Report on usage of Poisson CNN prediction model

Neelappagouda V Hiregoudar

August 30, 2022

Contents

Listings

List of Figures

Impact statement by author:

The Poisson equation is one of the most computationally intensive partial differential equations to solve, requiring very expensive iterative methods. The author propose a novel and flexible CNN architecture which can either serve as an alternative to existing iterative methods on Cartesian grids, or augment them. Outperforming existing NN models, the proposed framework can deal with different boundary conditions and domain aspect ratios, without the need for re-training as long as boundary condition types (Dirichlet, Neumann, etc.) are unchanged. The proposed model can enable engineers to run simulations faster, for instance in computational fluid dynamics, where a Poisson equation must to be solved at each time step of the simulation for incompressible flows.

1 Introduction

In this paper [2], the author proposes a novel fully convolutional neural network (CNN) architecture to infer the solution of the Poisson equation on a 2D Cartesian grid with different resolutions given the right-hand side term, arbitrary boundary conditions, and grid parameters. It provides unprecedented versatility for a CNN approach dealing with partial differential equations. The boundary conditions are handled using a novel approach by decomposing the original Poisson problem into a homogeneous Poisson problem plus four inhomogeneous Laplace subproblems.

One of the most important PDEs in engineering is the Poisson equation, expressed mathematically as

$$\nabla^2 \phi = f$$

where f is a forcing term, and ϕ is the variable for which a solution is sought. Appearing in a diverse range of problems, including heat conduction, gravitation, simulating fluid flows, and electrodynamics, the Poisson problem plays a central role in the design of many modern technologies. However, solving the Poisson equation numerically for large problems involving millions of degrees of freedom is only tractable by employing iterative methods.

The principal motivation is to develop a CNN-based Poisson equation solver that does not require re-training to perform inference on inputs with different types of boundary conditions (BCs) within a given range of grid resolutions and sizes in the context of Cartesian grids. In this work, the author presents a novel CNN architecture capable of handling arbitrary right-hand side (RHS) functions f and BCs of a given type on rectangular two-dimensional grids of different aspect ratios having a uniform grid spacing Δ.

"The proposed model is also included in a CFD solver to demonstrate its potential to provide an accurate initial guess (more accurate than the first step of conventional iterative methods) so that the rate of convergence can be dramatically increased."

2 Neural Network Architecture

The methodology involves splitting the original (2D) Poisson problem into one Poisson problem with homogeneous (zero Dirichlet) BCs plus four Laplace problems where each Laplace problem has three homogeneous BCs plus one inhomogeneous Dirichlet or Neumann BC identical to one of the BCs in the original problem on the corresponding boundary. Mathematically the proposed idea can be represented by exploiting the linearity of the Laplace operator

$$\nabla^2(\phi_h + \phi_{BC0} + \phi_{BC1} + \phi_{BC2} + \phi_{BC3}) = f$$

Thus, it is possible to solve the original problem by first solving the Poisson equation with the original RHS but zero BCs to find ϕ_h, then solving Laplace problems for each BC to find ϕ_{BCk} and summing these results.

The proposed NN architecture will be composed of two parts:

- **Homogeneous Poisson NN (HPNN)**: which approximates the solution of the Poisson equation with homogeneous BCs.
- **Dirichlet Boundary Condition NN (DBCNN)**: which solves the Laplace problem given one inhomogenous Dirichlet BC.

First, the DBCNN submodel makes predictions for the four Laplace problems. It applies reflection and rotation operations such that the inhomogeneous boundaries align with the corresponding boundary in the original problem. Then, the HPNN model makes a prediction for the Poisson problem with homogeneous BCs. Finally, these results are summed to obtain the final prediction for a Poisson problem with four inhomogeneous DBCs.

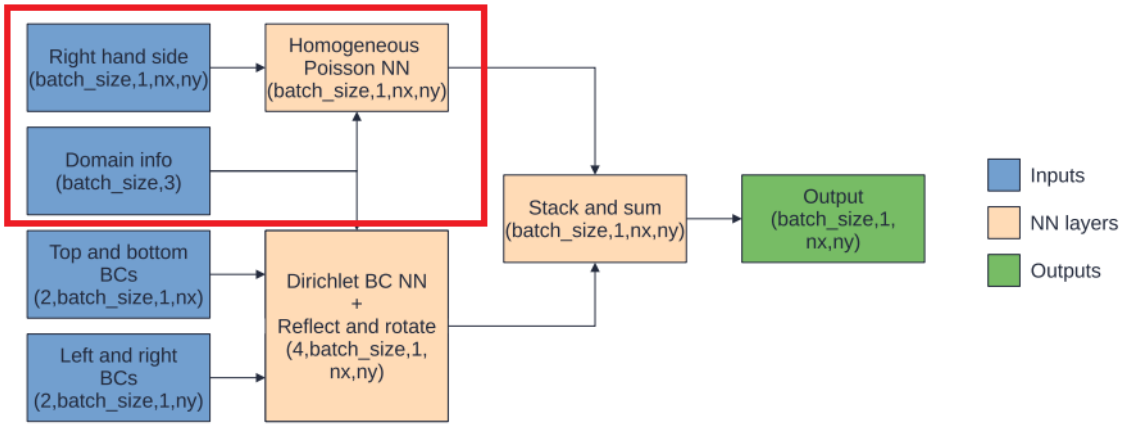

Figure 1: Poisson CNN model

2.1 HPNN

The HPNN model has two inputs: the RHS and the domain information (grid spacing, domain sizes per spatial dimension). Figure 2 provides an overview of the model architecture.

