Summary Report on Implementation of Neural Network for Discrete Inverse Conductivity Problem

August 2025

1 Introduction

This report details the implementation of a neural network (NN) approach to solve the discrete inverse conductivity problem on a square lattice, as described in [1]. The goal is to recover the conductivity profile from Dirichlet-to-Neumann (DtN) data using an interpretable NN, where the trained weights in the second layer directly represent the conductivities. The implementation addresses issues found in [1] such as matrix dimension errors, weight constraints and so on. The code is organized into four main files: `rect_grid.py`, `rect_NN_class.py`, `rect_NN_helper.py`, and `solve_linear.py`, with training orchestrated in `NN_wgpu.ipynb`. This report emphasizes the logical flow of the code in python, the rationale behind implementation choices, and their alignment with the paper's methodology, ensuring clarity for readers to follow the code.

2 Logical Flow and Implementation Rationale

2.1 Grid Setup and Forward Problem (`rect_grid.py`)

The implementation begins by modeling the square lattice as a network of nodes and edges, excluding corner nodes, to match the problem setup in Section 2 of [1]. The `GridStructure` class creates an $n \times n$ grid of interior nodes (indices 0 to $n^2 - 1$) and $4n$ boundary nodes (indices n^2 to $n^2 + 4n - 1$), ordered cyclically to align with the DtN map (Eq. 1-2).

```
def create_nodes(self):
    current_index = 0
    for i in range(1, self.n + 1):
        for j in range(1, self.n + 1):
            self.nodes[(i, j)] = Node(i, j, self.n, current_index, is_boundary=
                False)
            self.index_to_coords[current_index] = (i, j)
            current_index += 1
    # Boundary nodes: top, right, bottom, left
    for j in range(1, self.n + 1):
        self.nodes[(0, j)] = Node(0, j, self.n, current_index, is_boundary=True
            )
        self.index_to_coords[current_index] = (0, j)
        current_index += 1
    # ... (similar for right, bottom, left)
```

The forward problem, implemented in `solve_forward_problem`, solves the discrete conductivity equation (Eq. 3-5) to compute potentials and Neumann data given Dirichlet data. It uses sparse matrices (`scipy.sparse`) for efficiency, with a custom solver from `solve_linear.py` that was used to validate the data generation process.

```
def solve_forward_problem(self, dirichlet_data):
    A = lil_matrix((interior_nodes, interior_nodes))
    b = np.zeros(interior_nodes)
    for i in range(1, self.n + 1):
        for j in range(1, self.n + 1):
            node_idx = self.get_node_index(i, j)
```

```
            matrix_idx = interior_map[node_idx]
            sum_gamma = 0.0
            for neighbor_idx in node.neighbors:
                edge = tuple(sorted([node_idx, neighbor_idx]))
                gamma = self.conductivities[edge]
                sum_gamma += gamma
                if neighbor_idx in boundary_indices:
                    b[matrix_idx] += gamma * dirichlet_data[neighbor_idx]
                else:
                    A[matrix_idx, interior_map[neighbor_idx]] = -gamma
            A[matrix_idx, matrix_idx] = sum_gamma
    interior_potentials = spsolve(A, b)
    # ... (assign potentials and compute Neumann data)
```

Rationale: The cyclic boundary indexing simplifies DtN data handling, and the sine-based conductivity assignment (in `generate_conductivity`) ensures positive, physically meaningful values. Sparse matrices and the custom solver are used with the hope to handle larger grids efficiently. Note that there are other additional functions included in (`rect_grid.py`) beyond just the two functions mentioned above. Each function has been well commented for the user to understand.

2.2 Neural Network Architecture (`rect_NN_class.py`)

The `SquareEITNN` class implements the three-layer feed-forward NN from Section 3 of [1]: - **Input Layer**: $8n$ neurons ($4n$ Dirichlet, $4n$ Neumann data). - **Hidden Layer**: $n^2 + 8n$ neurons (n^2 interior potentials, $4n$ copied Dirichlet, $4n$ copied Neumann). - **Output Layer**: $n^2 + 4n$ neurons (Kirchhoff's law for interior, Neumann conditions for boundary).

