POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Project report

Graph Neural Networks for Partial Differential Equations

Scientific Computing Tools for Advanced Mathematical Modelling

Authors: Andrea Novellini, Randeep Singh and Luca Sosta

Academic year: 2021-2022

The aim of this project is to explore the potential of Graph Neural Networks and, in particular, try to understand if and where they could be useful in solving partial differential equations' problems [1].
As a matter of fact, the mesh of the problem can be easily considered as a graph, where each vertex of the mesh is a node in the graph and each connection between two vertices an edge.
As a further note, in a framework like this it is not possible to implement a simple Convolutional Neural Network: in fact, a mesh does not have a fixed structure like, for example, an image where every pixel has the same amount of neighboring pixels, distributed over a fixed grid. Meshes of PDEs problem, instead, not only differ in structure, but even nodes of the same mesh will have different connectivities, thus making impossible to use the traditional convolution operator. This is where Graph Convolution comes into play: it generalizes the convolution operator to irregular domains.
All this considered, using particular architectures that will be illustrated in this report, we want to see if adding the connectivity of the mesh as a feature fed into a neural network, can improve the accuracy in the estimation of the solution.

1. Mathematical formulation of the problem

Let us consider the following parametrized Poisson's problem:

$$\begin{cases} -\Delta u_p = -6p(x+y) =: f_p(\mathbf{x}) & \mathbf{x} \in \Omega = [0,1]^2 \\ u_p = 1 + p(x^3+y^3) & \mathbf{x} \in \partial\Omega \end{cases}, \quad p \in [p_{min},\, p_{max}] \subset \mathbb{R}^+ \tag{1}$$

The goal is to build a model based on Graph Neural Networks able to solve our problem, given the parameter p:

$$u_p \approx \mathcal{GNN}(G_p),\ \forall p \in [p_{min},\, p_{max}]$$

where with G_p we indicate the graph associated to the problem with forcing function f_p. But how do we construct these graphs and the full dataset to train our model? The workflow is the following:

1. Generate a triangular mesh $\mathcal{T}_h$ using $\mathcal{P}_1$ elements, with M vertices having coordinates $\{\mathbf{x_j}\}_{j=1}^M$
2. Choose $N \in \mathbb{N}$ and sample $\{p_i\}_{i=1}^N \subset [p_{min}, p_{max}]$ so that we obtain N different PDEs for each sampled p_i;
3. Compute numerical solutions $\{u_{p_i}\}_{i=1}^N$ on the mesh generated at point 1;
4. For each problem, we generate a graph G_{p_i}. For each graph, nodes $\{V_j^i\}_{j=1}^M$ are the vertices of the mesh and the connectivity is retrieved from the mesh edges (as explained in the introduction). For each node V_j^i, set the feature $h_j^i = f_{p_i}(\mathbf{x_j})$ and the target $t_j^i = H(u_{p_i}(\mathbf{x_j}))$, indicating with H a suitable normalization. Eventually, the dataset is the set of all graphs, namely $\mathcal{D} = \{G_{p_1}, G_{p_2}, \ldots, G_{p_N}\}$ [2].

We normalize the target values in [0,1]:

$$t_j^i = H(u_{p_i}(\mathbf{x_j})) := \frac{u_{p_i}(\mathbf{x_j}) - u_{min}}{u_{max} - u_{min}} \in [0,1]$$

where,

$$u_{min} = \min_{i \in \{1,\ldots,N\},\, j \in \{1,\ldots,M\}} u_{p_i}(\mathbf{x_j}), \quad u_{max} = \max_{i \in \{1,\ldots,N\},\, j \in \{1,\ldots,M\}} u_{p_i}(\mathbf{x_j})$$

2. Methods

We implement a model, made of two parts:

1. Graph Convolutional layer;
2. Feed Forward fully-connected layer.

The first is used to pool information from the neighborhood of each node, exploiting the connectivity of the mesh. There are many different ways to implement the convolution and we opted for the *Graph Attention Networks* (GAT/GATv2, see sections 2.1/2.2). The fully-connected layer instead is used to boost the overall generalization power of the network adding more degrees of freedom, making the network more suitable to learn the non-linear behavior of the problem.

