Solving Partial Differential Equations using Spatiotemporal Graph Neural Networks with Continual Learning

GDL 2022
Group 06
Project 08

Zenin Easa Panthakkalakath

Abstract

Continual Learning, represents the idea of learning continuously about the external world in order to enable the autonomous, incremental development of ever more complex skills and knowledge in current intelligent system. In this project, we explore the possibility of continual learning under the combination of graphic deep learning message passing and partial differentiation equations; to achieve a solver that learns to solve using data generated by classical solvers and then generalize it, with the extended ability to be retrained without forgetting what it had learnt in the past.

1 Introduction

In the field of Machine Learning and Artificial Intelligence, the concept of Continual Learning is gaining vital importance. It is inspired by our brain's ability to learn in a sequential manner without forgetting knowledge that was gathered earlier. Some of the approaches that have been adopted for this include storing previous training data information stochastically, and, regenerating past learning experiences with a generative model.

Graph Neural Networks (GNNs) are deep learning based methods that are used to make predictions on graph structural data; predictions tasks be at node-level, edge-level, and/or graph-level. Relational data that varies in both space and time can be analyzed with Spatiotemporal Graph Neural Networks, which is capable of modeling spatially separated time series of nodes and exchange of information between them.

It can be argued that the world around us is governed by Partial Differential Equations (PDEs). Navier-Stokes equation, which describe the behaviour of fluids, and, Heat equation are a couple of our favourite PDEs. It is generally hard to solve these equations analytically, and hence, computational methods are employed for the same. Finite element and finite difference methods are traditionally used; however, recently deep learning based methods are being experimented to see if they can provide better performance. These deep learning models are generally trained on synthetic data that are created using the differential equations as they are known, which means that there are no shortage of datapoints to train on.

Since Spatiotemporal GNNs are designed to work on structured data varying in both space and time, it is intuitive that the network should be able to work well when it comes to solving partial differential equations capturing behaviours varying in space and time.

2 Motivation behind methodology

Let us attempt to articulate our motivation behind moving forward with our methodology using a diffusive heat transfer example.

Bateman–Burgers equation is a partial differential equation that is used in fluid mechanics, nonlinear acoustics, gas dynamics and traffic flow. The following is it's mathematical form.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \alpha\frac{\partial^2 u}{\partial x^2}$$

Heat equation can be seen as a special case of conservative form of Bateman–Burgers equation extended to the three dimensional cartesian coordinates. It describes how diffusive heat transfer takes place in our universe. It is usually written as shown below:

$$\frac{\partial u}{\partial t} = \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)$$

where u is the heat, t is time, (x, y, z) represents a point in the three dimensional space, and α is thermal diffusivity.

There may be simple cases where the aforementioned equation would have a direct solution; however, it is generally hard to find the true solution for a complicated system, even the ones that very well appear simple in our intuition. For example, Figure 1 shows the temporal evolution of temperature in a room that is exposed to the ambient environment via a window when a heater is turned on. We may be interested in finding the optimal position where the heater should be placed so that the temperature in the room at steady state remains within a desired range. It would be hard to find the true solution for this problem; and hence, people usually use numerical methods like Finite Element or Finite Difference to solve the same.

The accuracy of numerical methods can be improved by decreasing the size of the grid points and the time difference are divided into. However, the smaller you make these, the more computationally intensive it becomes.

From the equations and our intuition, we could formulate this as a spatiotemporal graph wherein one of the nodes

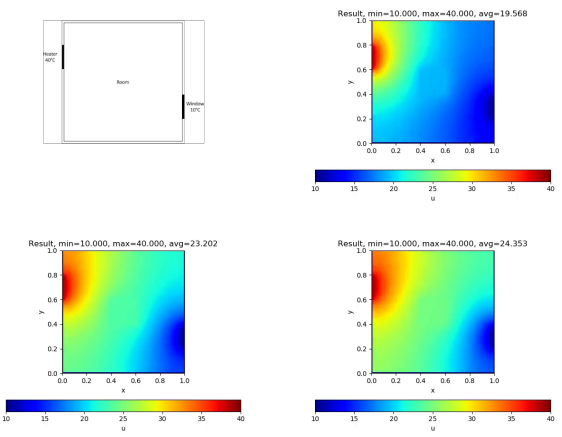

Figure 1. Observing variation in temperature of a room over time when a heater is turned on at a temperature of $40°$C with a window opened to the ambient at temperature of $10°$C

would lie near the window, one near the door, one near the pillow (the values at which needs to be predicted), and a few others around the room capturing the dimensions of the room. This means that, we can train a spatiotemporal graph neural network to model this problem, and we can train it for different values of temperatures in the hallway and outside the window. The datapoints could be measured by constructing an actual room like this, or, by doing a numerical simulation using Finite Element or Finite Difference methods.

