University of Strasbourg

Project report

Graph convolutional networks and applications

Corentin MENGEL

under the supervision of Vincent VIGON, Emmanuel FRANCK, Laurent NAVORET et Laurène HUME

Contents

1 Introduction

These past 15 years, Deep Learning has seen a massive gain in popularity. There is a lot of ongoing research about deep learning, and an enormous amount of work now includes it in their pipelines.

Deep learning is the automation of learning new tasks using neural networks. Various neural networks can be used, with the most populars being the multilayer perceptrons, the recurrent neural networks or the convolutional neural networks. Especially the convolutional neural networks (CNNs) can achieve outstanding results on tasks involving signals (images, video, text, sound...). They can also be used in the study of partial derivative equations. For example they can help reduce big models or detect numerical method failures.

However one big limitation of CNNs is that they cannot use unstructured data as input. For example it is not possible to feed a graph to a CNN. Graph Convolutional Networks (GCNs) are machine learning models that can be seen as a generalization of traditional CNNs. They use Fourier spectral theory to define convolution on graphs, and allow us to generalize methods used by CNNs to unstructured datasets. Particularly, GNNs can be used to solve problems like:

- Graph classification: we need to predict to which class the graph belongs to,
- Node classification: given a graph with incomplete node labeling, we need to predict the class of the remaining nodes,
- Community detection: we need to partition the nodes of the graph into clusters based on its edge structure.

During this project we focused on two problems. The first one is similar to graph classification, and the second one is a node classification problem. In our case, graphs will be 2D meshes. Currently in the scientific literature, it is stated that GCNs are topology dependent. It means that every time the graphs topology changes we have to retrain our GCN from scratch. However, what is not really talked about is what happens if we modify the graphs without changing their topology. So the main goal of this project is to study the stability of GCNs in relation to mesh variations.

2 Graph convolutional networks

Let us start by giving some general definitions about graphs and meshes. Moreover, we will define some graph neural networks' layers, and explain the model notations that we will be using throughout the report.

2.1 Graph definitions

A graph is a pair $G = (V, E)$, where V is a set whose elements are called **nodes**, and E is a set of paired nodes, whose elements are called **edges**. Let N be the number of nodes in G, i.e. the cardinal of V. We can then arbitrarily index the set E and note $(n_i)_{i=1}^N$ its elements. The degree of a node n_i is noted $d(n_i)$ and is the number of nodes n_j in V such that (n_i, n_j) is in E.
We can associate two $N \times N$ matrices to a graph: the **adjacency matrix** A and the **degree matrix** D. The degree matrix D is a diagonal matrix with diagonal $D_{ii} = d(n_i)$. The adjacency matrix A is defined by:

$$A_{ij} = \begin{cases} 1 & \text{if } (n_i, n_j) \in E \\ 0 & \text{else} \end{cases}$$

We can see that the diagonal of D is constituted of the sum of the respective rows of A. Moreover, we will only consider undirected graphs i.e. graphs such that $(n_i, n_j) \in E$ implies $(n_j, n_i) \in E$. As a result A is a symmetric matrix.
The **neighborhood** of a graph node n_i is the set of nodes n_j such that (n_i, n_j) is in E. Moreover, the p**-hop neighborhood** of a node n_i is the set of nodes that are reachable from n_i by following a path of p of fewer edges.

In this project we will use particular graphs called meshes. More precisely, we consider triangle meshes. A triangle mesh is a set of triangles in $\mathbb{R}^2$ that are connected by their common edges or corners. So we can see it as a graph whose nodes (n_i) in V are the corners of the triangles, and an edge $(n_i, n_j) \in E$ is represented by the segment $[n_i, n_j]$. Here are some examples of type of meshes that we will use in this project:

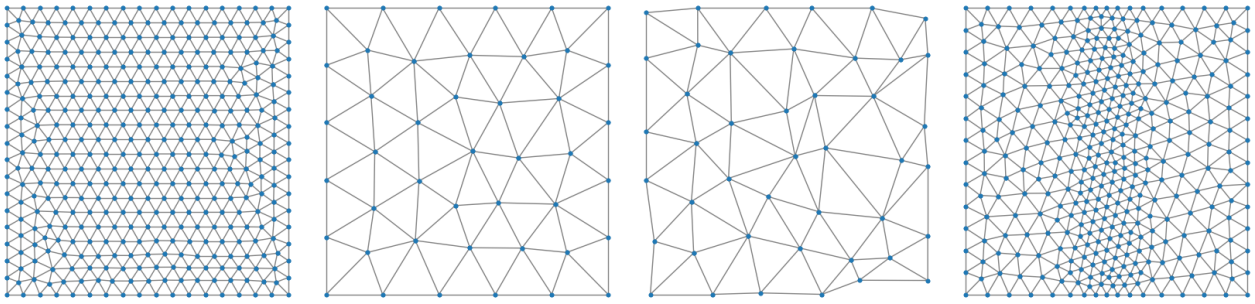

Figure 1: Examples of meshes.

2.2 Layers

Our models will be sequences of layers put one after the other, using the model object from the Keras library [1]. Each graph convolutional layer computes d' dimensional representations

for the nodes of the graph through recursive neighborhood diffusion and message passing, where each graph node gathers features from its neighbors to represent local graph structure. This way, stacking p GCN layers allows the network to build node representations from the p-hop neighborhood.
More precisely, let $X_i^l\mathbb{R}^d$ denote the feature vector of the node n_i at the layer l. Then the updated features $X_i^{l+1}\mathbb{R}^{d'}$ at the next layer $l+1$ are obtained by applying non-linear transformations to the central feature vector X_i^l and the feature vectors X_j^l for all the nodes n_j in the neighborhood of node n_i.
This way, the network builds local reception fields, like in standard convolutional layers, and more importantly is invariant by graph size and nodes re-indexing.

2.2.1 Vanilla GCN

The Vanilla GCN layer is a graph convolutional layer. It is one of the simplest GCN layers and updates node features via an averaging operation over the neighborhood node features. It takes node features $X \in \mathbb{R}^{N\times d}$ and an adjacency matrix $A \in \mathbb{R}^{N\times N}$ as input. The Vanilla GCN layer computes $X' \in \mathbb{R}^{N\times d'}$ by:

$$X' = \eta\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}XW + b\right)$$

where $\eta : \mathbb{R} \to \mathbb{R}$ is an activation function applied on each component of its input, $\hat{A} = A + I_N$ is the adjacency matrix of the graph G with self-loops added and $\hat{D}$ its degree matrix. The matrix $W \in \mathbb{R}^{d\times d'}$ is a trainable weights matrix, and $b \in \mathbb{R}^{d'}$ is trainable bias vector.
As we can see this layer computes for each node a weighted average of the feature of the nodes in his neighborhood.
In our models, a Vanilla GCN layer having d' output channels will be noted `GCN(d')`. This layer is implemented in Spektral [4].