2.1. Graph Attention layer (GAT)

The Graph Attention layer is one of the most used graph convolutional layer. It allows for (implicitly) assigning different importances to nodes of a same neighborhood without any kind of costly operations. Moreover, it is computationally very efficient, since the pooling operations are parallelizable across all edges.

From now on, we will discuss the unidimensional case, so the nodes features are just scalars (namely, the value of the forcing function on the specific node of the specific graph). However, all the arguments can be easily generalized to the vectorial case. For a more rigorous discussion see Velickovic et al. (2018) [3].

The input of the layer is a set of node features $\mathbf{h} = \{h_1, \ldots, h_M\}$, and it produces a set of transformed node features $\mathbf{h}' = \{h'_1, \ldots, h'_M\}$.

Let $w \in \mathbb{R}$ be a learnable *weight* parametrizing a shared linear transformation, applied to every node. It is just used to transform the input features into high-level features (so, in poor words, w acts just as a re-scaling factor).

The real heart of the network is the so called *shared attention mechanism* $a : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, through which we compute the *attention coefficients*:

$$e_{ij} = a(wh_i, wh_j) \tag{2}$$

that indicates the importance of node j's features to node i. Of course, we compute the attention coefficient e_{ij} only if exists an edge connecting the nodes (i, j).
Since the attention coefficients vary over all $\mathbb{R}$, to make coefficients more interpretable and easily comparable across different nodes we normalize each e_{ij} across all the nodes in the neighborhood $\mathcal{N}_i$ of node i, performing a softmax, obtaining the *normalized attention coefficients*:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \in [0,1] \tag{3}$$

so that, if $\alpha_{ij} = 0$, you can assume that node j and i are independent.
Typically, the attention mechanism a is a single-layer feed-forward neural network, parametrized by a weight vector $\vec{a} \in \mathbb{R}^2$, with a LeakyReLU non linearity so that:

$$\alpha_{ij} = \frac{\exp(LeakyReLU(\vec{\mathbf{a}} \cdot [wh_i || wh_j]))}{\sum_{k \in \mathcal{N}_i} \exp(LeakyReLU(\vec{\mathbf{a}} \cdot [wh_i || wh_k]))} \tag{4}$$

where $||$ indicates the concatenation operation.

Eventually, we can state the updating formula:

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij} w h_j\right), \; i = 1,\dots,M \tag{5}$$

where σ is any non-linearity. Note that the updating is nothing but a convex linear combination of the nodes in the neighborhood, weighted through the normalized attention coefficients, to which we apply a non-linearity.

Usually, to stabilize the learning process of the GAT layer, *multi-head attention* is exploited. In particular, K independent attention mechanism are trained and the final result is obtained performing an average (and applying the usual non-linearity):

$$h_i' = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{(k)} w^{(k)} h_j\right), \; i = 1,\dots,M \tag{6}$$