When processing the RHS, first, the data are passed through several convolutions. Then, the computation is split into multiple independent pooling operations. Each applies average pooling of progressively larger pool sizes to capture lower-wavenumber modes and applies further convolutions to the pooled results. Then, the pooled results are upsampled using either a polynomial reconstruction method (in the case of the pooling threads with the largest two pool sizes) or transposed convolutions (for branches with smaller pool sizes). Using polynomial interpolation-based upsampling methods for pooling branches with large pool sizes was observed to reduce artifacts in the output; upsampling, for example, a 2×2 input generated using 128 ×128 pooling from a 200 × 200 source image requires using a stride3 of 128 for the transposed convolution to upsample to the original size, which is excessively large. Finally, all branches' results are summed and divided by the number of branches times the number of channels in each branch.

Several dense layers process the domain information (i.e., Δ, L_x and L_y). The RHS and domain information branches are subsequently merged by multiplying every channel from the RHS branch by one of the outputs of the domain information branch expressed using tensor notation (without the implicit summation) as

$$A_{ijkl}B_{ij} = C_{ijkl}$$

Where index i is the batch dimension, index j is the channel dimension, and the remaining indices are for the spatial dimensions. Finally, several more convolutions apply to the result, thus reducing the final number of channels to 1 to produce the solution.

2.2 DBCNN

The DBCNN takes one 1D array containing the BC information, the same domain information used for the HPNN, and the number of grid points in the direction orthogonal to the provided BC.

Given homogeneous DBCs on three boundaries plus one inhomogeneous Dirichlet BC on another on a rectangular domain, the solution of the Laplace equation on the domain $[0, L_x] \times [0, L_y]$ can be written as a

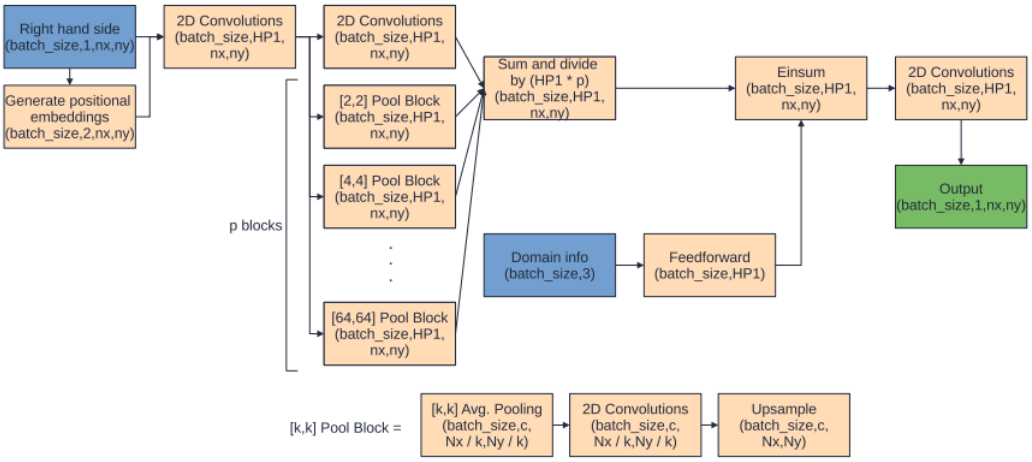

Figure 2: HPNN model

series of the form,

$$\phi = \sum_{m=1}^{\infty} A_m \sinh\left(\frac{m\pi\left(x - L_x\right)}{L_x}\right) \sin\left(\frac{m\pi y}{L_y}\right)$$

More on this can be found in the mentioned paper. It should be noted that this model returns results such that the inhomogeneous BC is always on the left boundary of the domain. However, a series of rotations and reflections are sufficient to modify the result such that any other boundary is the inhomogeneous boundary instead.

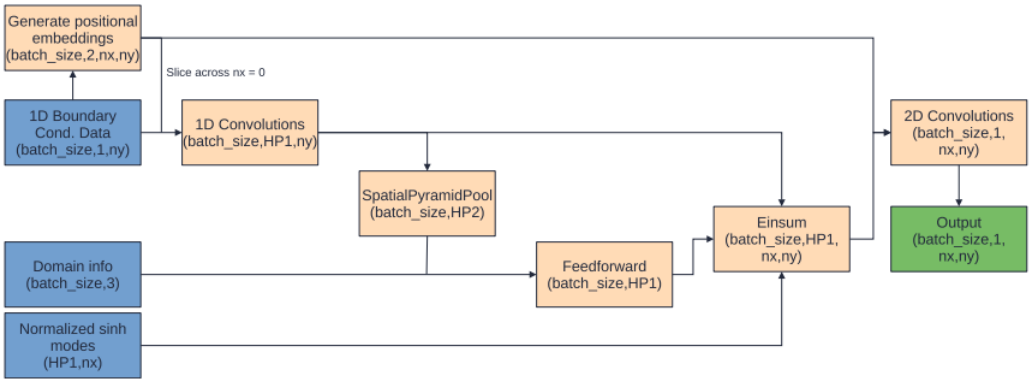

Figure 3: DBCNN model

3 Generating training data

The HPNN submodel was trained on an analytically generated dataset composed of synthetically generated solution-RHS pairs based on truncated Fourier and polynomial series.

During the dataset synthesis process, the number of terms for each sample in the Fourier and Taylor series components were chosen randomly from a uniform distribution within a predetermined range. Overall, this methodology to generate synthetic homogeneous Poisson problems provides four parameters per spatial dimension to control the roughness of the RHS—two parameters for both series components prescribing the range to draw the number of terms from—for a sum of eight parameters for the two-dimensional problems. The grid parameters were similarly chosen randomly from uniform distributions within predetermined ranges. This necessitates two parameters to choose the range of grid spacings, plus two parameters per spatial dimension for the maximum and a minimum number of grid points across each dimension, for a total of six parameters for a 2D problem.