2.2.2 ChebConv

Like before we denote by $X \in \mathbb{R}N \times d$ the input node features and $A \in \mathbb{R}N \times N$ the adjacency matrix. The ChebConv layer computes:

$$X' = \eta\left(\sum_{k=0}^{K-1} T_k(\hat{L})W_k + b_k\right),$$

where $\eta : \mathbb{R} \to \mathbb{R}$ is an activation function and $T_0, T_1, \ldots, T_{K-1}$ are Chebyshev polynomials defined as:

$$T_0(\hat{L}) = X, \quad T_1(\hat{L}) = \hat{L}X, \quad T_k(\hat{L}) = 2\hat{L}T_{k-1}(\hat{L}) - T_{k-2}(\hat{L}) \text{ for } k \geq 2,$$

and where:

$$\hat{L} = \frac{2}{\lambda_{\max}}(I_N - D^{-1/2}AD^{-1/2}) - I_N$$

is the normalized Laplacian of the graph G. Here $\lambda_{\max}$ denotes the biggest eigenvalue of $I_N - D^{-1/2}AD^{-1/2}$. This way $\hat{L}$ has its eigenvalues between -1 and 1.
The $W_k \in \mathbb{R}^{d\times d'}$ are K trainable weights matrix, and the $b_k \in \mathbb{R}^{d'}$ are K trainable bias matrices. The parameter K defines the number of hops in which each node gathers features from its neighbors.
A ChebConv layer with d' output channels will be denoted by `Cheb(d')` in the rest of the report. This layer is implemented in Spektral [4].

2.2.3 Dense

The dense layers are not specific to GCNs, but are layers used in classical neural network. They take as input a vector $X \in \mathbb{R}^p$ and output a vector $X' \in \mathbb{R}^q$ via the following operation:

$$X' = \eta\left(XW + b\right)$$

where $\eta : \mathbb{R} \to \mathbb{R}$ is an activation function, $W \in \mathbb{R}^{p \times q}$ is trainable weight matrix and $b \in \mathbb{R}^q$ is a trainable bias vector. A dense layer outputting a vector of size d' will be noted `FC(d')`, FC for fully-connected. This layer is implemented in Keras [1].

2.2.4 GlobalAveragePool

The GlobalAveragePool layer is a reduction layer, which means that it does not compute anything new from its input. On the contrary it will reduce the dimensions of its input matrix by averaging it along an axis. For example if the input of the GlobalAveragePool layer is a node feature matrix $X \in \mathbb{R}^{N \times d}$, then the output vector $X' \in \mathbb{R}^d$ is computed by:

$$X' = \frac{1}{N} \sum_{i=0}^{N-1} X_i$$

The GlobalAveragePool layer will be noted `GAvg` in the rest of the report. This layer is implemented in Spektral [4].

2.2.5 Model notation

We can describe a model by a sequence of layers, and their output size. For example the notation:

`Cheb(32)-Cheb(64)-Cheb(128)-GAvg-FC(256)-FC(10)`

indicates a model composed of three graph convolutional layers, a global average pooling layer, followed by two fully-connected layers. This model takes nodes features and adjacency matrices of different sizes as input, and outputs vectors of size 10.

2.3 Meshes

During this project we used lots of various meshes. They all are meshes of the unit square. In this section I will present the different kinds of meshes we used, and explain the parameters used to generate them.
We used PyGMSH, a GMSH python interface, to generate all the graphs used in this report. This allows us a great flexibility in the generated meshes.

2.3.1 Regular mesh

This is a regular mesh of the unit square. The only parameter is the characteristic elements' length, named h. It determines the average diameter of the triangles in the mesh. Here are some regular meshes:

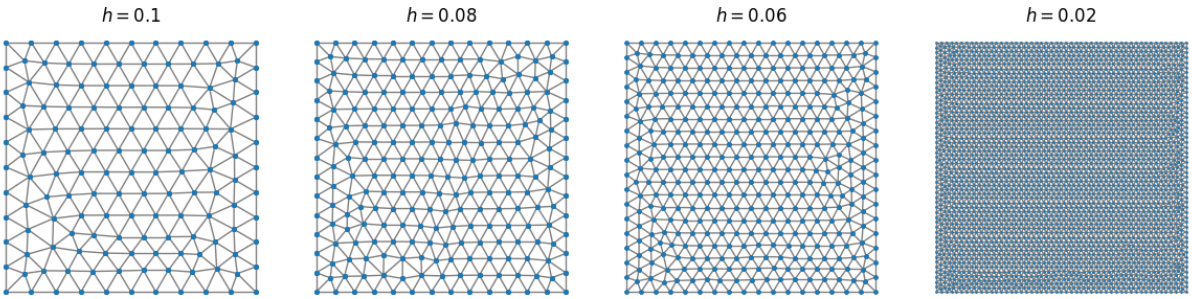

Figure 2: Regular meshes of various sizes.

2.3.2 Randomized mesh

A randomized mesh is a regular mesh to which we have moved each node in a random direction. The nodes positions are then clipped between 0 and 1 so that our mesh stays in the unit square. This mesh is determined by two parameters:

- $h > 0$, the base mesh characteristic elements' length,
- $\alpha > 0$, a number determining how much the nodes are moved.

More precisely, every node n_i of position $(x_i,\, y_i)$ of the regular mesh is moved in a random direction $(u_i,\, v_i)$ by the transformation:

$$(x_i,\, y_i) \leftarrow (x_i,\, y_i) + \frac{h}{\alpha}(u_i,\, v_i).$$

The direction $(u_i,\, v_i)$ is a vector whose components are random numbers between -1 and 1. As a result, the smallest α is, the more random the final mesh will be. Let us have a look at randomized meshes:

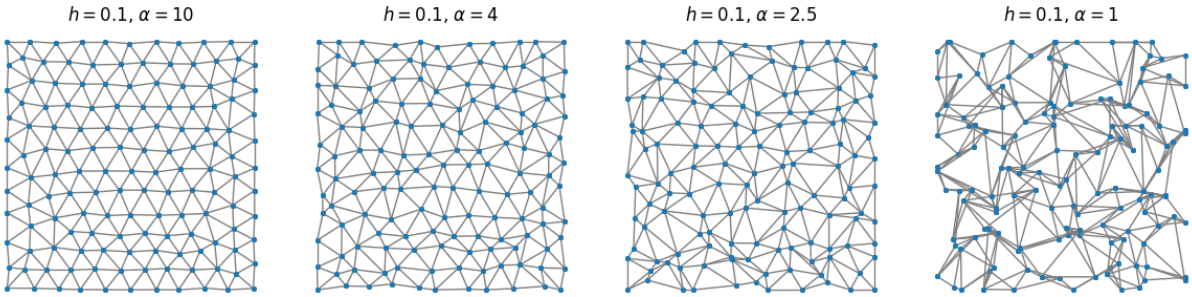

Figure 3: Randomized meshes with various parameters.

By definition if α is smaller than 2, the resulting graph can have edges crossing each others.

2.3.3 Locally refined mesh

A locally refined mesh is a mesh who has a smaller characteristic elements' length on a localized part of the unit square. In our program, we used meshes refined along a segment, or meshes refined around a specific point. These meshes are determined by 4 parameters:

- $h > 0$, the base mesh characteristic elements' length,
- $h_0 > 0$, the characteristic elements' length at the refining location,

- x_0 and y_0, optional parameters in [0, 1].