2.2.1 $W^{(1)}$ Matrix Structure

The $W^{(1)}$ matrix, as clarified in Issue 2 of [2], maps input data to the hidden layer. It is structured as a block matrix:

$$W^{(1)} = \begin{bmatrix} A & 0 \\ I_{4n} & 0 \\ 0 & I_{4n} \end{bmatrix}$$

- $A \in \mathbb{R}^{n^2 \times 4n}$: Trainable weights mapping Dirichlet data to interior potentials (discrete Green's kernel).
- I_{4n}: Identity matrix copying Dirichlet data.
- I_{4n}: Identity matrix copying Neumann data.
- 0: Zero matrices for unused connections.

```
def _initialize_W1_blocks(self):
    self.W1[:self.interior_size, :4*self.n] = torch.randn(self.interior_size,
        4*self.n)
    self.W1[self.interior_size:self.interior_size + 4*self.n, :4*self.n] =
        torch.eye(4*self.n)
    self.W1[self.interior_size + 4*self.n:, 4*self.n:] = torch.eye(4*self.n)
    self.W1[:self.interior_size, 4*self.n:] = 0.0
    self.W1[self.interior_size:self.interior_size + 4*self.n, 4*self.n:] = 0.0
    self.W1[self.interior_size + 4*self.n:, :4*self.n] = 0.0
```

Calculation: For an input vector $x = [D, N]^T \in \mathbb{R}^{8n}$, where $D \in \mathbb{R}^{4n}$ is Dirichlet data and $N \in \mathbb{R}^{4n}$ is Neumann data, the hidden layer output h is:

$$h = W^{(1)}x = \begin{bmatrix} A & 0 \\ I_{4n} & 0 \\ 0 & I_{4n} \end{bmatrix} \begin{bmatrix} D \\ N \end{bmatrix} = \begin{bmatrix} AD \\ D \\ N \end{bmatrix}$$

- $AD \in \mathbb{R}^{n^2}$: Potentials at interior nodes.

- $D \in \mathbb{R}^{4n}$: Copied Dirichlet data.
- $N \in \mathbb{R}^{4n}$: Copied Neumann data.

Rationale: This structure ensures the hidden layer contains the predicted interior potentials and preserves boundary data, aligning with Eq. 6-7 and addressing an issue that was seen with some ambiguous definition of $W^{(1)}$ by clearly separating the components.

2.2.2 $W^{(2)}$ Matrix and Weight Management

It also becomes easy to subdivide $W^{(2)}$ into multiple block matrices to keep consistent with the expected output depending on the hidden layer we obtain. We can sub-divide $W^{(2)}$ as the following block matrix.

$$W^{(2)} = \begin{bmatrix} A & B & C \\ D & E & I_{4n} \end{bmatrix}$$

where:

- $A \in \mathbb{R}^{n^2 \times n^2}$: Contains information for conductivities for interior nodes, with symmetric weights $w_{ij}^{(2)} = w_{ji}^{(2)}$ for neighboring nodes i, j and $w_{ii}^{(2)} = -\sum_{j \in N(i)} w_{ij}^{(2)}$.
- $B \in \mathbb{R}^{n^2 \times 4n}$: Encodes conductivity information between neighboring interior and boundary nodes.
- $C \in \mathbb{R}^{n^2 \times 4n}$: Zero matrix, as Neumann data does not contribute to interior equations.
- $D \in \mathbb{R}^{4n \times n^2}$: Encodes conductivity information between neighboring interior and boundary nodes.
- $E \in \mathbb{R}^{4n \times 4n}$: Encodes symmetric condition for boundary nodes.
- $I_{4n} \in \mathbb{R}^{4n \times 4n}$: Identity matrix, copying Neumann data for boundary conditions. Including an identity matrix here fixes the issue that was identified in the main paper by the inclusion of the weights -1 instead of 1.

$$W^{(2)} = \left[\begin{array}{c|c|c} A & B & C \\ \hline D & E & I_{4n} \end{array}\right]$$

where the blocks are defined as follows: - The left submatrix $(n^2 + 4n) \times (n^2 + 4n)$ is symmetric and is composed of blocks A, B, D, and E, with a orange squared diagonal representing the diagonal elements, where each diagonal term $w_{ii}^{(2)} = -\sum_{j \neq i} w_{ij}^{(2)}$ (the negative sum of non-diagonal entries in the row).