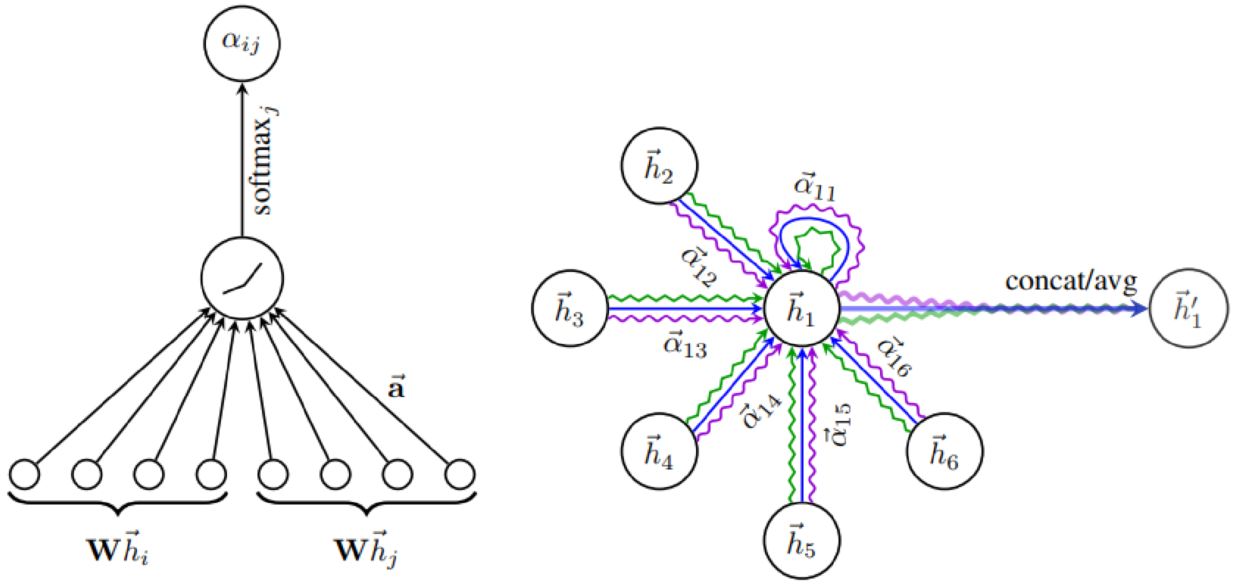


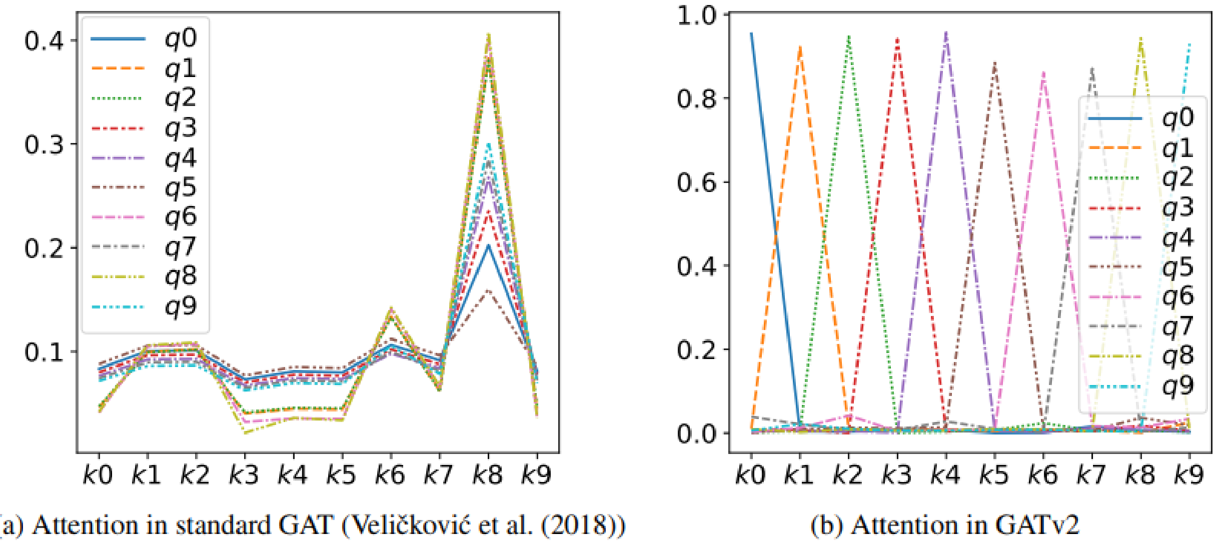
Figure 1: **Left:** Attention mechanism described in (4) for the vectorial case. For scalar case, just consider one neuron for h_i and one neuron for h_j. **Right:** Multi-head attention ($K = 3$) for node 1 and its neighborhood. Final features are averaged (but can be also concatenated for certain applications).

2.2. Graph Attention layer with dynamic attention (GATv2)

Brody et al. (2022) [4] showed that the GAT attention layer has some limitations though. In particular, it does not compute the more expressive *dynamic attention* but it is limited to the *static attention*. Without going too much in details (which are not difficult but require some technicalities), in the GAT layer, for any query node, the attention coefficients are actually *unconditioned* to the query node, but they are shared across all nodes in the graph.
So, for example, let us take two nodes i_1 and i_2, and indicate with $\mathcal{N}_* = \mathcal{N}_{i_1} \cap \mathcal{N}_{i_2}$, the intersection of the two neighborhoods. If it happens that $j \in \mathcal{N}_*$ is the most important node (highest normalized attention coefficient) for i_1 among all the nodes in $\mathcal{N}_*$, the very same node j will be the most important for node i_2 as well (again, among all the nodes in $\mathcal{N}_*$).
Of course, this heavily limits the expressiveness of the layer, as we can also see in the figure 2.
From the mathematical point of view, to obtain the GATv2 layer, it is sufficient to modify the expression (4) applying $\vec{\mathbf{a}}$ after the non-linearity (LeakyReLU) so that:

$$\alpha_{ij} = \frac{\exp(\vec{\mathbf{a}} \cdot LeakyReLU([wh_i || wh_j]))}{\sum_{k\in\mathcal{N}_i} \exp(\vec{\mathbf{a}} \cdot LeakyReLU([wh_i || wh_k]))} \tag{7}$$

This very simple modification allows the layer to compute the coefficients dynamically.



(a) Attention in standard GAT (Veličković et al. (2018)) (b) Attention in GATv2

Figure 2: Example with a complete bipartite graph of "query nodes" $q0, ..., q9$ and "key nodes" $k0, ..., k9$: standard GAT (Figure 2a) computes static attention, so the ranking of attention coefficients is global for all nodes in the graph, and is unconditioned on the query node. For example, all queries (q0 to q9) attend mostly to the 8th key (k8). In contrast, GATv2 (Figure 2b) can actually compute dynamic attention, where every query has a different ranking of attention coefficients of the keys (Brody et al. (2022)).

2.3. Feed Forward fully-connected layer

The second part of the model consists of a feed forward fully-connected with an input layer, a hidden layer and an output layer. It takes as input the transformed features provided by the convolutional layers and estimate the target values of each node as output. The high level structure is represented in figure 3. From now on, we may indicate the fully-connected layer with FC(n), with n indicating the number of hidden neurons (input and output neurons are given by the number of mesh vertices).

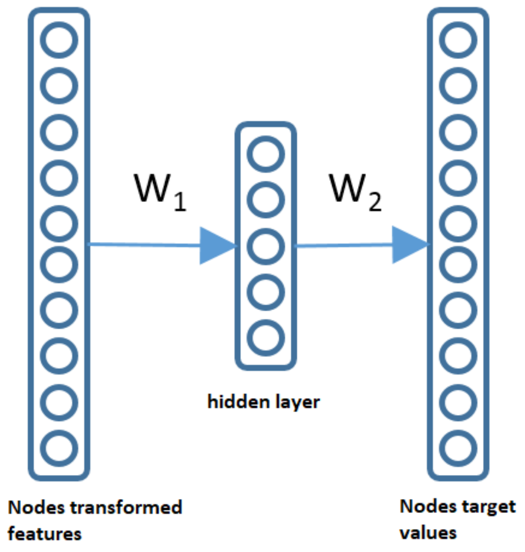


Figure 3: Fully-Connected layer

3. Numerical results

Let us discuss the numerical results we obtained from our analysis and comparisons. First of all, the reference Poisson's problem we use has the following parameters.

During training:

- $p_{min} = 2$, $p_{max} = 5$
- $M = 1089$, using a 32×32 mesh
- $N = 1000$, with uniform sampled $\{p_i\}_{i=1}^{N}$ in the interval $[p_{min}, p_{max}]$

During testing:

- $\tilde{p}_{min} = 2.01$, $\tilde{p}_{max} = 4.99$
- $\tilde{M} = 1089$, using a 32×32 mesh (same mesh used during training)
- $\tilde{N} = 50$, with uniform sampled $\{\tilde{p}_i\}_{i=1}^{\tilde{N}}$ in the interval $[\tilde{p}_{min}, \tilde{p}_{max}]$

The training and testing sets are generated by solving the above parametrized problems using `FreeFem++` [5]. Therefore, the train set consists of a thousand graphs, each one of them complete of the solution and the forcing term computed over all nodes of the mesh, and the connectivity. Analogously the test set has the same structure, but with just fifty instances.