The diameter of a triangle of the refined mesh depends on its position in the unit square. More precisely, if both x_0 and y_0 are defined, the diameter of a triangle around the position (x, y) will be:

$$\min(h, h_0 + (x - x_0)^2 + (y - y_0)^2).$$

This results in a refined mesh around the point (x_0, y_0).
If only one of the two parameters x_0 and y_0 is defined, for example x_0, the diameter will be:

$$\min(h, h_0 + (x - x_0)^2),$$

resulting in a mesh refined along the segment $x = x_0$. Here are some examples:

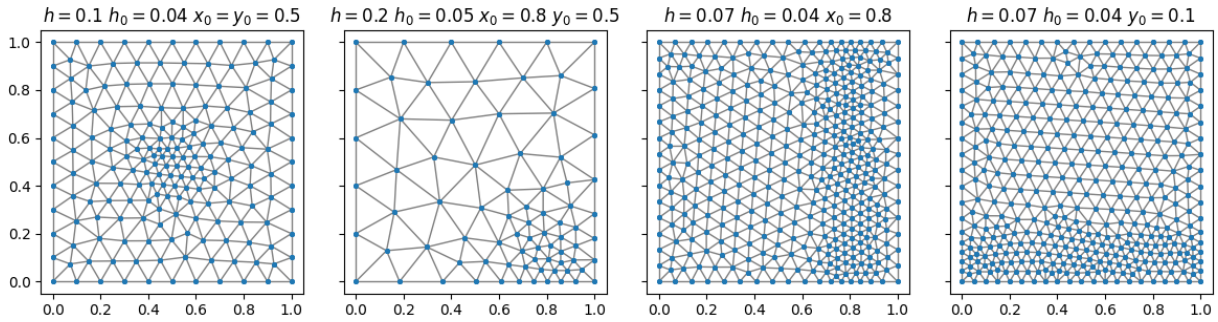

Figure 4: Locally refined meshes with various parameters.

3 Classification: MNIST dataset

The first goal of the project was to test GCNs on a simple classification problem. We chose the MNIST dataset [5], which is a dataset 28×28 images of handwritten digits. Here are 4 images from the dataset:

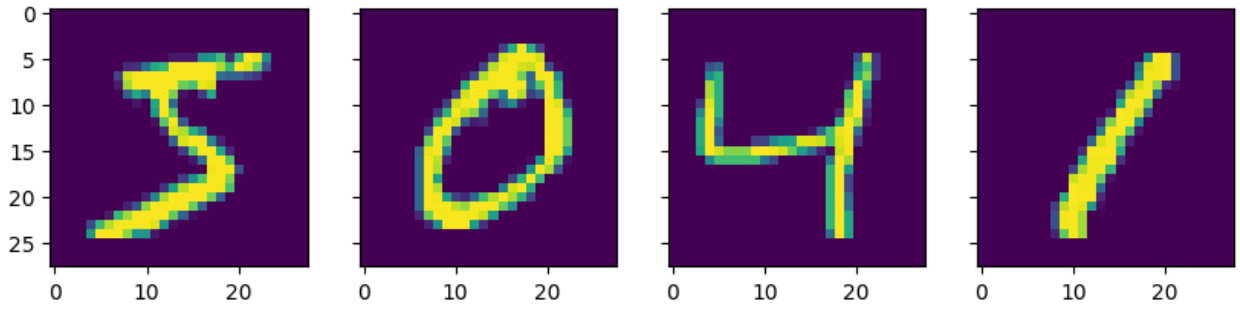

Figure 5: Images in the dataset MNIST.

Given an image from the dataset, the problem it to correctly predict which digit is written on the image. However, this dataset cannot be used as is by a GCN. In fact GCNs need graphs and nodes/edges features as inputs. So the first step was to extract the informations from the images and convert it into a usable format.

3.1 Image on mesh

The original 28×28 image can be seen as a scatter of points in $\mathbb{R}^2$, where each pixel of coordinate (i, j) is corresponding to the point $(i/27, j/27)$. This transformation sends the image in the unit square $[0, 1]^2$. To each pixel corresponds a value between 0 and 1, indicating the level of gray of the pixel (0 is black, 1 is white).
Now let M be a mesh in the unit square i.e. a mesh having its border on the border on the unit square. For each node of our mesh we need to compute a value based on the pixels of the images near the node. To do this we will associate every pixel in the unit square to the closest node of our mesh. By doing so we obtain Voronoi cells. So by construction, there is a bijection between the Voronoi cells and the nodes of the mesh. Here is an example with a simple mesh:

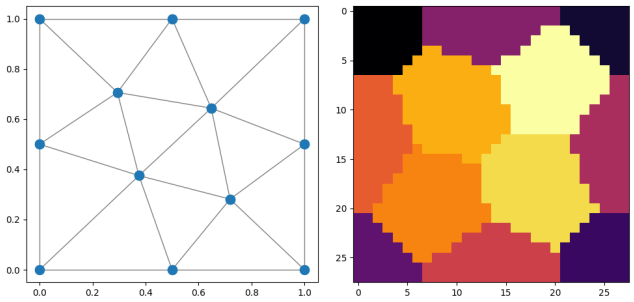

Figure 6: Mesh on the left, Voronoi cells on the right.

Now, to each node of the mesh, we will associate the mean of the pixels values in the Voronoi cell. As a result we obtain such nodes features:

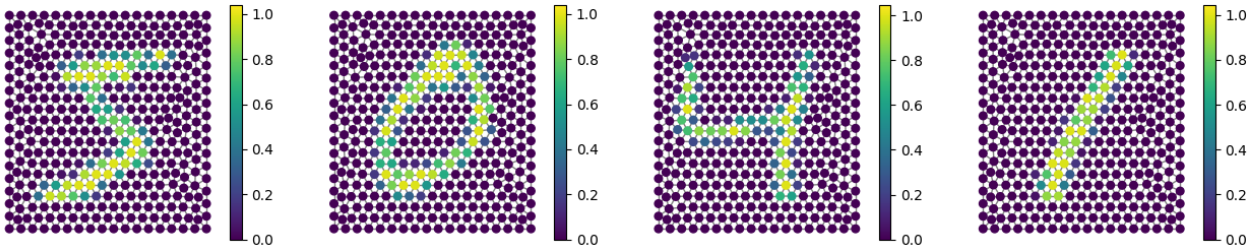

Figure 7: MNIST images transformed into node features on a regular mesh.

3.2 First attempt

Our first attempt to solve the MNIST classification problem involved only one mesh M. The mesh M is a regular mesh of the unit square, with a characteristic element length equal to $h = 0.06$ and $N = 379$ nodes. All the images of the dataset where transformed into node features for this specific mesh M, using the process explained in the previous section. This way each element of our new dataset consists in a vector of size N, containing the mean value of the pixels around each node of M. Examples of such node features can be seen on Figure 7.

The dataset is divided into 3 parts: a training dataset, a validation dataset and a test dataset. The train dataset is the dataset on which the model is trained using back-propagation. The validation dataset permits us to evaluate the model on data he did not see during its training, and adjust some parameters of the model so that it gets better results on the validation dataset. It also allows us to determine if our model is able to generalize, i.e. to be accurate on new data. The test set is the final dataset used to evaluate our model.

