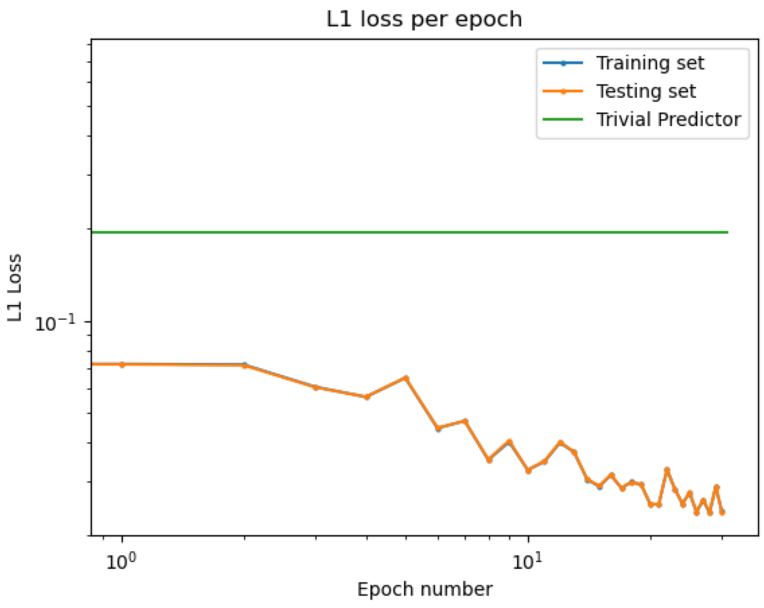
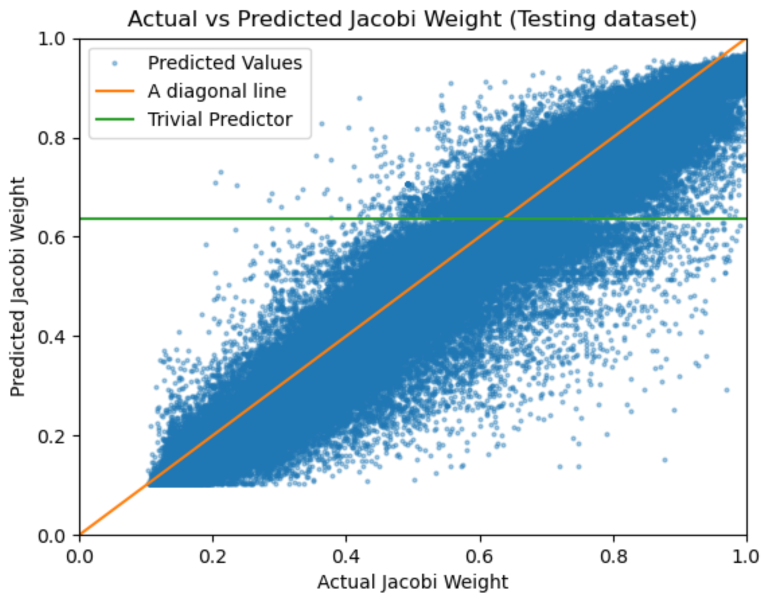