$$W^{(2)} = \left[\begin{array}{cccc|ccc|ccc} \square & \star_{a1} & 0 & \star_{a2} & 0 & \bullet & 0 & 0 & 0 & 0 \\ \star_{a1} & \square & \star_{a3} & 0 & 0 & 0 & \bullet & 0 & 0 & 0 \\ 0 & \star_{a3} & \square & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \star_{a2} & 0 & 0 & \square & \bullet & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & \bullet & \square & 0 & 0 & 1 & 0 & 0 \\ \bullet & 0 & 0 & 0 & 0 & \square & 0 & 0 & 1 & 0 \\ 0 & \bullet & 0 & 0 & 0 & 0 & \square & 0 & 0 & 1 \end{array}\right]$$

Notes:

- The matrix diagram above is not to scale. In the upper triangular part of block A, not all the elements are trainable, some are 0 too, as shown in the above representation. Note that the representation does not exactly satisfy the dimensions of $W^{(2)}$ matrix, so it is included there just for visual purposes to get a feel of what $W^{(2)}$ matrix would look like.
- The matrix is divided into a $(n^2 + 4n) \times (n^2 + 4n)$ left submatrix (blocks A, B, D, E) and a $(n^2 + 4n) \times 4n$ right submatrix (blocks C and I_{4n}).
- The symbols $\square$ along the diagonal of the left submatrix (spanning A and E) represent diagonal elements, where each $w_{ii}^{(2)} = -\sum_{j \neq i} w_{ij}^{(2)}$ for $j < n^2 + 4n$, enforcing Kirchhoff's law. It can be seen that the diagonal elements of the sub-matrix E is just the negative of the only non-zero element present in the same row of block D.

- In block A ($n^2 \times n^2$), trainable weights above the diagonal are marked with red stars ($\star_{a1}$, $\star_{a2}$, etc.), and their symmetric fixed counterparts below the diagonal are in blue ($\star_{a1}$, $\star_{a2}$, etc.), indicating they are derived from the trainable weights. In every iteration, we train the trainable elements of the upper traingle and enforce these symmetry conditions later on but before beginning the next iteration.

- Block B ($n^2 \times 4n$) can have at most one $\bullet$ per row, representing that an interior node can have a maximum of one boundary neighbor. Also, Block D has one single trainable element$\bullet$ per row indicating that every boundary node has exactly one interior node.

- The block E ($4n \times 4n$) is a diagonal matrix with elements $\square$, enforcing the symmetry conditions for the connections of the boundary nodes.

- Block C ($n^2 \times 4n$) is a zero matrix, and block I_{4n} ($4n \times 4n$) is the identity matrix, copying Neumann data.

The $W^{(2)}$ matrix encodes conductivities and enforces Kirchhoff's law and Neumann conditions (Eq. 8-9). The implementation in code uses three matrices to do so:

- **Beta**: A parameter matrix storing raw weights, initialized with positive random values for edges $\{i, j\}$ where $i < j$.
- **W2_fixed**: Stores the symmetric counterparts of trained weights, updated after training to enforce symmetry.
- **W2_dynamic**: Used in forward propagation, combining fixed and transformed weights using torch.where() functionality.

```
def _initialize_W2(self):
    for i in range(self.interior_size):
        node_i = self.grid.get_node_by_index(i)
        neighbor_indices = [j for j in node_i.neighbors if i < j < self.
            interior_size + 4*self.n]
        weights = torch.abs(torch.randn(len(neighbor_indices)))
        for k, j in enumerate(neighbor_indices):
            self.Beta[i,j] = weights[k]
            self.Beta[j,i] = weights[k]
            self.W2_fixed[j,i] = weights[k]
        self.Beta[i,i] = -torch.sum(torch.tensor([self.Beta[i,j] for j in ...])
            )
    for i in range(4*self.n):
        self.Beta[self.interior_size + i, self.neumann_copy_start + i] = 1.0
```