Note that grid spacings were randomly chosen per-batch instead of per-sample.

4 Prerequisite for Poisson_CNN

The prerequisite to running the code in the Paramshakti are as follows:

- The TensorFlow modules available in the Paramshakti are not compatible to running these codes. Additionally, the docker file (associated with the codes) cannot be run on Paramshakti. So, the roundabout to this constraint is to create a virtual environment *(pip venv is recommended over conda)*. Then, refer to the docker file and install the **requirements** *(exact to the version specified)* manually within the newly created virtual environment.

- Activate the virtual environment. Furthermore, install the poisson_CNN python package using the following command executed in the folder containing the setup file (setup.py).

```
pip3 install --no-deps -e .
```

Listing 1: Install poisson CNN package

- Download the AmgX from the NVIDIA developer website to the home directory. Add the following commands to the **.bashrc** file:

```
# AMGX
LD_LIBRARY_PATH=$HOME/amgx/build:$LD_LIBRARY_PATH
LD_PRELOAD="/opt/ohpc/pub/compiler/gcc/8.3.0/lib64/libstdc++.so.6 $LD_PRELOAD"
export LD_LIBRARY_PATH
export LD_PRELOAD

export AMGX_DIR=$HOME/amgx
```

Listing 2: AmgX path

- Install **pyamgx**, a python binding to the AmgX, using the following command:

```
pip3 install pyamgx
```

Listing 3: Pyamgx installation

- Load the required modules using the following commands:

```
module load compiler/gcc/8.3.0
module load compiler/cudnn/8.1.0
module load compiler/cuda/11.0
```

Listing 4: Required Modules

Users can automate the execution of listings 4 by adding it to their .bashrc file. With this setup, the user is one step ahead to execute the Poisson_CNN codes.

5 Poisson_CNN code structure

Before executing code, the user must understand the code structure. It will help the user during code modification. The figure 5 draws the Poisson_CNN tree structure.

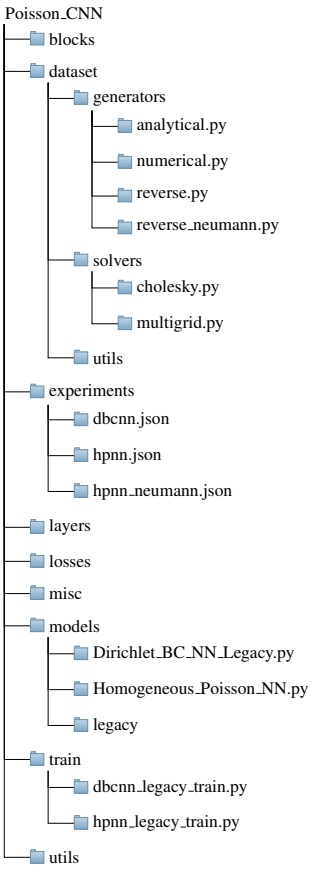

Figure 4: Poisson_CNN tree

During the dataset generation phase, for HPNN, the analytical function scripted within analytical.py is used. However, for DBCNN, the numerical approach (numerical.py) is used. All inputs to the dataset generation and the training phase are extracted from the JSON file present within the experiments folder. Several models are architecture within the folder models. Among them, Dirichlet_BC_NN_Legacy and Homogeneous_Poisson_NN are prominently used once. Others are used for comparison. The dbcnn_legacy_train.py

and the hpnn_legacy_train.py are the python scripts that call the data generation and training phase in sequential order. Further output the checkpoints containing the weights and other information.

6 Training the models

The layers of the models are designed for NHWC data type, which can only be run on GPUs. So, the primary task is to get GPU time.

```
srun -q somnathme --partition=gpu --time=30:00:00 --nodes=1 --gres=gpu:1 --pty /bin/bash
```

Listing 5: GPU allocation

From listing 5, a single node with one GPU is suitable to train the model. Once the allocation is credited, the user can execute the training process. For HPNN and DBCNN model training, execute the following commands.

```
#hpnn command.
python3 -m poisson_CNN.train.hpnn_legacy_train ~/AI_CFD/poisson_CNN/poisson_CNN/
    poisson_CNN/experiments/hpnn.json

#dbcnn command.
python3 -m poisson_CNN.train.dbcnn_legacy_train ~/AI_CFD/poisson_CNN/poisson_CNN/
    poisson_CNN/experiments/dbcnn.json
```

Listing 6: Training the models

To restart the previously trained models, the user can use the above commands by appending them with the previous checkpoints path.

```
#dbcnn retraining
python3 -m poisson_CNN.train.dbcnn_legacy_train ~/AI_CFD/poisson_CNN/poisson_CNN/
    poisson_CNN/experiments/dbcnn.json --continue_from_checkpoint /scratch/j20210241/
    poisson_cnn_model_test/

#hpnn_restart_command.
python3 -m poisson_CNN.train.hpnn_legacy_train_neumann ~/AI_CFD/poisson_CNN/poisson_CNN/
    poisson_CNN/experiments/hpnn_neumann.json --continue_from_checkpoint /scratch/
    j20210241/poisson_cnn_model_test/hpnn_legacy_train_neumann/

#dbcnn_rnn_restart_command.
python3 -m poisson_CNN.train.dbcnn_rnn_train ~/AI_CFD/poisson_CNN/poisson_CNN/poisson_CNN/
    experiments/dbcnn_rnn.json --continue_from_checkpoint /scratch/j20210241/
    poisson_cnn_model_test/dbcnn_rnn_train_salloc/

#pcnn retrain command.
python3 -m poisson_CNN.train.pcnn_end_to_end ~/AI_CFD/poisson_CNN/poisson_CNN/poisson_CNN/
    experiments/pcnn_end_to_end.json --continue_from_checkpoint /scratch/j20210241/
    poisson_cnn_model_test/pcnn_train/
```