But if we can solve the problem using the numerical methods, and we are to generate the data using numerical methods, why are we bothering to solve it with a neural network? Well, for starters, if we want to change the temperature in the hallway or outside the window, we would need to run the simulation from the start for the numerical methods. If we wanted to find the results for N combinations of these temperatures, then we would have to run N such simulations. However, with the neural network approach, we could potentially run the numerical simulations for a fraction of combinations of these and then train the neural network with these as the datapoints and attempt to find the solution for the rest of the combinations by evaluating the network.

Now, after we did this for a room, we would want to try it for another room that is shaped differently, has different arrangements and shape for windows and doors. We would desire to train our original model with the new room, so that we can compute things faster there as well. In fact, we may have several such rooms to run such simulations for. We desire that our network should be able to predict accurately for any room and combinations of parameters; and we intend on teaching it with new data when we find that the network performs poorly on certain combination of parameters.

We may even desire to have a network that has the capability of predicting for any kind of differential equations; not just limited to heat diffusion equation. For instance, the following is a kind of reaction diffusion equation, named Fisher's equation, which is used for simulating anything from travelling waves or population dynamics.

$$\frac{\partial s}{\partial t} = D\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} + Rs(1-s)\right)$$

where s is a function of (x, y, t), t is time, (x, y) represents a point in the two dimensional space, and D is the diffusion constant.

We do not wish that the network forgets what it has learned from the training data; however, if we keep on increasing the training data with every newly coming data, then the training process would be tedious. Therefore, it would be to our advantage to use continual learning strategies so that the network can avoid catastrophic forgetting while being able to learn new things.

3 Related works

Here are some of the previously done works that we found interesting and relevant while researching on how to proceed with our work. Our list is short as these were detailed enough to help us with the problem at hand.

There is a work done by Brandstetter, Worrall and Welling titled *Message Passing Neural PDE Solvers*[1] where they attempt to solve one-dimensional partial differential equations using neural message passing, replacing all heuristically designed components in the computation graph with neural network based function approximators.

In the paper titled *Beltrami Flow and Neural Diffusion on Graphs*[2] by Chamberlain, Rowbottom, Giovanni, Dong and Bronstein, the researchers attempt to solve a non-Euclidean diffusion PDE by using positional encoding derived from the graph along with node features.

The book written by Lesort titled *Continual Learning: Tackling Catastrophic Forgetting in Deep Neural Networks with Replay Processes*[3] explains continual learning concepts in great detail.

The article by Ruder titled *An Overview of Multi-Task Learning in Deep Neural Networks*[4] provides information and guidelines on Multi-Task Learning.

4 Implementation

Broadly speaking, the tasks we have undertaken can be categorized into three different umbrellas - generating datasets, defining the model, and, implementing the continual learning algorithm. The following subsections provide more information about the same.

4.1 Generating datasets

As mentioned in the previous sections, we are using data generated from traditional methods for training our model. We used a parallel implementation of heat-equation generator in accordance with courses named PDELab by Patrick Zulian[5] and High Performance Computing by Olaf Schenk[6] for this task, which use numerical methods to solve two dimensional heat equation and two dimensional reaction diffusion equation respectively.