Figure 2: MSE, L1 loss per training iteration for convergence factor



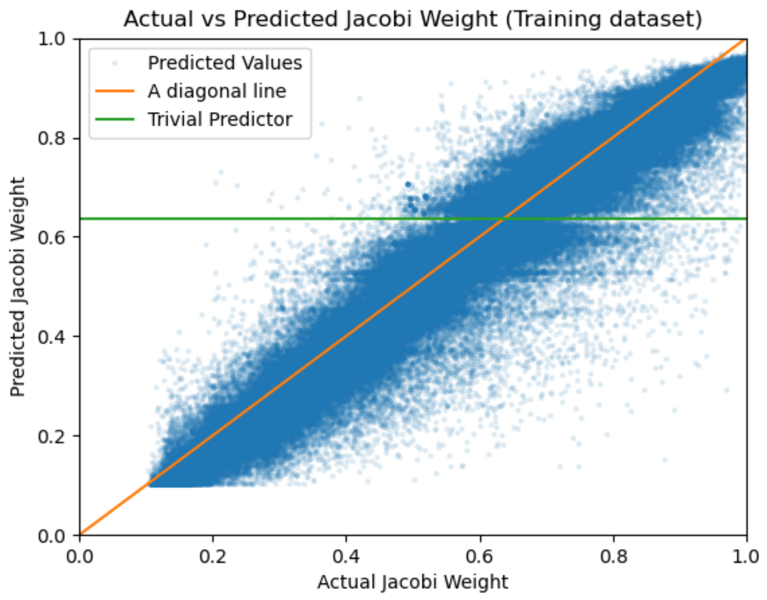
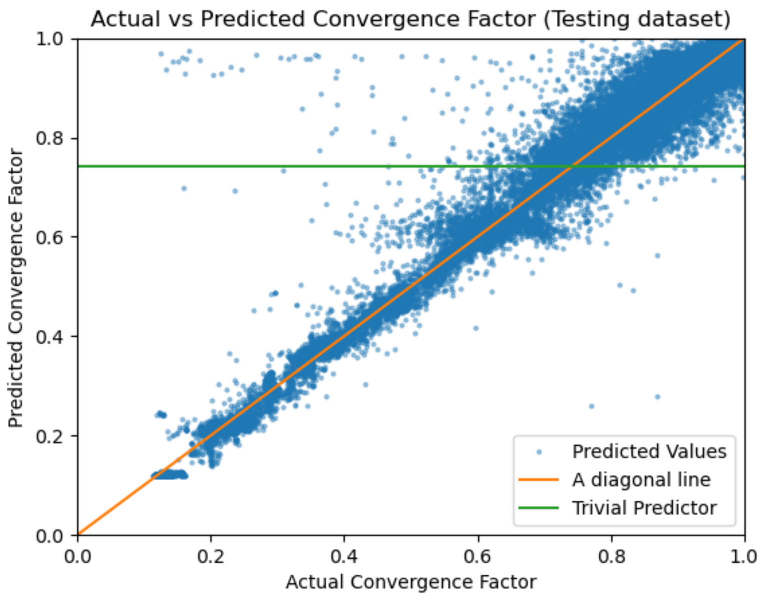


Figure 3: Predicted vs Actual Jacobi Weights



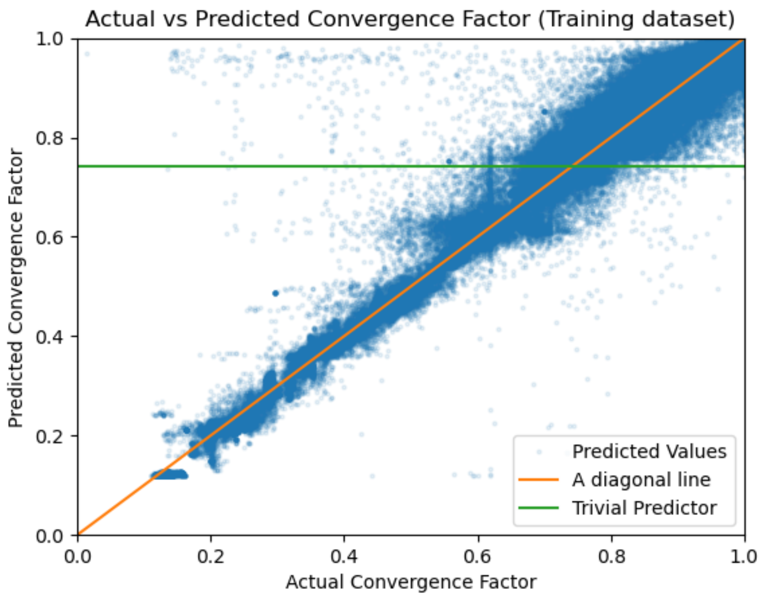
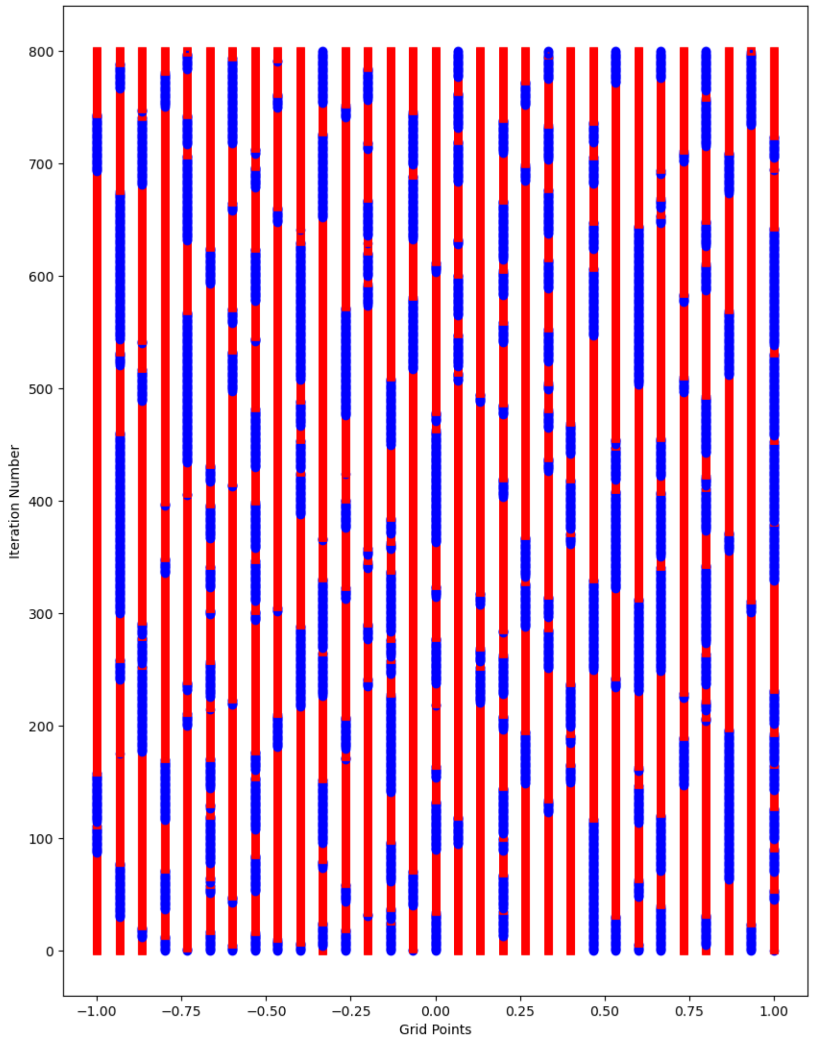