Listing 7: Retraining the models

The data parallelism process is successfully tested using the horovod package. To obtain the horovod implemented codes, the user needs to change the working branch to modi_CNN by executing the commands mentioned below. *(Note: User needs to make sure that the sample size is reduced to the number of parallel GPU cores accessed for training the model).*

```
#git command
git checkout modi_CNN

```

```
#Execute the following command from the Poisson_CNN folder.
time CUDA_VISIBLE_DEVICES='0,1' horovodrun -np 2 -H localhost:2 python3 /scratch/j20210241
    /test_poisson_CNN_folder/poisson_CNN/poisson_CNN/poisson_CNN/train/pcnn_end_to_end.py
    /scratch/j20210241/test_poisson_CNN_folder/poisson_CNN/poisson_CNN/poisson_CNN/
    experiments/pcnn_end_to_end.json
```

Listing 8: Horovod run

7 Interlinking with CFD solver

Once the models are trained to an acceptable level, the outcome is saved in the form of checkpoints. These checkpoints have to be used to predict the solutions, in the present case, its CFD solutions. Following listing 9, represents an attempt to predict the solution from HPNN trained model. The inputs (rhs, dx) for the function *prediction_model_hpnn* are obtained from other function (ground truth) which will be mentioned in the upcoming section.

```
def predition_model_hpnn(rhs, dx, Nx, Ny):
    cfg = poisson_CNN.convert_tf_object_names(
    json.load(open("/scratch/j20210241/test_poisson_CNN_folder/poisson_CNN/poisson_CNN/
    poisson_CNN/experiments/hpnn.json")))
    model = poisson_CNN.models.Homogeneous_Poisson_NN_Legacy(**cfg["model"])
    _ = model([tf.random.uniform((1, 1, 500, 500)), tf.random.uniform((1, 1))])
    model.compile(loss="mse", optimizer="adam")
    model.load_weights(
    "/scratch/j20210241/test_poisson_CNN_folder/poisson_CNN/poisson_CNN/
    hpnn_batch_2_steps_5000_epoch_7/chkp-epoch-07.mse-0.0000")
    trhs, sf = poisson_CNN.utils.set_max_magnitude
    _in_batch_and_return_scaling_factors(
    rhs.reshape([1, 1] + list(rhs.shape)).astype(np.float32), 1.0)
    tdx = tf.cast(tf.constant([[dx]]), tf.float32)
    trhs = tf.cast(trhs, tf.float32)
    pred = model([trhs, tdx])
    # pred = tf.abs(pred)
    # pred = (((dx * (Ny - 1)) * 2) / sf) * pred
    plt.imshow(pred[0, 0], cmap="RdBu", origin="lower")
    plt.colorbar()
    plt.savefig("predicted_hpnn" + str(Nx) + "_" + str(Ny) + ".png", dpi=300)
    plt.close()
return pred[0, 0]
```

Listing 9: DL_CFD Interlink

The outcome of the function is saved in the format *.png* and compared with ground truth in later stages. If results from the predicted are found to be satisfactory, then they can be embedded into the solver (IBM, CFD solver) to accelerate the Poisson equation solution. Otherwise, the model is tuned for hyperparameters to obtain better results.

Generally, the efficient CFD codes are written in C or Fortran. However, the lines corresponding to the solution prediction are in Python. So, there is a need for embedding the Interpreted language (Python) with Compiled language (C or Fortran). To address this, the user can take the help of **Forcallpy**. Since the IBM code is written in Fortran, the above link will provide total information for the users to embed the Python call function within the IBM code.

8 Computed/Actual CFD solution

In order to compare the predicted solution with the actual, a gauge is required. For this, one can use the whole CFD code (for example OpenFOAM) to generate the actual solution. However, in our case of predicting the solution for the Poisson equation, the user can directly extract the actual solution by the sparse method as shown in listing 10.

```
from __future__ import print_function
import time
import numpy as np
import pylab as py
import scipy.sparse as sp  # import sparse matrix library
from scipy.sparse.linalg import spsolve
py.rcParams.update({'font.size': 14})
from diff_matrices import Diff_mat_1D, Diff_mat_2D

def diffusion_eqn_sparse(Nx, Ny, T_batch, g):
    # Dirichlet boundary conditions
    uL = T_batch[0]
    uR = T_batch[1]
    uT = T_batch[2]
    uB = T_batch[3]

    # Defining custom plotting functions
    def my_contourf(x, y, F, ttl):
        py.imshow(abs(F), cmap='RdBu', origin='lower')
        py.colorbar()
        py.title(ttl)
        py.savefig('new_save_contour' + ' ' + str(Nx) + '_' + str(Ny) + '.png', dpi=300)
        py.close()
        return 0

    x = np.linspace(-3, 3, Nx)  # x variables in 1D
    y = np.linspace(-3, 3, Ny)  # y variable in 1D

    dx = x[1] - x[0]  # grid spacing along x direction
    dy = y[1] - y[0]  # grid spacing along y direction

    X, Y = np.meshgrid(x, y)  # 2D meshgrid

    # 1D indexing
    Xu = X.ravel()  # Unravel 2D meshgrid to 1D array
    Yu = Y.ravel()

    # Loading finite difference matrix operators
    Dx_2d, Dy_2d, D2x_2d, D2y_2d = Diff_mat_2D(Nx, Ny)  # Calling 2D matrix operators from
     funciton

    # Boundary indices
    ind_unravel_L = np.squeeze(np.where(Xu == x[0]))  # Left boundary
    ind_unravel_R = np.squeeze(np.where(Xu == x[Nx - 1]))  # Right boundary
    ind_unravel_B = np.squeeze(np.where(Yu == y[0]))  # Bottom boundary
    ind_unravel_T = np.squeeze(np.where(Yu == y[Ny - 1]))  # Top boundary

    ind_boundary_unravel = np.squeeze(
        np.where((Xu == x[0]) | (Xu == x[Nx - 1]) | (Yu == y[0]) | (Yu == y[Ny - 1])))  #
    All boundary
    ind_boundary = np.where((X == x[0]) | (X == x[Nx - 1]) | (Y == y[0]) | (Y == y[Ny -
    1]))  # All boundary