The former dataset, let's call it Dataset 1 from this point forward, was created using the heat diffusion equation. We fixed the room and the temperature of the source and the sink, and, generated datapoints for varying grid sizes and timesteps. Grid sizes include 30×30, 50×50, 100×100, 150×150 and 200×200, and, the maximum simulation time steps are either 100 or 200 steps. A sample generated

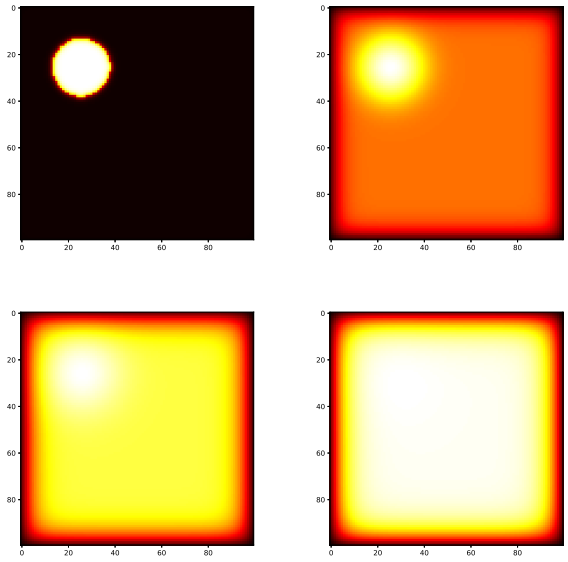

Figure 2. Observing the solution for Fisher's equation (reaction diffusion equation) at different times for a given initial and boundary conditions

data for a particular grid size at three different time steps has already been shown in Figure 1.

The latter dataset, let's call it Dataset 2 from this point forward, was created using a type of reaction diffusion equation, the Fisher's equation. We created several datasets with different initial and boundary conditions. A sample generated data for a particular grid size at three different time steps is shown in Figure 2.

An object of type 'StaticGraphTemporalSignal' from 'torch_geometric_temporal.signal' is created using the aforementioned data such that

- Hundred uniformly distributed points of interest are chosen as nodes.
- Each node is connected to the immediate neighbours in the north, south, east, west, north-east, north-west, south-east and south-west (if they exist) with an edge of unit weight.
- The values at the nodes from four consecutive timesteps form an input feature.
- The value at a later node (10 timesetps from the last input features) is the target feature. (Note that we also have a version of our code with multiple timesteps are taken together in accordance with temporal bundling. However, to demonstrate continual learning, we figured this would be ideal).

Our approach is to train and validate the model on the Dataset 1 first, and then further train it on the Dataset 2 with and without continual learning methods to see how well the continual learning method performs.

4.2 Model

We created a model involving Graph Convolutional Gated Recurrent Unit layer as a network that can learn spatiotemporal features. This is connected to a couple of Rectified Linear Unit layers and a Linear layer. We used Pytorch and PyTorch Geometric Temporal libraries for the same.

The following is the summary of our model generated using the 'torchsummary' package.

```
==========================================================
Layer (type:depth-idx)                        Param #
==========================================================
|-GConvGRU: 1-1                               --
|    |-ChebConv: 2-1                          --
|    |    |-ModuleList: 3-1                   512
|    |-ChebConv: 2-2                          --
|    |    |-ModuleList: 3-2                   2,048
|    |-ChebConv: 2-3                          --
|    |    |-ModuleList: 3-3                   512
|    |-ChebConv: 2-4                          --
|    |    |-ModuleList: 3-4                   2,048
|    |-ChebConv: 2-5                          --
|    |    |-ModuleList: 3-5                   512
|    |-ChebConv: 2-6                          --
|    |    |-ModuleList: 3-6                   2,048
|-ReLU: 1-2                                   --
|-Linear: 1-3                                 33
==========================================================
Total params: 7,713
Trainable params: 7,713
Non-trainable params: 0
==========================================================
```

4.3 Continual learning algorithm

The following is the continual learning strategy that we adopted for the network. The idea is to store and replay old training data while training on a new data, while making use of the information that the stored training data actually acts as a representative for a larger pool of data; we adjust the learning rate and storage of future samples accordingly. This is in accordance with the broad category of "Rehearsal" methods. We believe that this method should be able to work on Multi-Incremental-Tasks to a great extend.

Let n_s represent the number of stochastically sampled stored training datapoints, and, $X_{s,1}, ..., X_{s,n_s}$ be the stoachastically sampled stored training datapoints. Let N be the total number of different training iterations. Let n_{tot} be the total number of datapoints that we have encountered before.