The model giving us the best results on the validation dataset for this first attempt is the following:

`GCN(32)-GCN(64)-GCN(128)-GCN(256)-GAvg-FC(256)-FC(10).`

All the layers have a relu activation, except the last one having a softmax activation. We use Vanilla GCN convolutional layers. In fact, the ChebConv layer gave us very similar results, but takes more time during training.
This model gives us an accuracy of **81.6%** on the test dataset. However, the test dataset is only composed of the same mesh used during training. We can try to evaluate the model on different meshes to measure the fluctuations in accuracy. To do this we chose the 4 following meshes:

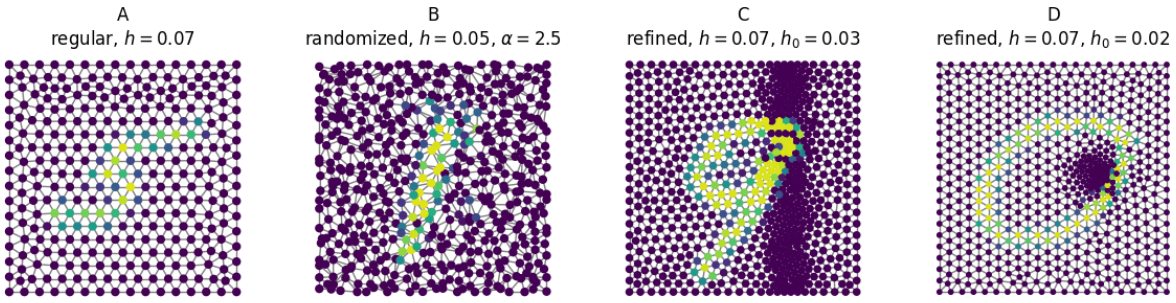

Figure 8: The 4 meshes on which to evaluate our trained models.

For simplicity, the meshes are named A, B, C and D from left to right. We evaluated the

model on 2000 images for each mesh. Here are the results:

A	B	C	D
37.05	21.05	33.90	38.05

Figure 9: First model performances on the 4 meshes.

The model drop from 81% accuracy on a regular mesh of size $h = 0.06$ to an accuracy of 37% on a regular mesh of size $h = 0.07$. Moreover the accuracy drops on the other meshes too. This proves that our current model is not robust at all against global/local refining/deformations.

3.3 Second attempt

The results showed us that our model still has room for improvement. We read in [3] that GCNs can perform better when having the nodes positions incorporated to the node features. This way, each node has a vector of size 3 as features: the first coefficient is the average of the pixels around the node, and the two last ones are the position of the node.
By modifying the input of our model this way, and keeping the same architecture, we managed to get much better results than in the first attempt. In fact the model now obtains an accuracy of **96.5%** on its test dataset. We can also evaluate the new model on the four test meshes. Here are the results:

A	B	C	D
95.15	91.30	74.90	68.90

Figure 10: Second model performances on the 4 meshes.

As you can see, the model barely looses any accuracy on the first and second mesh. However, the model still looses 30 to 40 percent of accuracy on the refined meshes. In fact those two meshes are locally really different at the refined nodes compared to the mesh used during training.
In order to change this, we train a new model but this time on multiple random meshes, and not only one. The 20 random meshes used are plotted on the Figure 12. We generated these meshed using the 3 type of meshes presented before, and using random parameters each time. The MNIST dataset is split in 20 equal parts, where each part is associated to a mesh. This way the model will see different meshes during training.
The model trained on those 20 meshes obtains an accuracy of **96.8%** on the test dataset, which is about the same as the previous model. However, the model loses only about 7 to 10 percent accuracy on the refined meshs:

A	B	C	D
98.50	95.65	89.75	87.30

Figure 11: Performance of the model trained on the 20 random meshes.

All the models trained on the MNIST dataset took about 45 minutes of training on a GeForce GTX 780 graphics card. The prediction of a model on a mesh takes a few milliseconds.

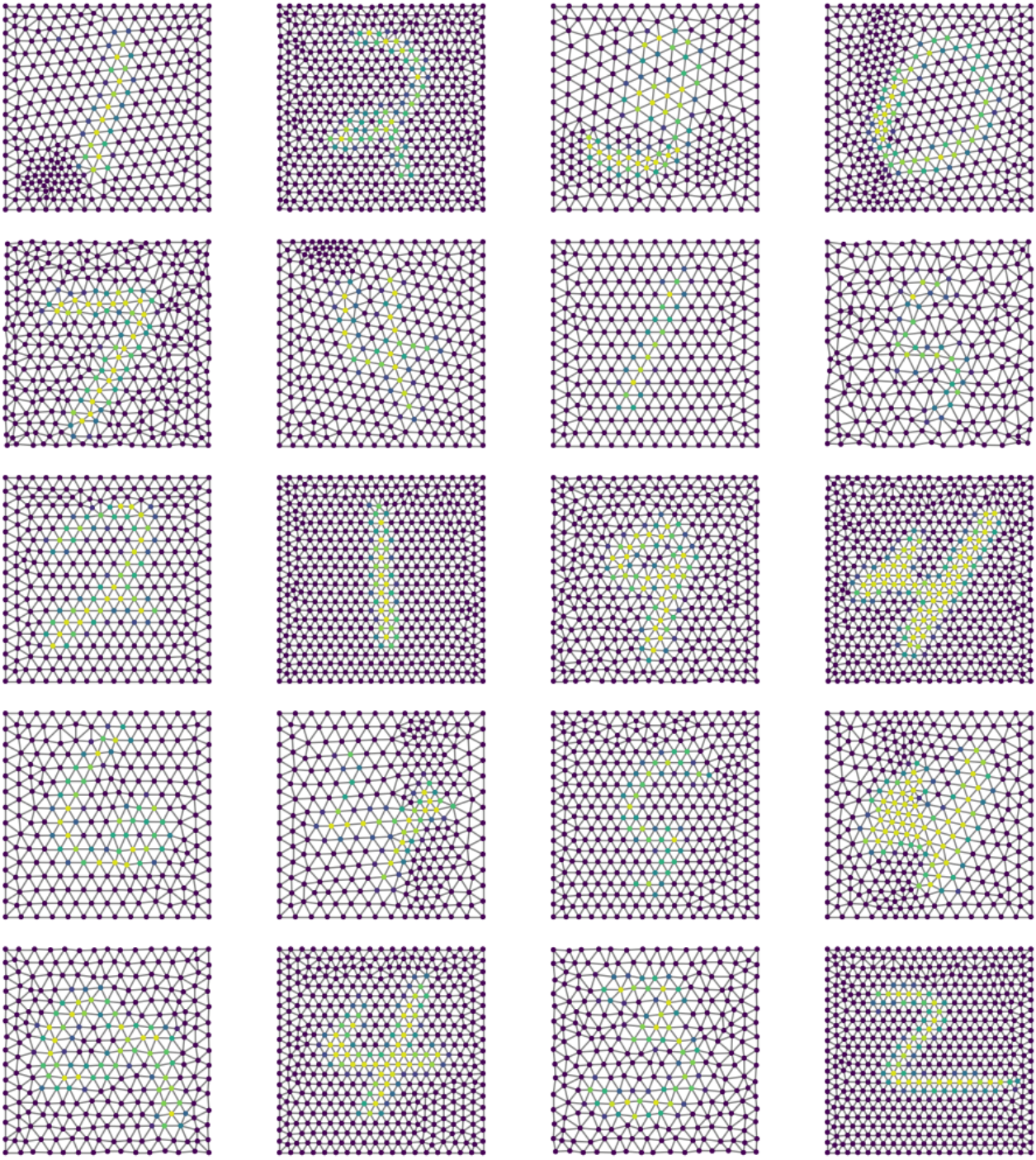

Figure 12: The 20 random meshes.