```

```
    # Construction of the system matrix
    start_time = time.time()
    I_sp = sp.eye(Nx * Ny).tocsr()
    L_sys = D2x_2d / dx ** 2 + D2y_2d / dy ** 2  # system matrix without boundary
    conditions

    L_sys[ind_boundary_unravel, :] = I_sp[ind_boundary_unravel, :]

    # Construction of right hand vector (function of x and y)
    b = g
    b[ind_unravel_L] = uL
    b[ind_unravel_R] = uR
    b[ind_unravel_T] = uT
    b[ind_unravel_B] = uB

    # solve
    u = spsolve(L_sys, b).reshape(Ny, Nx)

    # Plot solution
    py.figure(figsize=(14, 10))
    my_contourf(x, y, u, r'$\nabla^2 u = f(x,y) OR constant$')

    return b, dx, dy, u
```

Listing 10: 2D Conduction Equation

Here, the input parameters are the number of grid points and domain limits. The code calculates the grid spacing and further to the solution. The rhs (source term in Poisson equation) and the dx (grid spacing) are assigned to the array and further used to predict the solution.

9 Results

The HPNN model is trained with batch size = 2, steps = 5000, and epoch = 7 to obtain the required accuracy *(refer [2])*. Further, the model is prepared to predict the solution *(refer to appendix)*. Meanwhile, the sine function creates the source effect *(RHS of Poisson function)* in the domain (365×365).

$$\boxed{b = sin\left(1 - \frac{x_i}{n}\right)}$$

Here, x_i is the index value of the grid point, n is the total grid points. The outcome of the function is represented by *figure 5*. The source term b and the grid points *(N_x, N_y)* are passed as inputs to the conventional solver ($diffusion\_eqn\_sparse$, *refer line 61 of the appendix*) to obtain the ground truth, as shown in *figure 7*. The same inputs are passed to the model, and the predicted solution is obtained *(figure 8)*. Finding the relative difference between them will result in figure 6.

As mentioned before, the results are obtained from batch size 2. However, the author used the batch size of 50 for training the model. The reason for using a lower batch size is the limitation of GPU resources. Because of this, the corners are under-predicted. (figure 6). Also, the error at the mid and upper section of the domain is slightly higher than in the lower section. The reason for this behaviour is yet to be identified. DGX generated data:

However, model prediction improved when it was trained with higher batch size (=50)*(as mentioned by the author)*. *Figure 10* depicts the same. Here, the NVIDIA DGX machine is used to train the model. The following are the machine specs: 512GB memory, single node, 8 GPUs, each with 32GB GPU cards. The HPNN model is trained for the loss $\leq 1.13 \times 10^{-2}$.

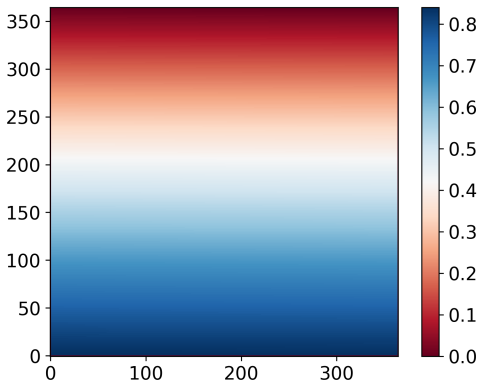

Figure 5: RHS of Poisson equation, 365×365.

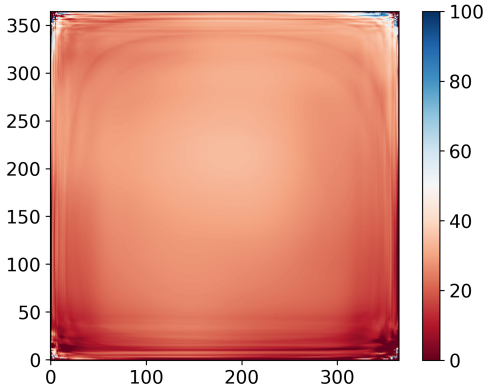

Figure 6: Relative error for grid size: 365×365.

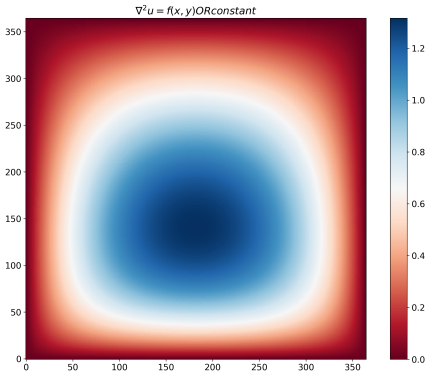

Figure 7: Ground truth for grid size: 365×365.

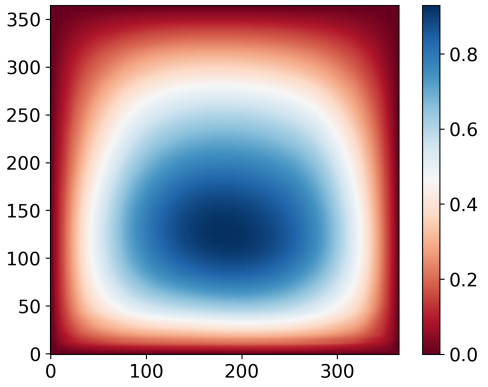

Figure 8: Predicted solution for grid size: 365×365.