Initialize $n_{tot} = 0$
FOR $i = 1, ..., N$

1. Train the network with $X_{i,1}, ..., X_{i,n_t}$ datapoints and $X_{s,1}, ..., X_{s,n_s}$ datapoints with learning rates α and $\frac{n_{tot}}{n_s}\alpha$ respectively.
2. Stochastically sample n_s training datapoints such that
   (a) Probability of choosing a datapoint from original stored dataset $= \frac{n_{tot}}{n_t+n_{tot}}$
   (b) Probability of choosing a datapoint from the new training dataset $= \frac{n_t}{n_t+n_{tot}}$

   Store the new sampled dataset as the new $X_{s,1}, ..., X_{s,n_s}$.
3. Increment $n_{tot} = n_{tot} + n_t$

END FOR

5 Experiments and results

We experimented to see how well the model performs on the two datasets with and without continual learning.

5.1 Training without continual learning

In this experiment, our goal is to observe how much the network forgets when two subsequent training is done. We first train on Dataset 1 and observe the validation loss, and then, we train on Dataset 2 and see how the validation loss on the first dataset has changed.

Training on Dataset 1: Figure 3 shows the training and validation loss over the epochs for Dataset 1. Following the training, we observed that the smallest validation loss was about 0.00267 on the validation fraction of Dataset 1. Figure 4 shows the target prediction and the prediction from a datapoint in Dataset 1 that the model makes side-by-side.

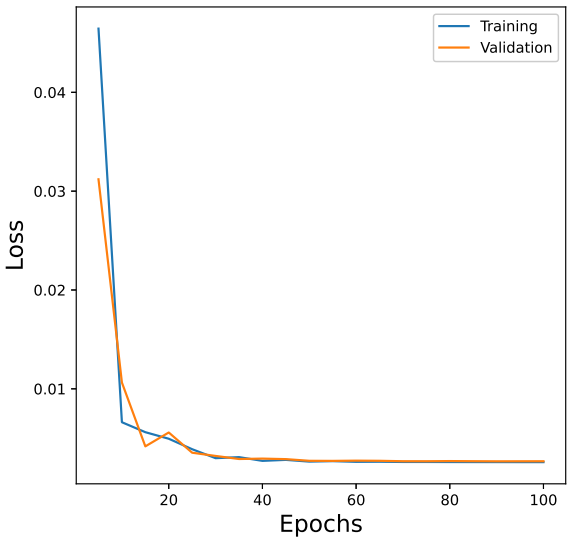

Figure 3. Training and validation loss over different training epochs for Dataset 1

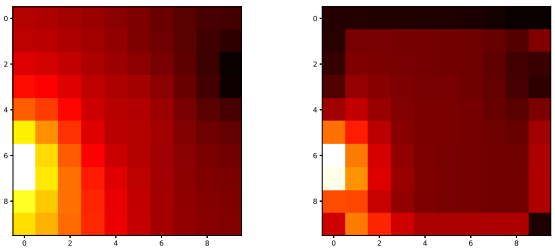

Figure 4. Showing the target and prediction side-by-side for Dataset 1 on the model trained on Dataset 1

Before we went ahead to train on Dataset 2, we wanted to see the how well the current model would work on the dataset without training. We observed that the validation loss on Dataset 2 was about 0.3364. Figure 7 shows the target prediction and the prediction from a datapoint in Dataset 2 that the model makes in it's current state.

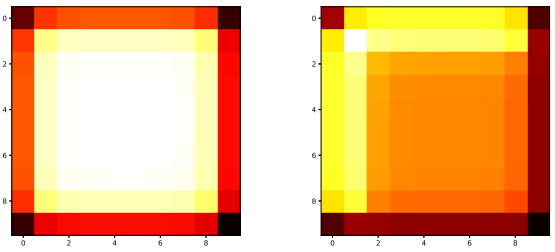

Figure 5. Showing the target and prediction side-by-side for the Dataset 2 on the model trained only on the Dataset 1

Training on Dataset 2: Figure 6 shows the training and validation loss over the epochs for Dataset 2. Following the training, we observed that the smallest validation loss was about 0.0344 on the validation fraction of Dataset 2. Figure 7 shows the target prediction and the prediction from a datapoint in Dataset 2 that the model makes side-by-side. Although the result may not be ideal, we can see that it is getting better at the task.