3.4 Robustness measures

As we have seen previously, our last model trained on 20 random models is able to keep a high accuracy even on graphs it did not train on. In this section we will measure the robustness of the model by evaluating it on meshes more and more deformed, and keeping track of the model's accuracy.
In order to be able to compare those results, we trained a second model but this time on 20 picked meshes. The 20 meshes consist of:

- 5 regular meshes of different sizes,
- 4 randomized meshes of different sizes and ratios respectively equal to 6, 5, 3, and 2.5,
- 4 meshes refined on a horizontal segment,
- 4 meshes refined on a vertical segment,
- 3 meshes refined around a point.

The meshes are available on Figure 15. As you can see the chosen meshes are more refined and more randomized than the 20 random ones. Thus, we can expect the new model to be more robust against deformations. The new model trained on those 20 picked meshes has an accuracy of **95.3%** on its test dataset.

3.4.1 Robustness on randomized meshes

We first propose to measure the models' accuracy on meshes more and more randomized. The meshes considered will all have the same characteristic elements' length $h = 0.06$. However, the parameter α will take 20 values from 0.25 to 4.0 included. A plot of all the meshes used is available on Figure 16. Here are the results obtained:

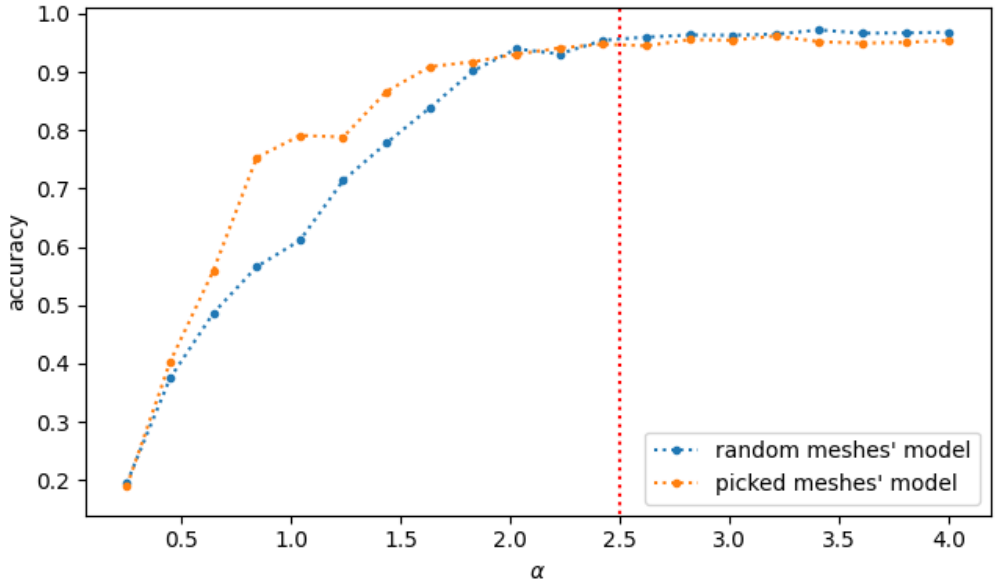

Figure 13: Accuracies of the models when α varies.

The vertical red bar indicates the minimal α of the randomized meshes in the 20 picked meshes. It represents the most randomized mesh the second model has seen during training.

The most randomized mesh the first model has seen during training was a mesh with $\alpha \geq 4$. On Figure 13 we can see that the second model is more robust than the first one. In fact it was trained on more randomized meshes. We can also observe that the accuracy of the first model starts dropping for $\alpha \leq 2$. That is when edges of the randomized meshes start crossing each others.

3.4.2 Robustness on refined meshes

We will now evaluate the robustness of the two models on meshes more and more refined. The meshes used for the evaluations are displayed on Figure 17. They are refined meshes with parameters $h = 0.06$, $x_0 = 0.5$ and h_0 taking 20 values from 0.01 to 0.06 included. Here are the results:

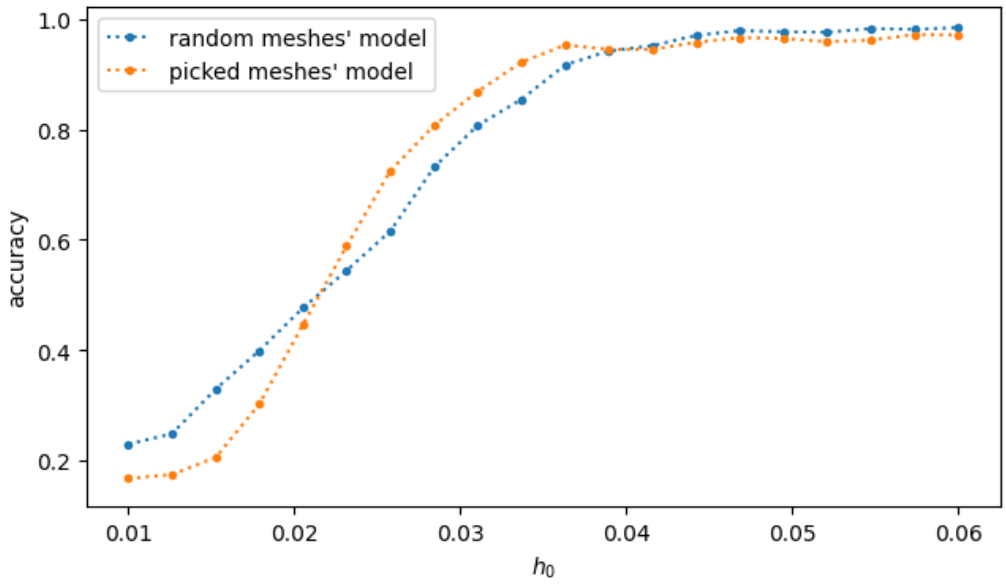

Figure 14: Accuracies of the models when h_0 varies.

As you can see on the figure, the second model is a little more robust than the first one against local refining. However, the accuracy drop is more abrupt than for the first one.