As regard the model, we performed several comparison with different structures but here we just report the best configuration obtained:

1. Convolutional layer: we use 2 consecutives GATv2 layers, with 4 heads each ($K = 4$) with $\sigma = ReLU$. Since each head has 3 learnable parameters, namely $\vec{\mathbf{a}}$ and w, the total number of learnable parameters of this part of the model is:
$$\#parameters = 2 * 4 * 3 = 24$$
2. fully-connected layer: we use the same structure described in 2.3, using *ReLU* activation functions between input/hidden and hidden/output layers. For the output instead, we use the *Sigmoid* activation function since the output is bounded in $[0, 1]$, thanks to our normalization. The input and output layers have 1089 neurons, equal to the number of vertices of the mesh, and the hidden layer has 32 neurons. The total number of parameters here is:
$$\#parameters = (1089 * 32 + 32 + 32 * 1089 + 1089) = 70817$$

Note how small the number of convolutional layer parameters is. However, as we will show in the comparisons, the training time is quite remarkable anyway, since the attention coefficients must be computer for each couple of nodes (i, j) connected by an edge.

To develop our model, we used the *PyTorch Geometric* library [6]. Below you can see the structure of the model trough our code, where we also show the used loss, namely the *Mean Squared Error* and the learning optimizer, namely *Adam*:

```
class GAT(torch.nn.Module):

    def __init__(self):
        super().__init__()

        self.heads1 = 4
        self.conv1 = GATv2Conv(conv_input_size, conv_input_size, heads = self.heads1,
                                                   concat = False)

        self.heads2 = 4
        self.conv2 = GATv2Conv(conv_input_size, conv_input_size, heads = self.heads2,
                                                   concat = False)

        self.fc = torch.nn.Sequential(
            Linear(input_size, 32),
            ReLU(),
            Linear(32, 32),
            ReLU(),
            Linear(32, output_size),
            Sigmoid()
        )


    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        x = torch.relu(x)
        x = self.fc(torch.flatten(x))

        return x

GAT_model = GAT().double()
```

```
learning_rate = 1e-3
loss_fn = torch.nn.MSELoss() # Mean Squared Error LOSS
optimizer = torch.optim.Adam(GAT_model.parameters(), lr=learning_rate)
```

```
def train_loop(dataloader, model, loss_fn, optimizer):
    tot_loss = 0
    loss = 0
    for batch, data in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(data.x, data.edge_index)
        loss = loss_fn(pred, torch.flatten(data.y))

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        tot_loss += loss.item()
    print("loss: {:1.3e}".format(tot_loss / len(dataloader)))
```

```
epochs = 30
start = time.time()

for t in range(epochs):
    print("-------------------------------")
    print(f"Epoch {t+1}")
    train_loop(dataloader, GAT_model, loss_fn, optimizer)
print("-------------------------------")
print("Done!")

end = time.time()
elapsed_time = end - start
minutes = elapsed_time //60
seconds = elapsed_time - 60*minutes
print(f"Elapsed time: {minutes:2.0f} min, {seconds:2.0f} sec")
```

3.1. Performance of the model

We train the network described above for 30 epochs, which takes in total 10 minutes circa.
In table 1 we report the errors obtained on the training and test sets and in figure 4 we show the results on a sample of the test set. We gather that the GATv2-FC(32) network is able to capture the trend of the solution and, in general, predict it with a low degree of error at testing time.
So, overall this graph attention based model proved to be successful both for the low training time requested and the actual performance on the test set.

In the following sections we go a little bit in detail, explaining what the GATv2 and fully-connected layers are doing, and we also report comparisons between different models to prove that graph convolution is indeed useful.

GATv2-FC(32) model			
	Average	Minimum	Maximum
Training MSE	4.4e−4	2e−4	1.9e−3
Testing MSE	4.6e−4	3e−4	1.7e−3

Table 1: Mean Squared Errors of our best model GATv2-FC(32).