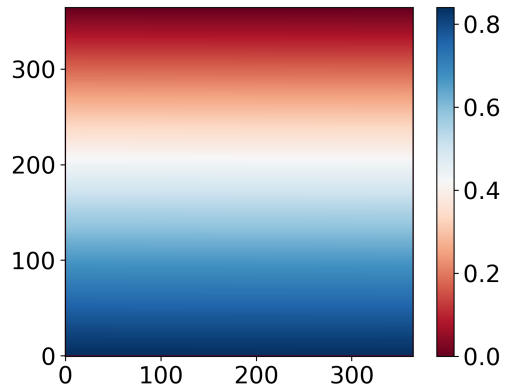

Figure 9: RHS of Poisson equation, 365×365.

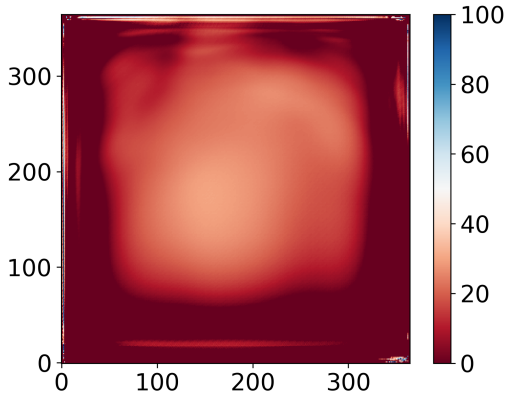

Figure 10: Relative error for grid size: 365×365.

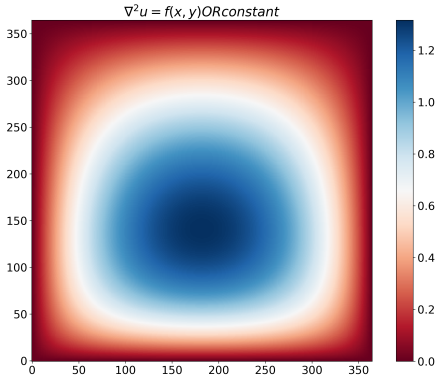

Figure 11: Ground truth for grid size: 365×365.

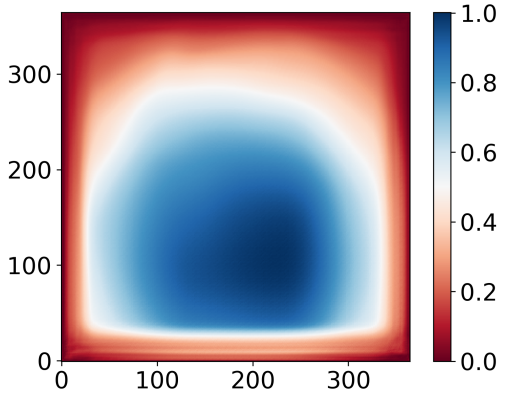

Figure 12: Predicted solution for grid size: 365×365.

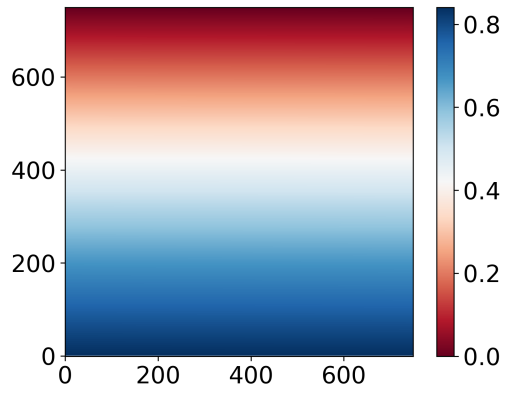

Figure 13: RHS of Poisson equation, 750×750.

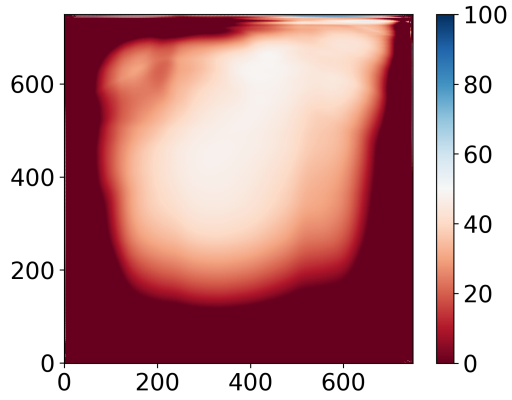

Figure 14: Relative error for grid size: 750×750.

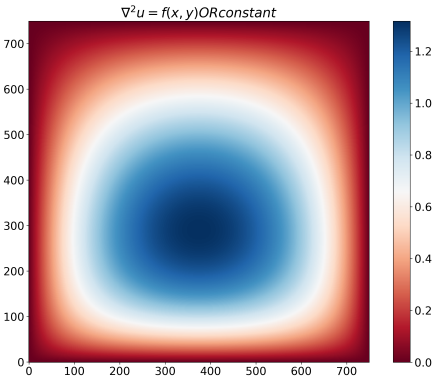

Figure 15: Ground truth for grid size: 750×750.

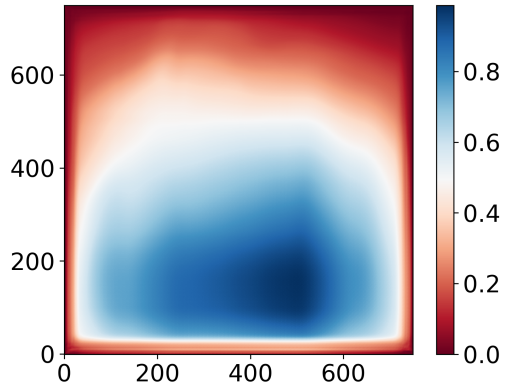

Figure 16: Predicted solution for grid size: 750×750.