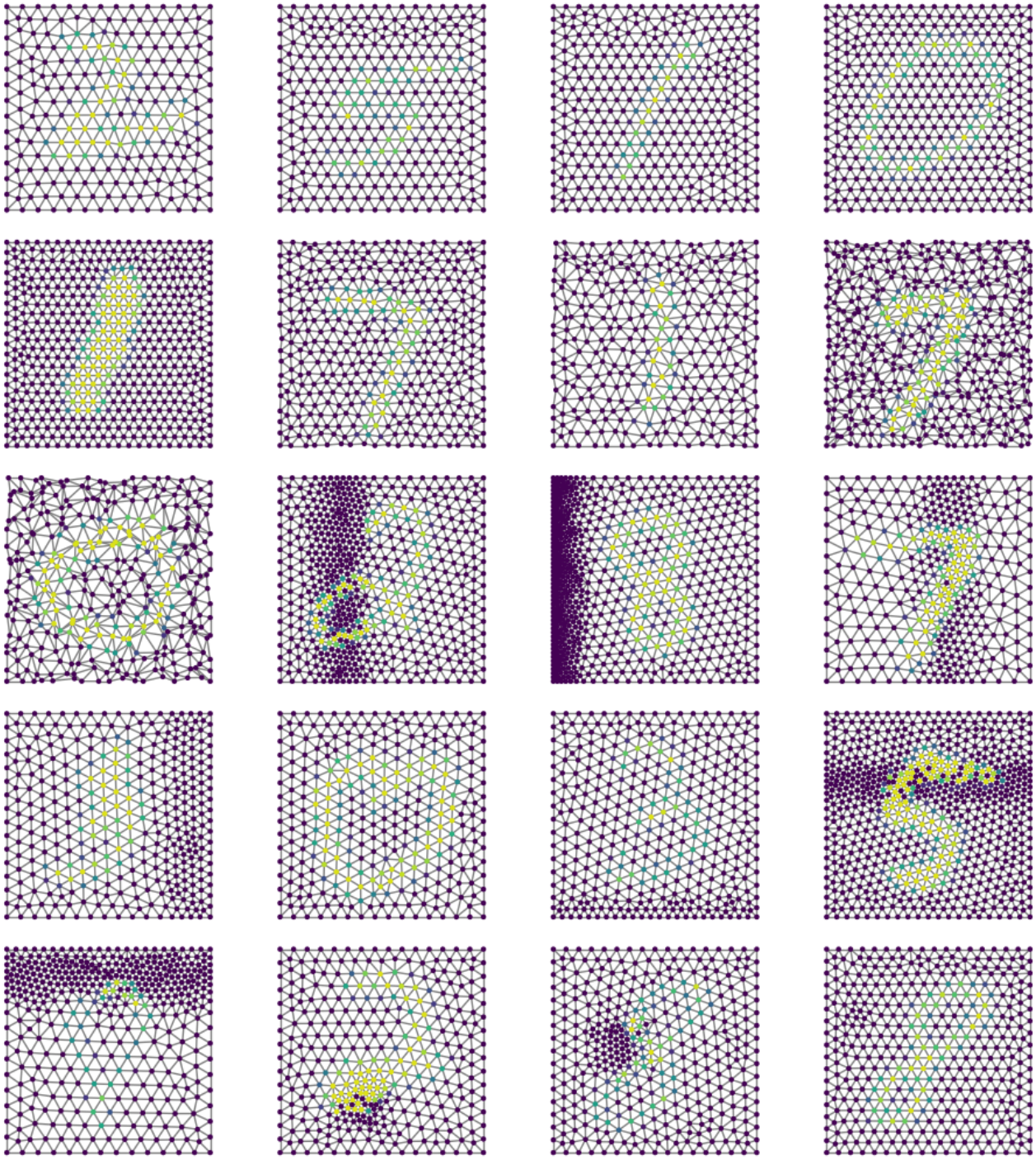

Figure 15: The 20 picked meshes.

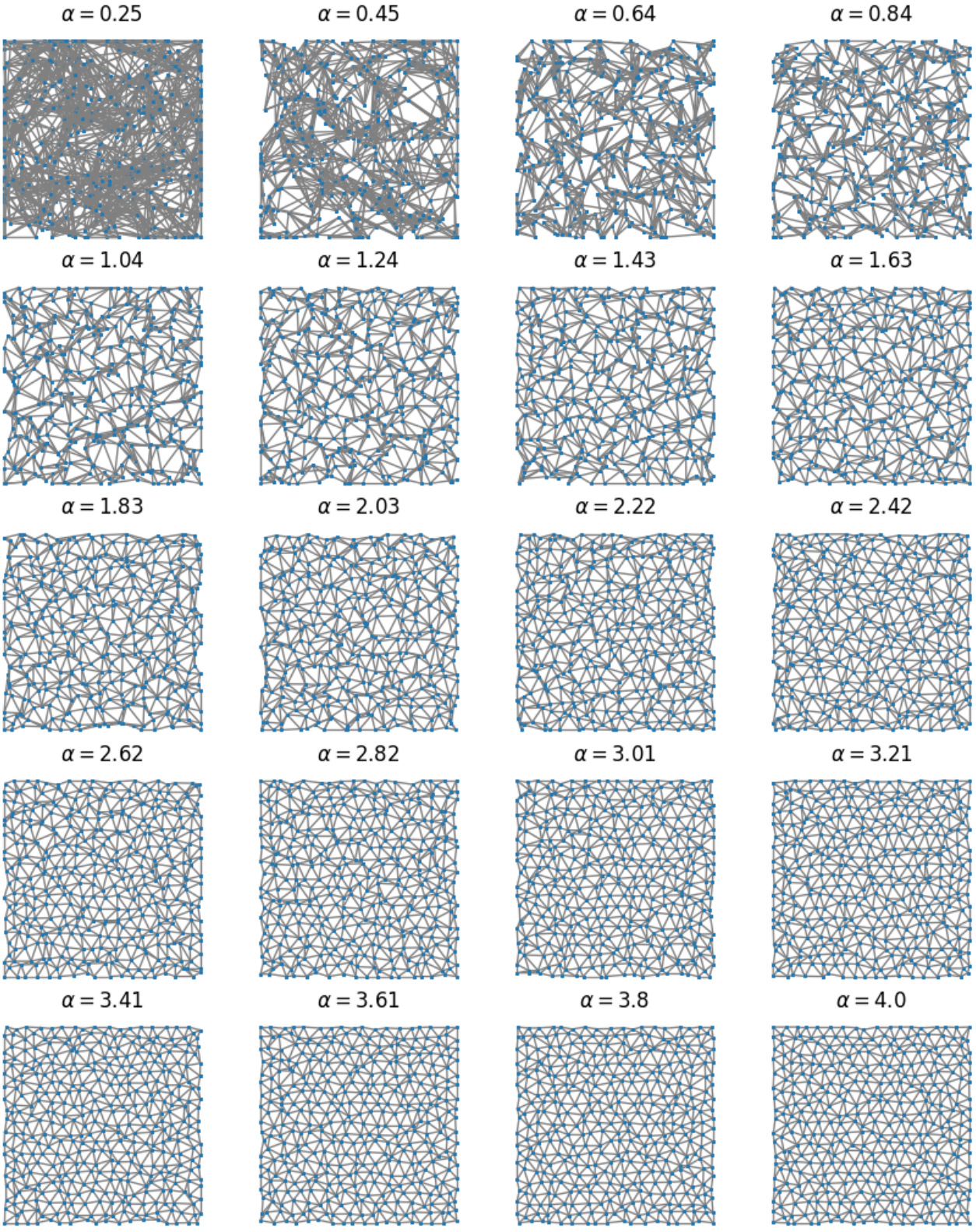

Figure 16: The 20 randomized meshes used for the measures, $h = 0.06$.