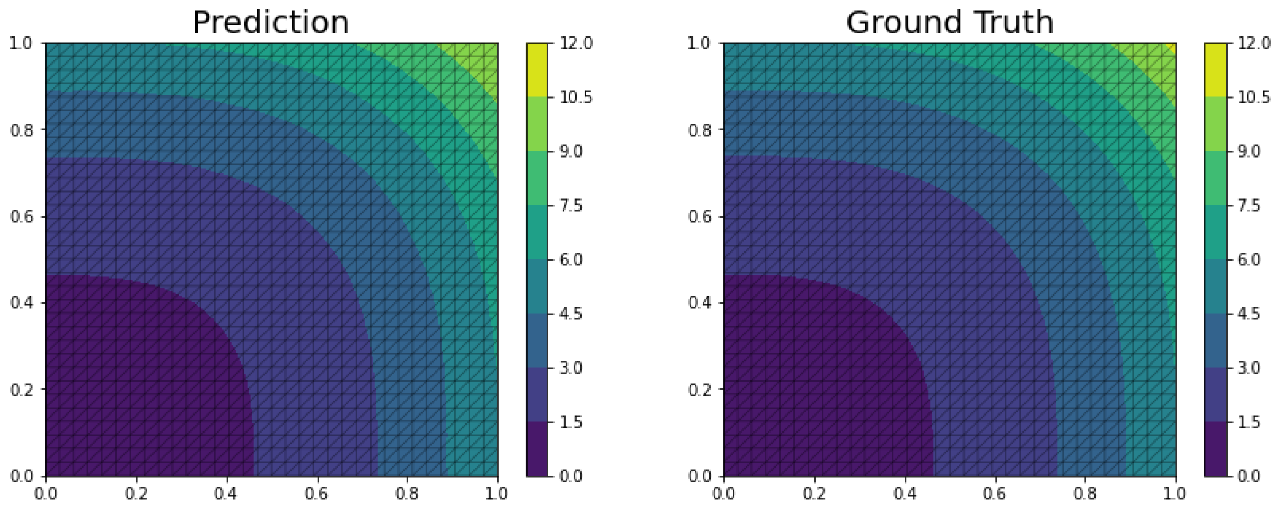


Figure 4: GATv2-FC(32) model prediction against ground truth solution for $p = 4.99$.

3.2. The effect of the convolutional and fully-connected layers

At this point a couple of questions come to mind: what is the convolutional layer GATv2 actually doing? Are the performance we discussed in the section 3.1 attributable to the whole model or, in fact, the fully-connected layer is doing all the work?

Let us now analyze the behavior of the model if we drop the fully-connected layer, so that we can understand how the convolutional layer transforms the features. In the figure 6, we show what the convolutional part of the model would predict, without applying the fully-connected layer. As we can see from the plots, GATv2 layers are able to learn the trend of the solution very well, even though just in a "linear" way. Nevertheless, we have to keep in mind that the number of parameters used in this part of the model is very small compared to the fully-connected layer (24 instead of 70817), but still the GATv2 layers are able to learn quite effectively the overall behavior of the parametrized Poisson's problem solutions.

However, we still have to verify whether the sole fully-connected layer could reach the same performance of our whole model or not.
To do so we train two models: the first one (FC(32)) keeps the same fully-connected structure of our best performing model, and gets rid of the convolutional part. The second one (FC(512)), instead, increments the number of hidden neurons up to 512, again without any convolution. Note that, the number of parameters in this latter configuration is huge, more than one million $(1, 116, 737$ to be exact) so we could expect this network to outperform all the other models, but, as we will show, that is not the case.

While the first one (FC(32)) is able to train significantly faster with respect to any other model we implemented (training only took 5 minutes), it achieves the biggest MSE (averaged), of 0.39. By looking at the prediction of a sample of the test set, reported in figure 5, and focusing on the two bar plots aside each solution, we can notice a huge prediction error. Actually, it is caused by the fact that the model is not able to predict effectively for the full range $[p_{min}, p_{max}]$, making big mistakes on some nodes, for specific values of p. However, to strike a blow for the FC(32), it is still able to grasp the general non linear trend of the solution (and that's why if it is combined with the GATv2 layers, performance are boosted so much).