There is a clear difference between the relative error (10, 6) when trained with different batch sizes. The overall prediction of the model from the higher batch size is superior to that with a lower batch size. Even domain corners points are well predicted in figure 10.

A domain with grid points between 192 and 384 is used to generate the dataset. The model is excellent in solution prediction for the range of grid points to which training dataset is generated. However, the results are erroneous while predicting the solution outside the domain (extrapolation). Figure 14 represents the same. The prediction results with different domain lengths are attached in appendix (refer 22, 18).

10 Future work

It took a long time to understand the code and reproduce a fraction of the results. However, in the way of understanding, many bottlenecks are identified. The Poisson_CNN code assumes that the grid spacing is

identical in both the x and y directions. This is not true in many cases. The code fails to predict results in the case of higher aspect ratio grids. Following are some more factors on which users can concentrate to improve the code performance.

- For batch size mentioned in the paper [2], the models cannot fit into the available (16 GB) GPU resources (OOM issue). I tried data parallelism, but it did not result in any changes. The model parallelism is the way to be worked upon.
- There is no mention of hyperparameters optimization. This can be addressed with the help of Neural Architecture and Hyperparameter Search engine [1] called **DeepHyper**.
- Most concerning issue of any ANN is **Spectral Bias** [3], i,e, during training, the model learns the low-frequency features quickly, while struggles or sometimes fail to learn the high-frequency features. By addressing it, more accurate results can be obtained. This also paves the way for predicting turbulent flows.
- Attempt to replicate the work using Physics-based Neural Network (PINNs) [4]. Recently, different versions of PINNs have been excellent for predicting the solution. Beginning with **DeepXDE**, one can explore other frameworks like **Modulus** (formerly called as SimNet) and others.
- Later, the working PINN code can be embedded into the IBM codes using the Python wrapper for Fortran. This work is in similar lines with the **PythonFOAM**.

11 Code and data accessibility

The updated code can be accessed from the following hyperlink: *Poisson-CNN.git*.
Contact me at **neelu065atgamil.com** for more help.

References

[1] Romit Maulik et al. "Recurrent neural network architecture search for geophysical emulation". In: *IEEE* (2020).

[2] Ali Girayhan Özbay et al. "Poisson CNN: Convolutional neural networks for the solution of the Poisson equation on a Cartesian mesh". In: *Data-Centric Engineering* (2021). DOI: 10.1017/dce.2021.7.

[3] Nasim Rahaman et al. "On the Spectral Bias of Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5301–5310. URL: https://proceedings.mlr.press/v97/rahaman19a.html.

[4] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

Appendices

The below code 11 is the combination of the above mentioned function and this when executed will generate the actual CFD solution and with same inputs predicts the solution and calculates the absolute error between them.

```
from __future__ import division
import matplotlib.pyplot as plt
import poisson_CNN
import tensorflow as tf
import json
from Poisson_2D_Dirichlet_v1 import diffusion_eqn_sparse
import os
import numpy as np

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

Nx = 750  # Along y-direction, check aspect-ratio plot.
Ny = 750

T_left = 0
T_bottom = 0
T_right = 0
T_top = 0


def predition_model_hpnn(rhs, dx, Nx, Ny):
    cfg = poisson_CNN.convert_tf_object_names(
        json.load(open(
            '/scratch/j20210241/test_poisson_CNN_folder/poisson_CNN/poisson_CNN/poisson_CNN/experiments/hpnn.json')))
    model = poisson_CNN.models.Homogeneous_Poisson_NN_Legacy(**cfg['model'])
    _ = model([tf.random.uniform((1, 1, 500, 500)), tf.random.uniform((1, 1))])
    model.compile(loss='mse', optimizer='adam')
    model.load_weights(
        '/scratch/j20210241/test_poisson_CNN_folder/poisson_CNN/poisson_CNN/hpnn_batch_2_steps_5000_epoch_7/chkpt'
        '-epoch-07.mse-0.0000')

    trhs, sf = poisson_CNN.utils.set_max_magnitude_in_batch_and_return_scaling_factors(
        rhs.reshape([1, 1] + list(rhs.shape)).astype(np.float32), 1.0)
    tdx = tf.cast(tf.constant([[dx]]), tf.float32)
    trhs = tf.cast(trhs, tf.float32)
    pred = model([trhs, tdx])
    pred = tf.abs(pred)
    pred = (((dx * (Ny - 1)) ** 2) / sf) * pred
    plt.imshow(pred[0, 0], cmap='RdBu', origin='lower')
    plt.colorbar()
    plt.savefig('predicted_hpnn' + ' ' + str(Nx) + '_' + str(Ny) + '.png', dpi=300)
    plt.close()
    return pred[0, 0]


def abs_error_hpnn(actual_temp, pred_temp_hpnn, Nx, Ny):
    abs_per_error = ((abs(actual_temp) - abs(pred_temp_hpnn)) / abs(actual_temp)) * 100
    abs_per_error = np.nan_to_num(abs_per_error)

    plt.imshow(abs_per_error, vmax = 0, vmin = 100, cmap='RdBu', origin='lower')
    plt.colorbar()
```

```
    plt.savefig('abs_error_hpnn' + ' ' + str(Nx) + '_' + str(Ny) + '.png', dpi=300)
    plt.close()


if __name__ == "__main__":
    T_batch = np.array([T_left, T_right, T_top, T_bottom])
    b = []
    n = Nx*Ny
    for i in range(n):
        b.append(np.sin(1 - i/n)) # sine function is used for rhs evaluation

    b = np.array(b)
    # ------ Conventional solver ------ #
    rhs, dx, dy, actual_temp = diffusion_eqn_sparse(Nx, Ny, T_batch, b)

    # ------ rhs plot ------ #
    rhs = rhs.reshape(Nx, Ny)
    plt.imshow(rhs, cmap='RdBu', origin='lower')
    plt.colorbar()
    plt.savefig('rhs' + ' ' + str(Nx) + '_' + str(Ny) + '.png', dpi=300)
    plt.close()

    # ------ Predicted solution HPNN ------ #
    pred_temp_hpnn = predition_model_hpnn(rhs, dx, Nx, Ny)

    # ----- absolute error plot ------ #
    abs_error_hpnn(actual_temp, pred_temp_hpnn, Nx, Ny)
```