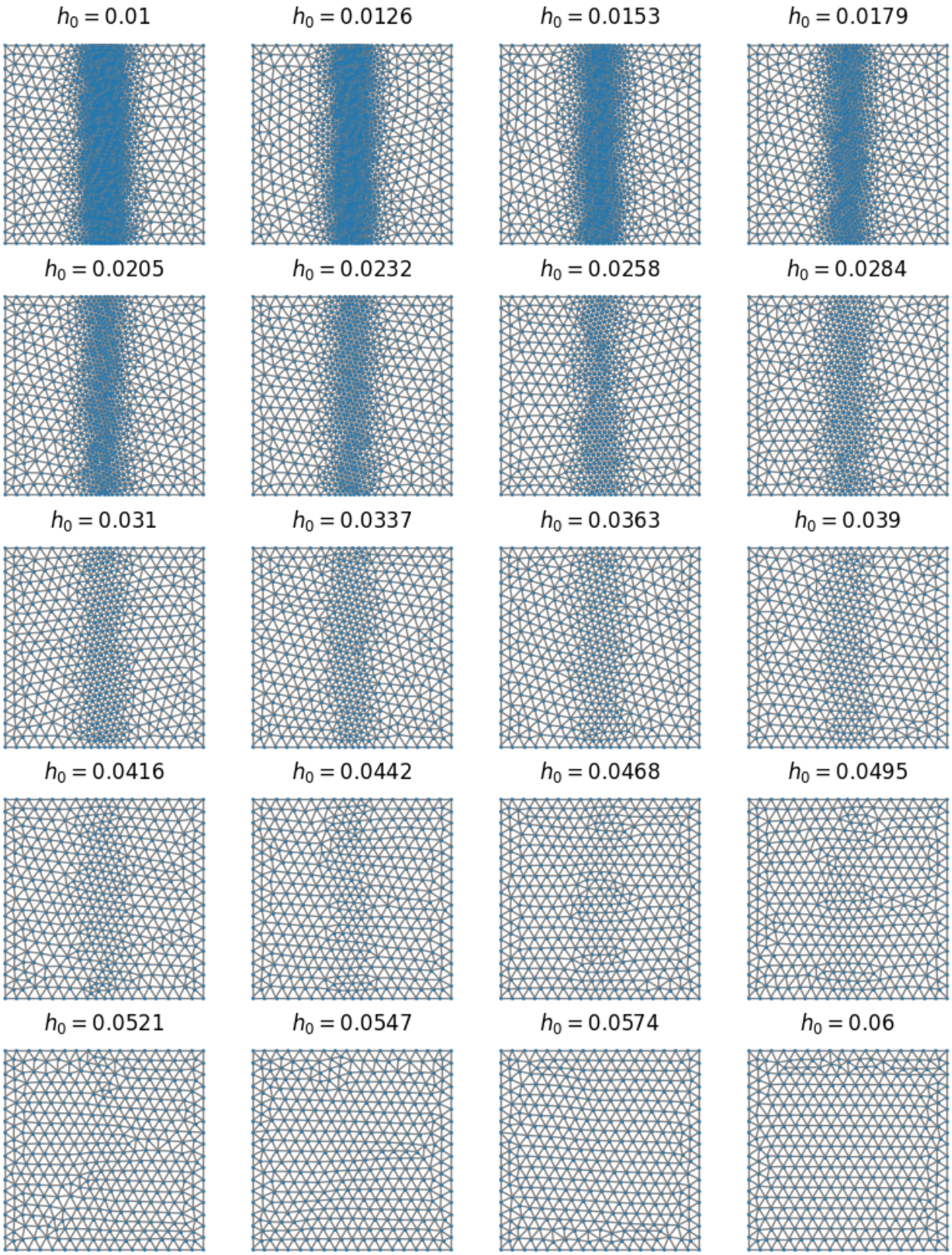

Figure 17: The 20 refined meshes used for the measures, $h = 0.06$.

4 Frontier detection

This dataset consists of images in which 2 areas can be clearly identified. More precisely this dataset can be divided into two separate datasets. In the first dataset, named trivial dataset, the two areas of the images are two different constant colors. Here are some example images from this dataset:

Figure 18: Examples of images in the trivial dataset.

The second dataset, named islands dataset, is a more complex one. Here are some images from this dataset:

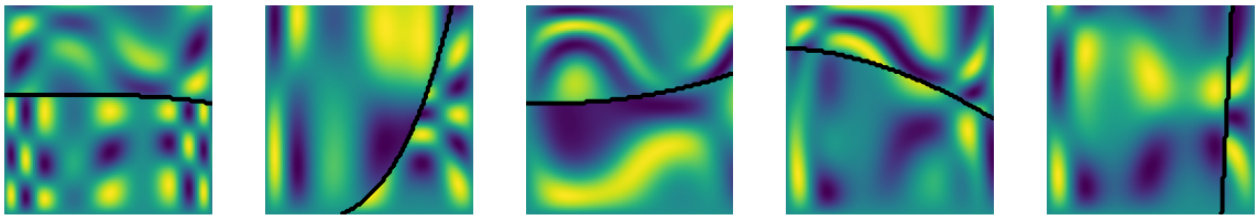

Figure 19: Examples of images in the islands dataset.

Given an image of the dataset, the problem is to correctly predict the frontier between the two areas of the image. As you can see the difference between the two areas is more difficult to perceive in the islands dataset. As a result, finding the frontier between the areas will be harder than in the trivial dataset.
Regular convolutional networks can easily solve this problem. For example a popular architecture used to solve this is the Vnet architecture. This architecture internally computes representations of the input image, with a decreasing resolution. When the minimal resolution is obtained, it constructs the output image by upscalling and combining the different representations previously computed.
For this problem, our models are trained on a single mesh depending on the dataset considered. The models need to predict where the frontier is on the inputted meshes. As a result they need to output a matrix the same size as the input, composed of numbers between 0 and 1 indicating the probability of a node being on the frontier.

4.1 Node features

The frontier dataset is dynamically generated i.e. the images are generated during execution time. The function generating the data takes the positions of the pixels as input, and returns a value between -1 and 1 for each pixel. Thus, to obtain usable data with graph convolutional networks, we can directly feed the nodes positions to the function. This way the resolution of the image obtained depends of the number of nodes in our mesh. On Figure 20 are a couple examples of the trivial dataset node features on different meshes.

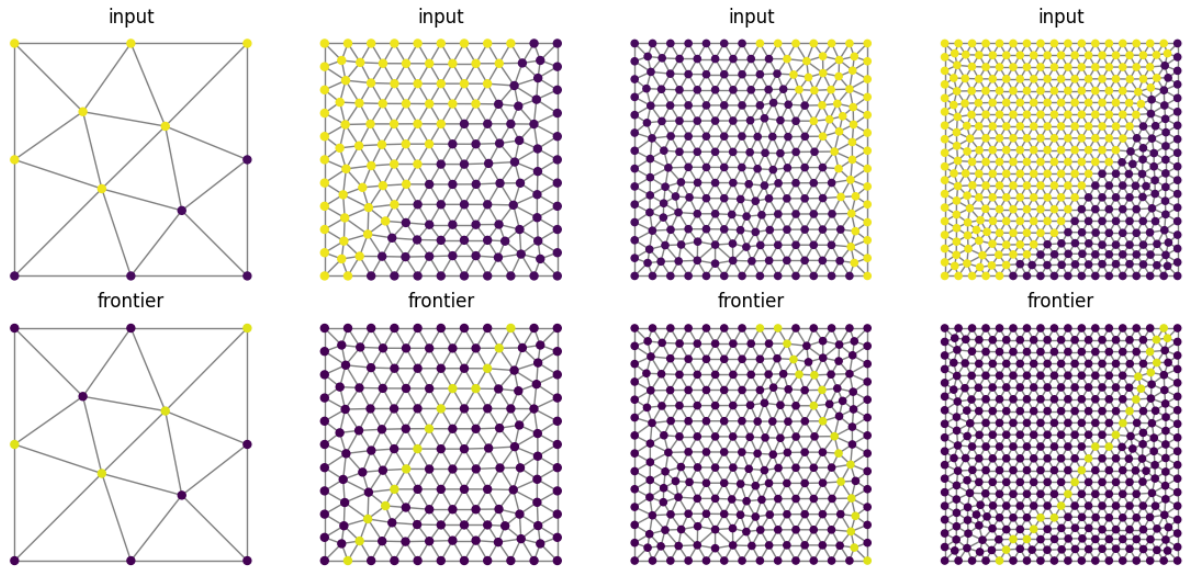

Figure 20: Meshes sizes are from left to right $h = 0.5$, $h = 0.1$, $h = 0.08$ and $h = 0.06$.