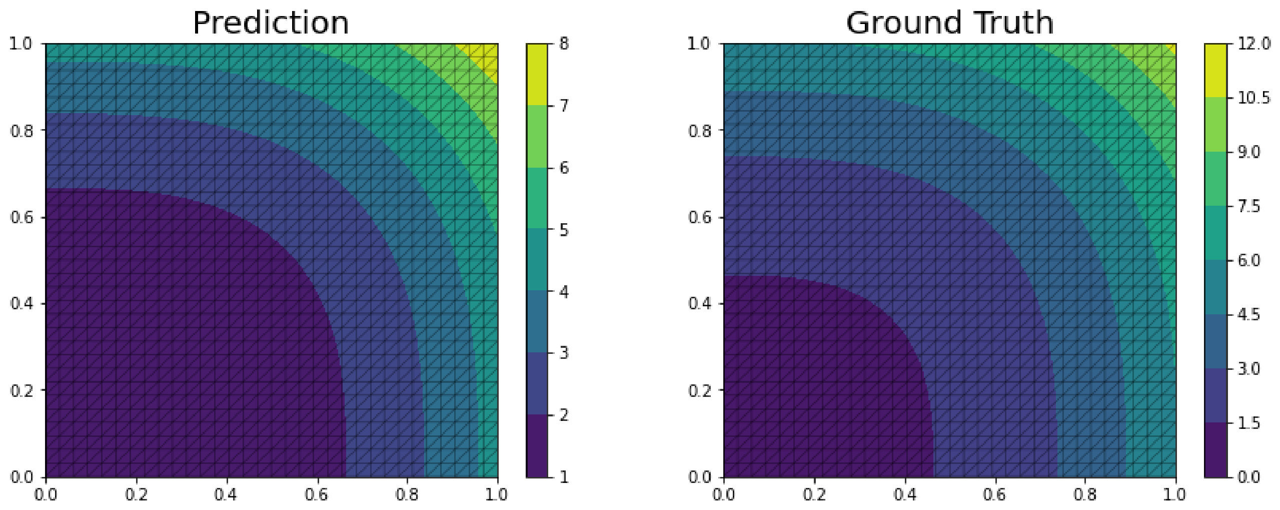


Figure 5: FC(32) model prediction against ground truth solution for $p = 4.99$.

On the other hand, the FC(512) network is able to achieve an MSE of 1.7e−3, much closer to the one of our best performing model, but it is still one degree of magnitude behind. Therefore, even though it has at disposal much more learnable parameters than the GATv2-FC(32) model, it does not manage to reach the same prediction accuracy, as we might have expected. Moreover, it takes almost double the time, completing the training in 20 minutes.

Mean Squared Error			
	Average	Minimum	Maximum
FC(32)	3.4e−1	2e−4	1.106
FC(512)	1.7e−3	2e−4	5.7e−3
GAT-FC(32)	1.8e−3	9.9e−5	3.8e−3
GATv2-FC(32)	4.6e−4	3e−4	1.7e−3

Table 2: Mean Squared Error indicators for different networks, on the test set.

	FC(32)	FC(512)	GAT-FC(32)	GATv2-FC(32)
Training Time	5 min, 10 sec	19 min, 58 sec	10 min, 58 sec	10 min, 22 sec

Table 3: Training times of the networks used.

4. Conclusions

To conclude, let us wrap up the remarks on the performance of our model.
First of all, looking at the errors reported in table 2, it appears clear that exploiting the mesh connectivity as an input feature of the network yields to much better predictions. In fact, both architecture using GAT layers score the lowest error among the ones we tested. In fact, even though the fully-connected network with 32 neurons trains in the least amount of time, it is not able to reach a satisfying performance when moving to the prediction task. On the other hand, the one with 512 neurons indeed achieves results somehow closer to the ones of the graph networks but the amount of training time is almost double (see table 3).