Figure 4: Predicted vs Actual Convergence Factors



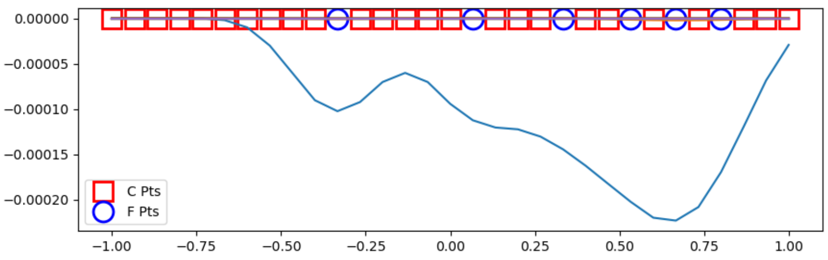


Figure 5: (Top) Time evolution of grid obtained by optimizing CNN output on a normal poisson problem, initial grid on bottom and final grid on top. (Bottom) convergence behaviour of the resulting grid. This was computed using simulated annealing, though only optimizing on the *convergence factor*. True convergence factor: 0.14123, predicted factor: 0.12423. C/F ratio: 0.80645. Elapsed time: 1.42712 seconds.

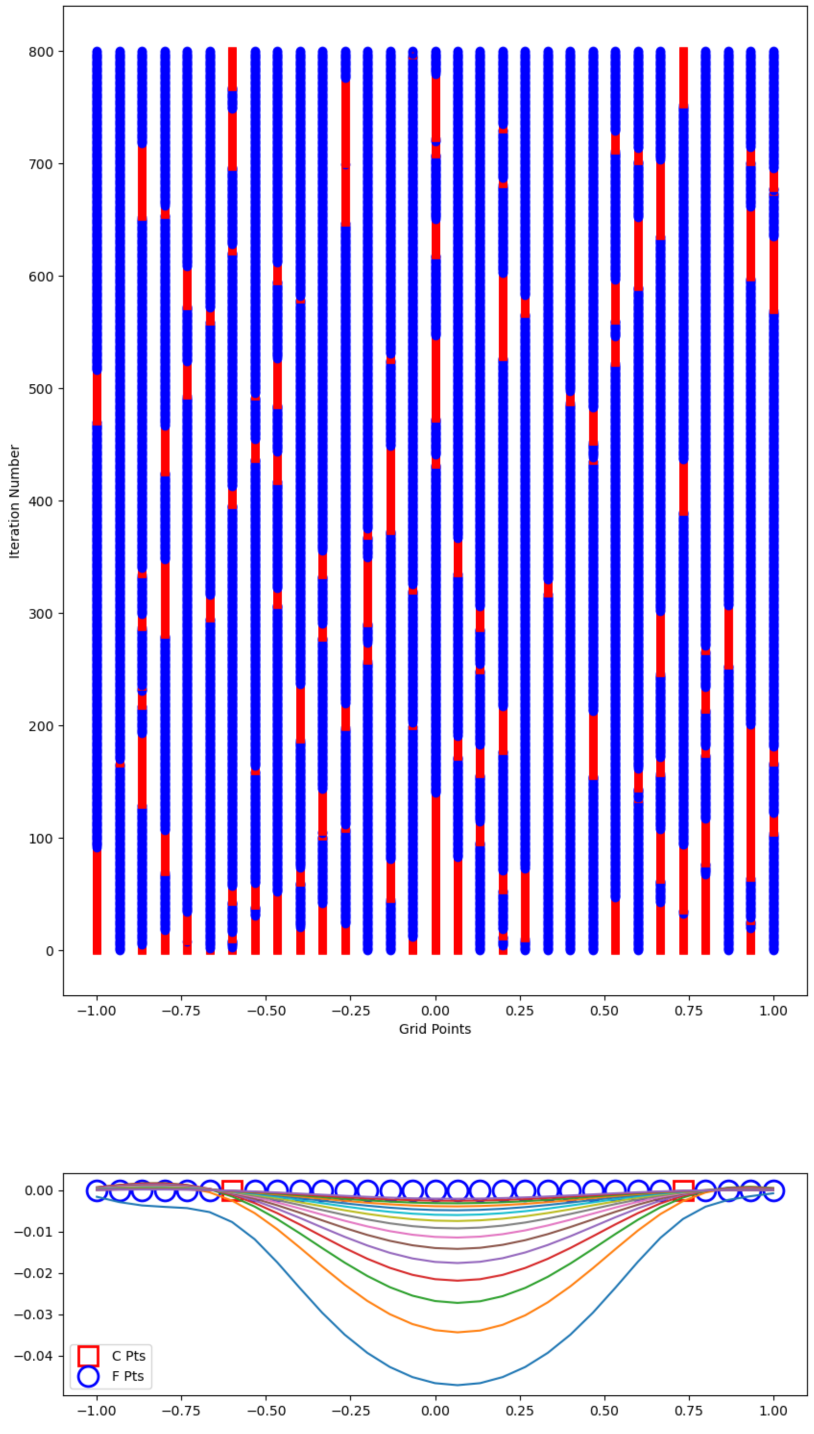
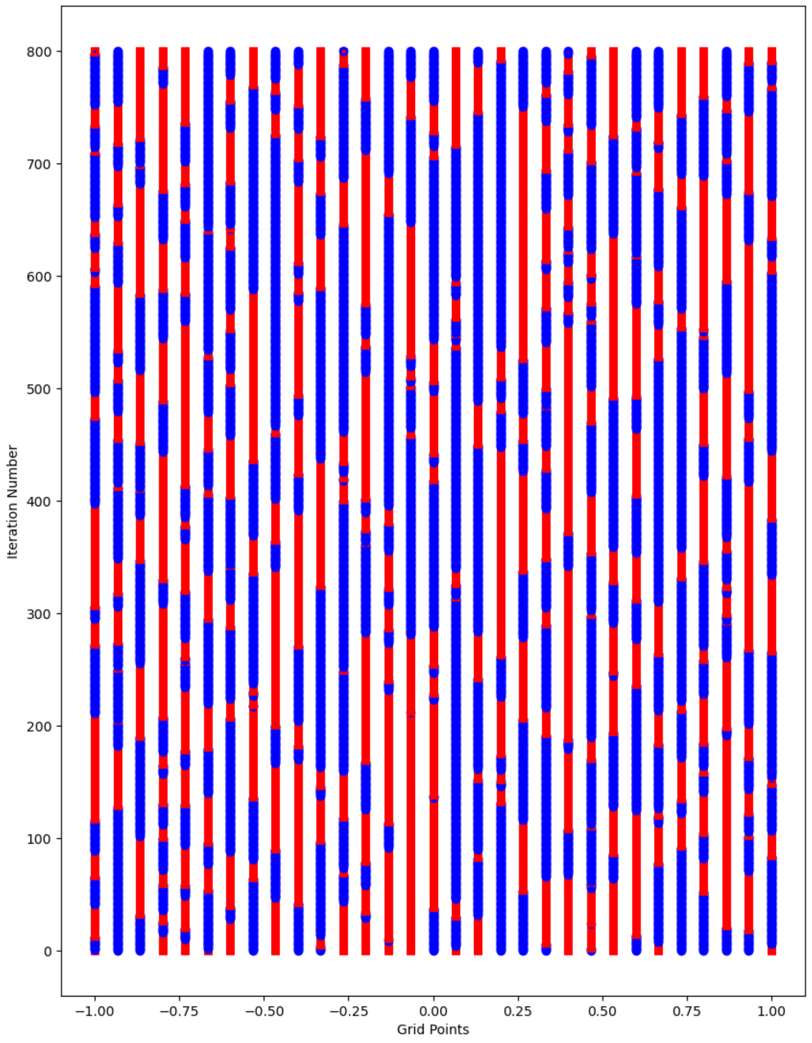


Figure 6: (Top) Time evolution of grid obtained by optimizing CNN output on a normal poisson problem, initial grid on bottom and final grid on top. (Bottom) convergence behaviour of the resulting grid. This was computed using simulated annealing, though only optimizing on the *C/F ratio*. True convergence factor: 0.92223, predicted factor: 0.91534. C/F ratio: 0.064516. Elapsed time: 0.91058 seconds.