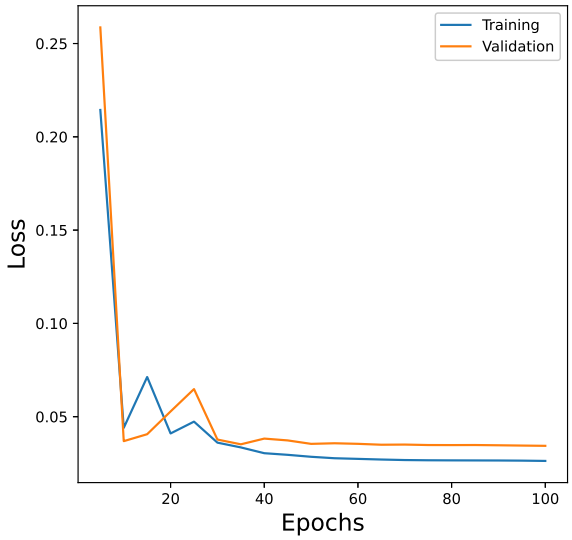

Figure 6. Training and validation loss over different training epochs for the Dataset 2

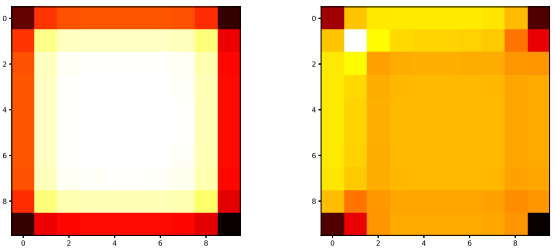

Figure 7. Showing the target and prediction side-by-side for the Dataset 2 on the model trained on the Dataset 2 following the Dataset 1 without continual learning

At the same time, the validation loss for the validation fraction of Dataset 1 now increased to about 0.01385, which shows some forgetting by the network. Figure 8 shows the target prediction and the prediction from a datapoint in Dataset 1 that the model makes in it's current state. The results are clearly getting worse for Dataset 1 while trying to train the model on Dataset 2.

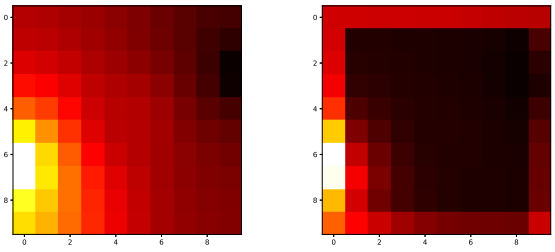

Figure 8. Showing the target and prediction side-by-side for Dataset 1 on the model trained on Dataset 2 following Dataset 1 without continual learning

This justifies the need to implement continual learning so that the increase in loss on Dataset 1 can be reduced.

5.2 Training with continual learning

Using continual learning approach, we observe that the validation loss on the Dataset 1 only increases to about 0.00743, which is better than without having continual learning (which had a loss of 0.01385). The validation loss on Dataset 2 is observed to be 0.0356, which is not significantly different from the case where continual learning was not used. Figure 9 shows the training and validation loss over the epochs for Dataset 2. Note that the training and validation loss over the epochs for Dataset 1 would be identical to that in Figure 3.

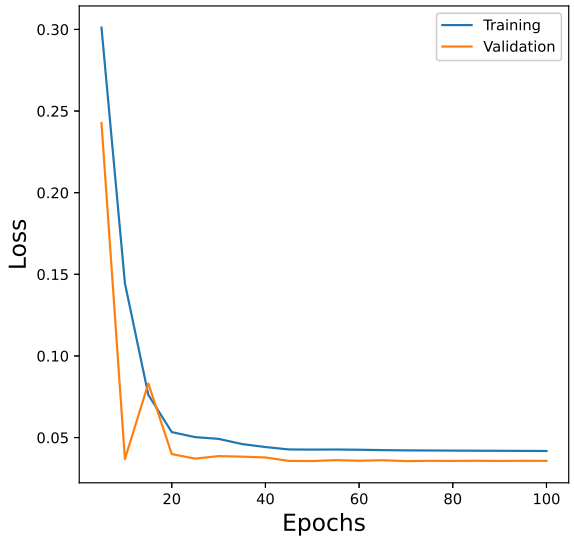

Figure 9. Training and validation loss over different training epochs for the continual learning on Dataset 2

5.3 More comprehensive comparison

To make a more comprehensive comparison, we decided to bring together all the datasets we have and divide it into eight equal parts. Following that, we subdivide them into training and validation datasets. We train on each of these pairs with and without using the continual learning algorithm. We compare the validation loss on the very first validation set after training on each of the eight sets.