The `transform` function ensures positive conductivities. Although, it has been theoretically established that a 0 loss is obtained and in such a case all the trained weights of $W^{(2)}$ are positive values, allowing negative values even for just training purposes creates numerical instability. Hence, we use the transform function to change the trainable parameters (parameters in Beta $\in \mathbb{R}$) to a positive value when used in W2_dynamic.

```
def transform(self, x):
    return torch.arctan(x) + torch.pi/2
```

In forward propagation, `W2_dynamic` is computed using `torch.where` to apply the transformation to trainable weights. In the boolean matrix `W2_mask`, we have the entries as False in only those positions that we want to train in the Beta matrix.

```
self.W2_dynamic = torch.where(self.W2_mask, self.W2_fixed, torch.arctan(self.
    Beta) + torch.pi/2)
```

The output is computed as $y = W^{(2)}h$, where y represents residuals of Kirchhoff's law (interior nodes) and Neumann conditions (boundary nodes). Ideally, what we want is for all values of y to be 0. So, the aim of the model is to minimize the loss function (Eq. 11). In code, we differentiate the loss into interior loss and boundary loss in order to keep track of how boundary and interior loss are evolving. Since, we have a hyperparameter α, we can alter it mid-training to make the model focus on minimizing either interior or boundary loss if one becomes larger than the other. If the interior loss is larger than boundary loss, we increase the value of alpha so as to tell the model to focus on minimizing the interior loss whereas if the boundary loss is larger, we decrease alpha. It is seen that the interior loss is usually larger, hence we start with a smaller alpha and increase it as we proceed with the training.

```
def loss_function(output, grid, alpha=1.0):
    interior = output[:, :-4*grid.n]
    boundary = output[:, -4*grid.n:]
    interior_loss = alpha * torch.sum(interior ** 2) / (2 * batch_size)
    boundary_loss = torch.sum(boundary ** 2) / (2 * batch_size)
    total_loss = interior_loss + boundary_loss
    return total_loss, interior_loss, boundary_loss
```

Gradients are computed manually in `assign_gradients`. Despite us working with python and despite the presence of extensive functions like Autograd, we do not use those. The reason being, Autograd works fine only when the trainable parameters are independent(Maybe there could be a better term for this instead of "independent", but anyways). What does it mean? Given a weight matrix(i.e. matrix with trainable parameters), Autograd is able to compute the gradients properly if one element of the matrix does not depend on the other. Autograd uses the ideas of computational graph to individually track the effect of every parameter on the loss. But since we have a symmetric W2 matrix and the same parameter influences the loss in different ways, Autograd is not able to take into account our special case. Hence, we compute the gradients manually using `assign_gradients` function.

```
def assign_gradients(self, x_data, h, y, alpha):
    dL_dy = torch.zeros_like(y)
    dL_dy[:, :self.n**2] = alpha * y[:, :self.n**2] / batch_size
    dL_dy[:, self.n**2:] = y[:, self.n**2:] / batch_size
    dW2 = dL_dy.T @ h
    w2_indices = [(i, j) for i in range(self.n**2) for j in self.grid.
        get_node_by_index(i).neighbors if i < j < self.interior_size + 4*self.n]
    for i, j in w2_indices:
        grad[(i, j)] = dW2[i, j] + dW2[j, i] - dW2[i, i] - dW2[j, j]
        self.W2__grad[i, j] = grad[(i, j)] / (1 + self.Beta[i, j]**2)
    dW1 = self.W2_dynamic.T @ dL_dy.T @ x_data
```

Parameters ($W^{(1)}$ and `Beta`) are updated using a custom Adam optimizer with gradient clipping. Gradient clipping is a technique used to ensure that during the loss minimization, we do not take a very large steps to avoid the problem of stalling loss. So, we cap the L2 norm of the gradient to `max-grad-norm` and normalize the gradients everytime it exceeds the value assigned in `max-grad-norm`:

```
def adam_with_grad_clip(param, grad, alpha, m_t, v_t, beta1, beta2, eps,
    time_step, max_grad_norm=1.0):
    grad_norm = torch.norm(grad, p=2)
    if grad_norm > max_grad_norm:
        grad = grad * max_grad_norm / grad_norm
    m_t1 = beta1 * m_t + (1 - beta1) * grad
    v_t1 = beta2 * v_t + (1 - beta2) * (grad ** 2)
    m_corr = m_t1 / (1 - beta1 ** time_step)
    v_corr = v_t1 / (1 - beta2 ** time_step)
    delta = alpha * m_corr / (torch.sqrt(v_corr) + eps)
    param = param - delta
    return param, m_t1, v_t1
```