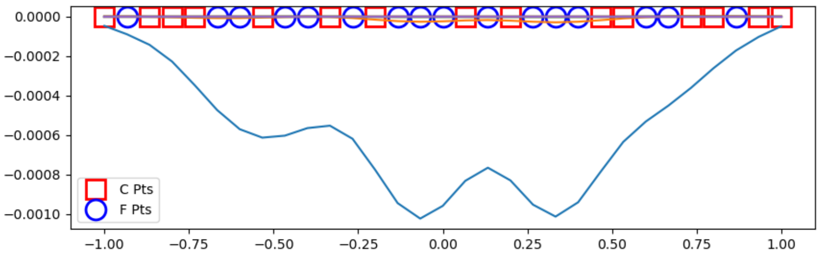
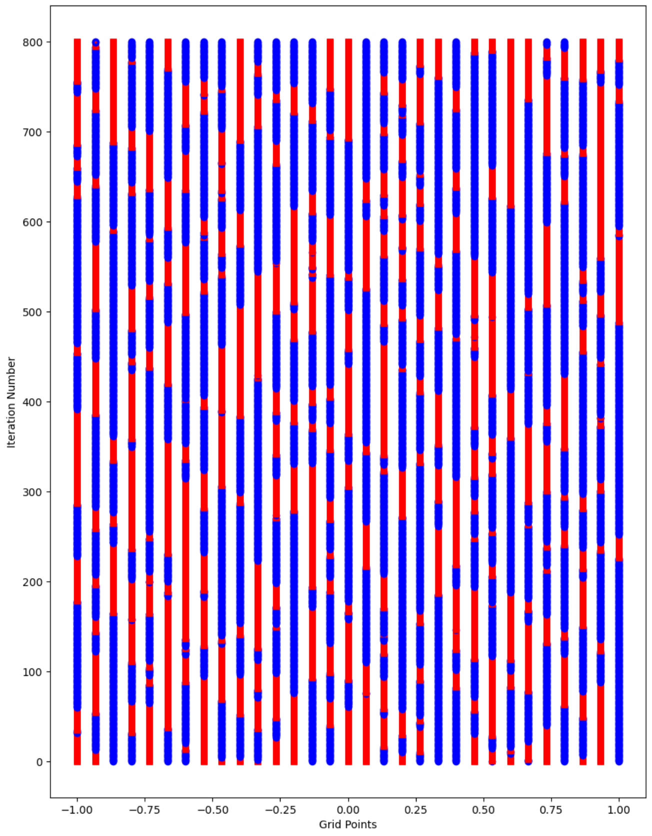
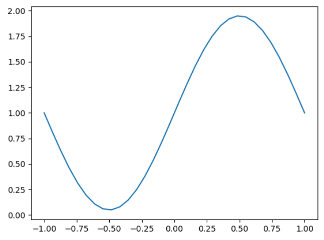


Figure 7: (Top) Time evolution of grid obtained by optimizing CNN output and C/F ratio on a normal poisson problem, initial grid on bottom and final grid on top. (Bottom) convergence behaviour of the resulting grid. This was computed using simulated annealing as described in Algorithm 1. True convergence factor: 0.37479, predicted factor: 0.38923. C/F ratio: 0.48387. Elapsed time: 1.4640 seconds.





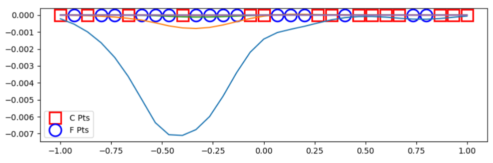


Figure 8: (Top) Time evolution of grid obtained by optimizing CNN output on the variable coefficient poisson problem, initial grid on bottom and final grid on top. (Left) Variable coefficient function, $k(x) = 1 + 0.95\sin(\pi x)$. (Right) final convergence behaviour. True convergence factor: 0.47893, predicted factor: 0.85469. Elapsed time: 1.47459 seconds.