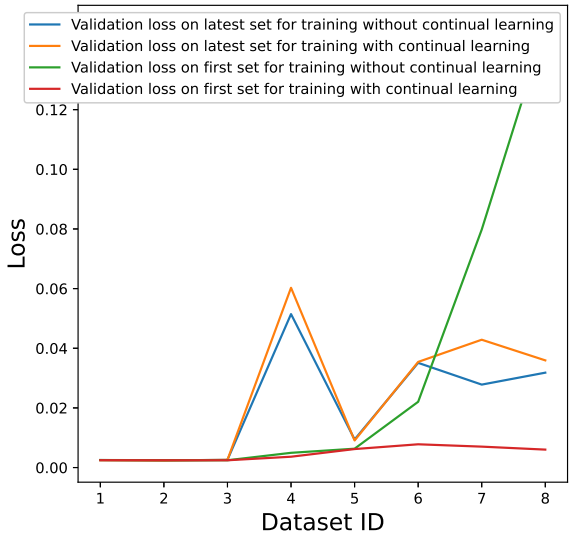

Figure 10. Comparing validation losses while training with and without continual learning when it encounters different datasets

Figure 10 shows our results comparing training using continual learning method versus without continual learning method. From this, we can safely say that the continual learning algorithm that we have used is able to train on the model without losing information it had learnt before, while at the same time, it can still learn from new datasets as well. Catastrophic forgetting can be observed in the training wherein continual learning is not involved.

However, it is worth noting that continual learning version of training took an average time of 446 seconds to run, while the version without continual learning took an average time of 223 seconds to run. In other words, our continual learning version of the training takes about double the time it takes to train compared to the version that doesn't use continual learning.

6 Conclusion

From our experiments, we find that graph based neural networks may be useful for solving different kinds of partial differential equations. In fact, it might actually be possible to arrive at a solver that can potentially provide numerical solution to any kinds of differential equations with incredible speed, if we put in enough effort into creating it. We also found that the continual learning method that we described works well while working with graph neural networks with partial differential equations.

7 Discussion

It is important to criticize some of the things that we have done in this project. We have used data generated using limited types of differential equations with limited initial and boundary conditions. It might be important to try this on wider variety of datasets to have a more complete picture.

Furthermore, instead of storing samples stochastically, training a generative network that creates data similar to what the network has already been trained for would be an interesting thing to try.

In this project, we arranged the graph network in a uniform manner. We strongly believe that having a graph network arranged in a different manner (possible even randomly) would also be able to produce good results. This would be important as in engineering domain, we are usually interested in the solutions at certain critical points only; and this could potentially have fewer number of nodes to work with, which would reduce the computational load, thereby enabling to get the solutions faster.

References

[1] Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers, 2022. URL https://arxiv.org/abs/2202.03376.

[2] Benjamin Paul Chamberlain, James Rowbottom, Davide Eynard, Francesco Di Giovanni, Xiaowen Dong, and Michael M Bronstein. Beltrami flow and neural diffusion on graphs, 2021. URL https://arxiv.org/abs/2110.09443.

[3] Timothée Lesort. Continual learning: Tackling catastrophic forgetting in deep neural networks with replay processes, 2020. URL https://arxiv.org/abs/2007.00487.

[4] Sebastian Ruder. An overview of multi-task learning in deep neural networks, 2017. URL https://arxiv.org/abs/1706.05098.

[5] Patrick Zulian. Pdelab - high performance computing, 2021-2022. URL https://bitbucket.org/zulianp/pdelab.

[6] Olaf Schenk, Pratyuksh Bansal, Juraj Kardos, Malik Lechekhab, and Timothy Holt. High-performance computing 2021 (master usi), 2021. URL https://github.com/oschenk/hpc2021.

Appendix

Our work and results related to this project has been published in the following GitHub page:
https://github.com/zenineasa/Solving-PDE-using-STGNN-with-Continual-Learning