After each update, `symmetrize_W2_after_training` enforces symmetry in `W2_fixed`:

```
def symmetrize_W2_after_training(self):
    for i in range(self.interior_size):
```

```
        neighbor_indices = [j for j in self.grid.get_node_by_index(i).neighbors
            if i < j < self.interior_size + 4*self.n]
        for k, j in enumerate(neighbor_indices):
            self.W2_fixed[i,j] = self.transform(self.Beta[i,j])
            self.W2_fixed[j,i] = self.transform(self.Beta[i,j])
        self.W2_fixed[i,i] = -torch.sum(torch.tensor([self.W2_fixed[i,j] for j
            in ...]))
```

Rationale: The `Beta` matrix allows training a single weight per edge, with `transform` ensuring positive conductivities (Issue 6 in [2]). `W2_dynamic` enables forward propagation with updated weights, while `symmetrize_W2_after_training` enforces $w_{ij}^{(2)} = w_{ji}^{(2)}$ and conservation, addressing Issue 4 and ensuring physical consistency. Manual gradient computation accounts for symmetry, and the custom Adam optimizer stabilizes training for larger grids.

2.3 Training and Optimization (`rect_NN_helper.py`, `NN_wgpu.ipynb`)

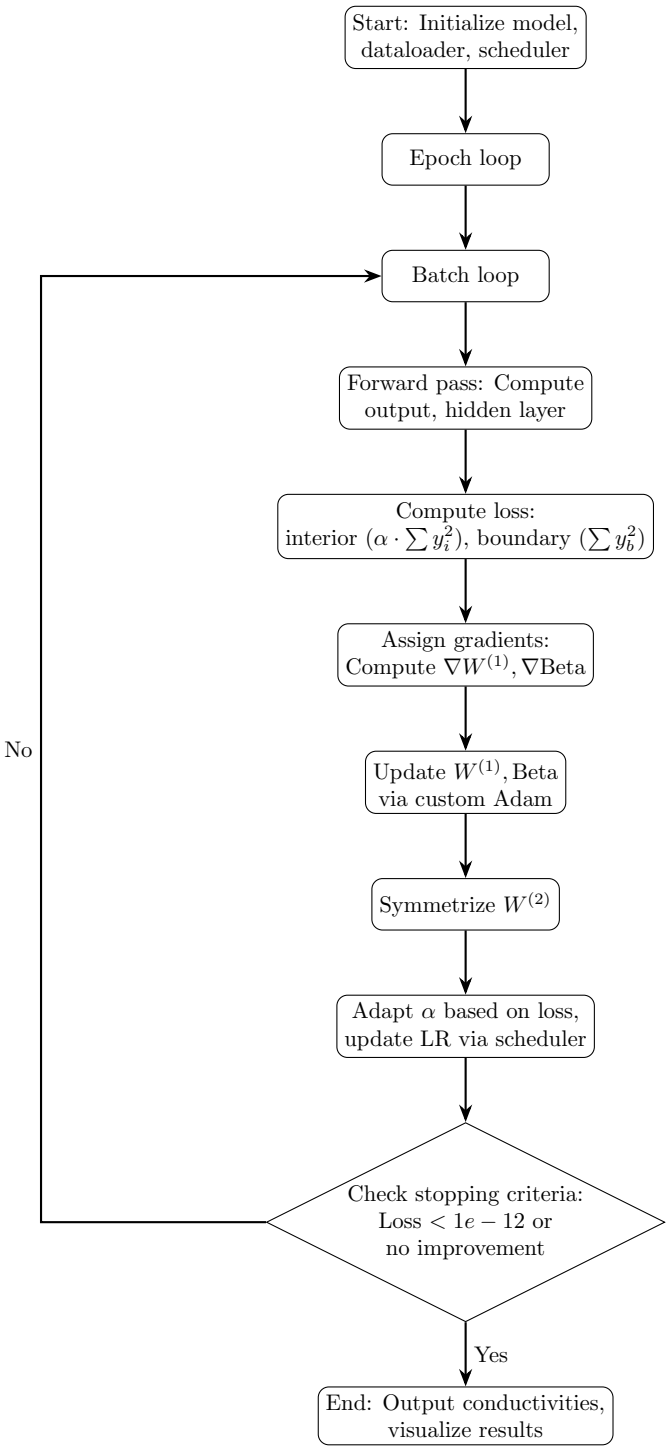
The training loop in `train_adaptive` (in `NN_wgpu.ipynb`) orchestrates the optimization process, using a custom cosine annealing scheduler with warm restarts to adjust the learning rate dynamically:

```
class CustomCosineAnnealingWarmRestarts:
    def __init__(self, T_0, initial_lr, T_mult=1, eta_min=1e-8, eta_max_factor
        =0.08):
        self.T_0 = T_0
        self.base_lr = initial_lr
        self.eta_min = eta_min
        self.eta_max_factor = eta_max_factor
    def step(self):
        if self.t >= self.T_cur:
            self.cycle += 1
            self.base_lr *= self.eta_max_factor
            self.T_cur = self.T_0 * (self.T_mult ** self.cycle)
            self.t = 0
        lr = self.eta_min + 0.5 * (self.base_lr - self.eta_min) * (1 + np.cos(
            np.pi * self.t / self.T_cur))
        self.t += 1
        return lr
```