Listing 11: Calculates absolute error between actual and predicted solution.

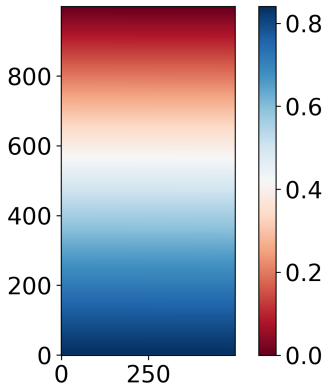

Figure 17: RHS of Poisson equation, 500×1000.

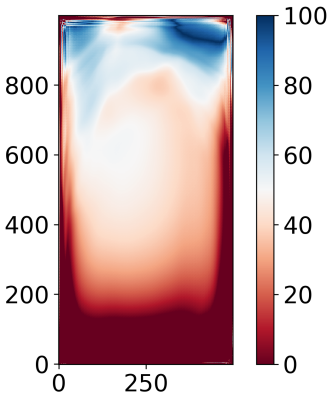

Figure 18: Relative error for grid size: 500×1000.

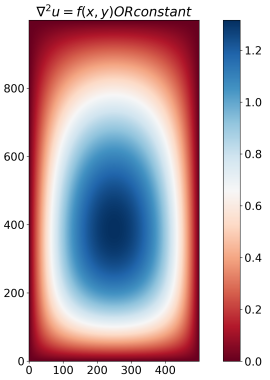

Figure 19: Ground truth for grid size: 500×1000.

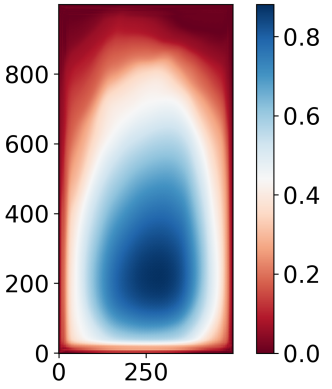

Figure 20: Predicted solution for grid size: 500×1000.

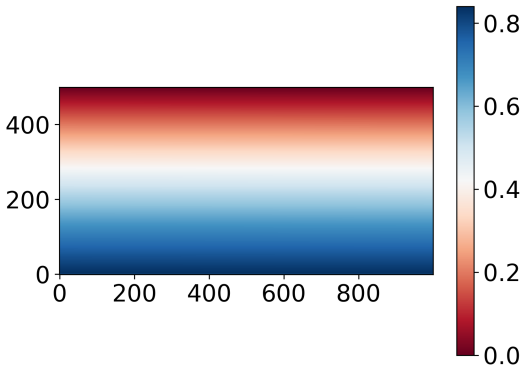

Figure 21: RHS of Poisson equation, 1000×500.

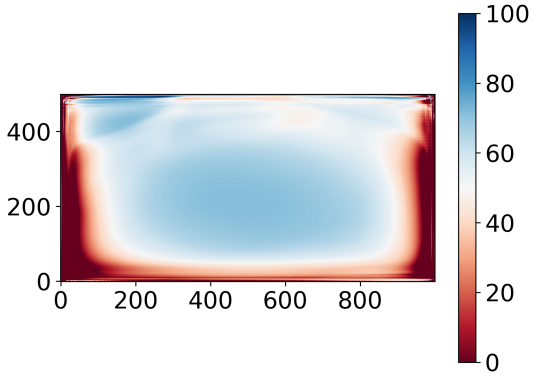

Figure 22: Relative error for grid size: 1000×500.

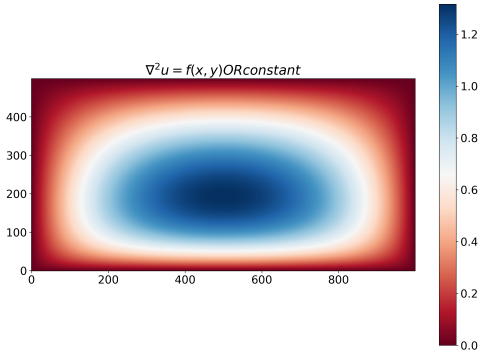

Figure 23: Ground truth for grid size: 1000×500.

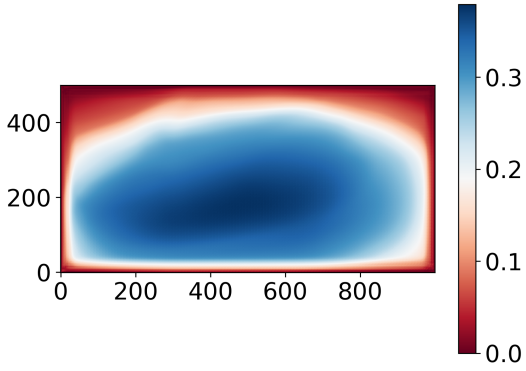

Figure 24: Predicted solution for grid size: 1000×500.