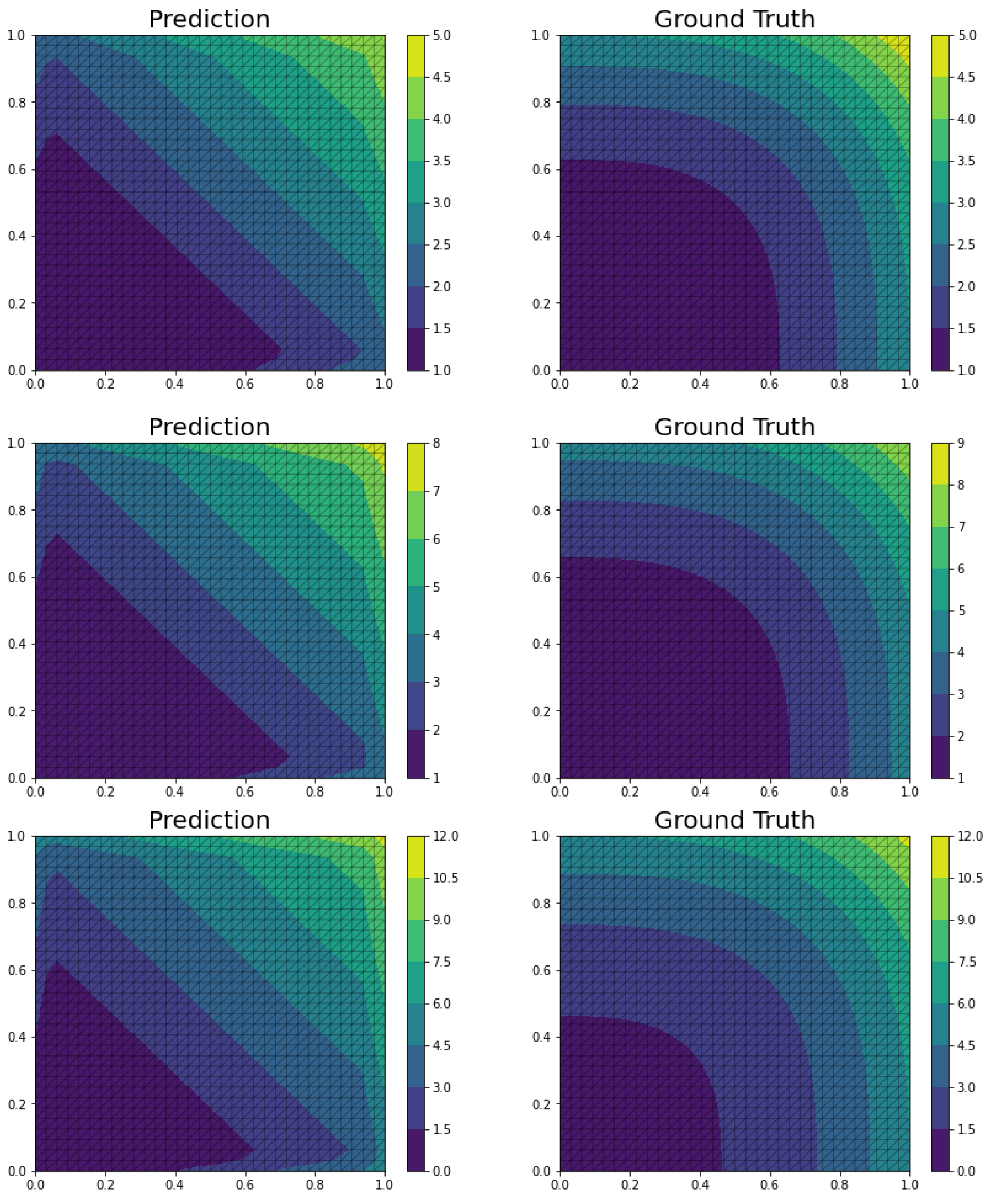


Figure 6: Effect of the GATv2 layer only, respectively for $p = 2$, $p = 3.5$, $p = 5$

As for the two graph neural networks we can see that they take the same amount of time for training, but the one using GATv2 layers outperforms the other, confirming that dynamic attention is determining to pool information correctly.

Finally, although the problem we studied is quite simple and easily solvable by most of numerical solvers, it convinced us that the use of graph neural networks when solving PDE problems can be crucial for achieving better performance and lower training time with respect to traditional neural networks. The same promising results were obtained over an unstructured mesh, reason why they were not reported here.

References

[1] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. 2020. doi: 10.48550/ARXIV.2010.03409. URL `https://arxiv.org/abs/2010.03409`.

[2] Wenzhuo Liu, Mouadh Yagoubi, and Marc Schoenauer. Multi-resolution graph neural networks for pde approximation. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 151–163, Cham, 2021. Springer International Publishing. ISBN 978-3-030-86365-4.

[3] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017. URL `https://arxiv.org/abs/1710.10903`.

[4] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2021. URL `https://arxiv.org/abs/2105.14491`.

[5] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012. ISSN 1570-2820. URL `https://freefem.org/`.

[6] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.