The training loop adapts α to balance interior and boundary losses and adjusts β_1, β_2 if the learning rate becomes too low:

```
def train_adaptive(num_epochs, dataloader, model, alpha, learning_rate, grid,
    ...):
    scheduler = CustomCosineAnnealingWarmRestarts(T_0=cycle_length, initial_lr=
        learning_rate, ...)
    for epoch in range(num_epochs):
        for batch_x in dataloader:
            output, hid = model(batch_x)
            total_loss, interior_loss, boundary_loss = loss_function(output,
                grid, alpha)
            model.assign_gradients(batch_x, hid, output, alpha)
            for state in adam_states:
                new_param, new_m, new_v = adam_with_grad_clip(...)
                param.data = new_param.data
            model.symmetrize_W2_after_training()
            # ... (update alpha, learning rate)
        min_loss = min(interior_loss_epoch, boundary_loss_epoch)
        if min_loss < 1e-9 and alpha < 1.0:
            alpha = 1.0
        # ... (similar for 0.9, 0.8)
        if scheduler.base_lr < 1e-7 and not beta_updated:
            scheduler.reset(new_base_lr=1e-4)
            beta1_current = 0.95
            beta2_current = 0.9999
```

The training loop is visualized in the following flowchart:



Rationale: The cosine annealing scheduler helps escape local minima, crucial for larger grids ($n = 7, 8$), as noted in `NN_wgpu.ipynb`. Adaptive α prioritizes boundary loss initially, then balances with interior loss, aligning with the paper's stability focus (Section 4). Adjusting β_1, β_2 when the learning rate drops prevents stagnation, addressing poor performance for $n = 7$.

2.4 Custom Linear Solver (`solve_linear.py`)

The `linear_solver` function implements LU decomposition for solving the forward problem:

```
def linear_solver(A, b):
    L, U = lu_decomposition(A)
```

```
    y = forward_substitution(L, b)
    x = backward_substitution(U, y)
    return x
```

Rationale: The custom solver ensures compatibility10.0pt robustness for large systems, supporting numerical experiments (Section 5).

3 Relation to the Paper

The implementation realizes the NN architecture in Section 3 of [1], encoding conductivities in $W^{(2)}$ for interpretability (Remark 1). It addresses issues in [2]: - **Issue 2**: Clarified $W^{(1)}$ structure. - **Issue 3**: Corrected $W^{(2)}$ dimensions to $(n^2 + 4n) \times (n^2 + 8n)$. - **Issue 4**: Fixed Neumann weight to $+1$. - **Issue 6**: Detailed weight initialization and symmetrization.

References

[1] E. Beretta, M. Deng, A. Gandolfi, and B. Jin. The discrete inverse conductivity problem solved by the weights of an interpretable neural network. *Journal of Computational Physics*, 538:114162, 2025.

[2] Issues in paper. Manuscript, July 2025.