As we can see, the size of our mesh is not really important. We can visually determine where the frontier is even when the mesh does not have a lot of nodes.
Now let us observe some node features obtained from the islands dataset. The meshes are displayed on Figure 21. This time the size of our mesh plays an important role. In fact for the two smallest meshes we cannot even visually see the frontier.

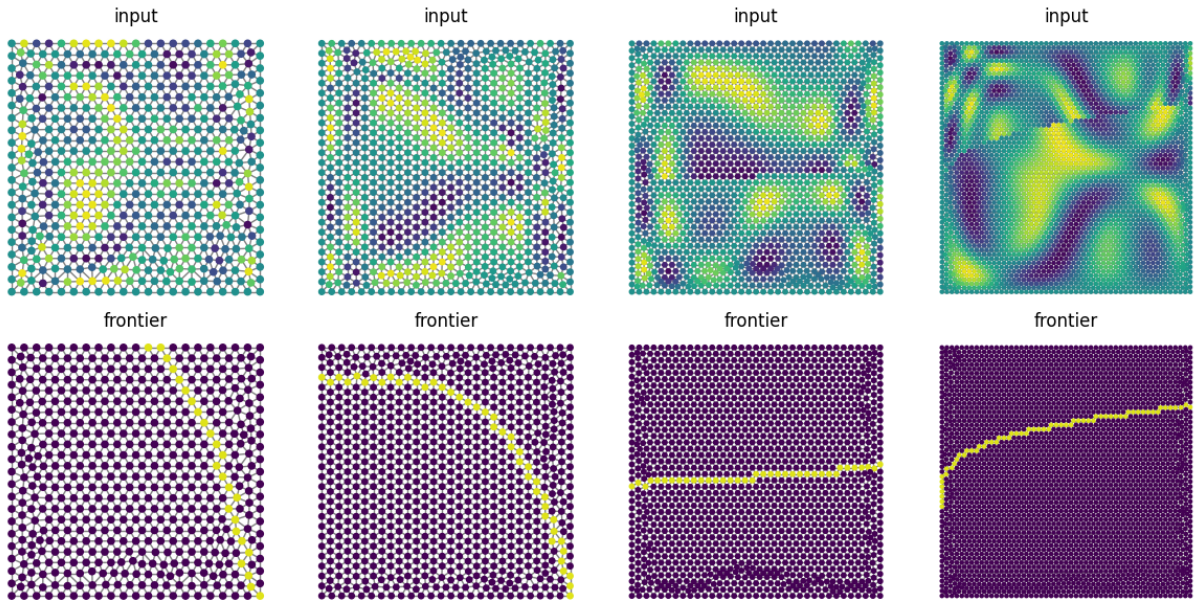

Figure 21: From left to right the meshes sizes are $h = 0.05$, $h = 0.04$, $h = 0.03$ and $h = 0.02$.

4.2 Models architecture

Our models will not use any irreversible reduction like mean or max reduction. We could have tried to reproduce the Vnet architecture, however the processes allowing us to subsample or

upscale a mesh and its node features are not implemented in Spektral.
Thus, our models will only be composed of convolutional layers put one after another. Moreover, the last convolutional layer will have a single output channel, and the sigmoid function as its activation.

4.3 Results on the trivial dataset

For this dataset the model used was composed of the following convolutional layers:

`GCN(32)-GCN(64)-GCN(128)-GCN(64)-GCN(32)-GCN(1).`

All the layers have a relu activation, except the last one having a sigmoid activation. The training of the model took about 5 minutes. The mesh used for the training is a regular mesh of the unit square and of characteristic length $h = 0.05$. Here are some examples of nodes features inputted to the model during training, and their respective expected output:

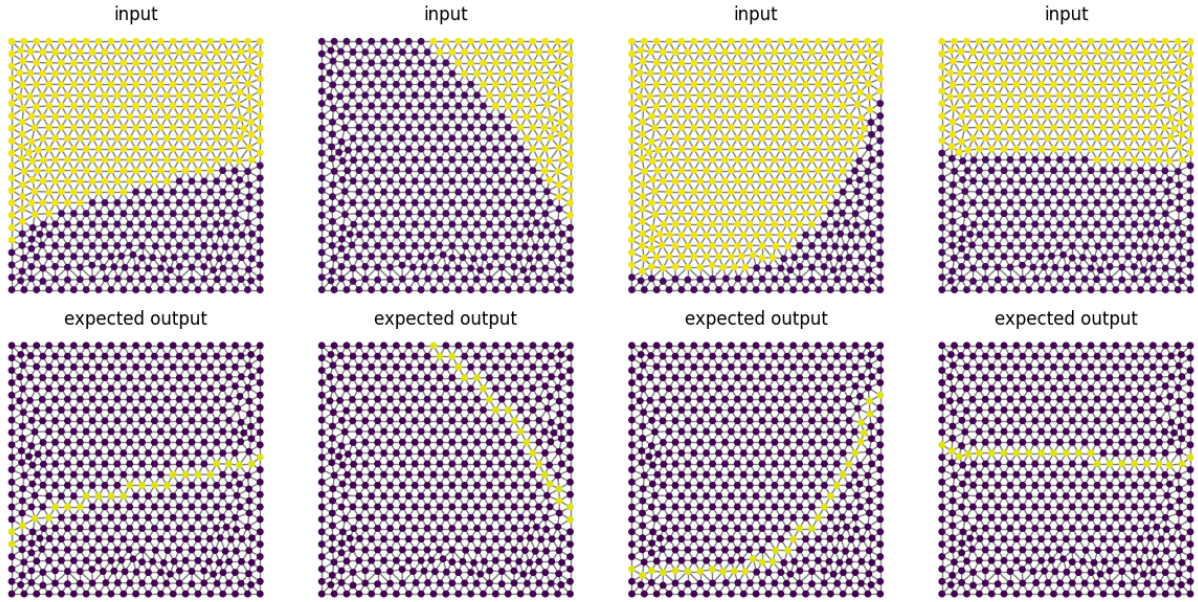

Figure 22: Node features inputted to the model and their expected output.

After training, the model is able to correctly predict where is situated the frontier on meshes of the same size. An example of such a prediction can be seen on Figure 23. Moreover the model was able to predict the frontier location on meshes different than the one used during training. For example the prediction on a mesh of characteristic size $h = 0.1$ is displayed on Figure 24.

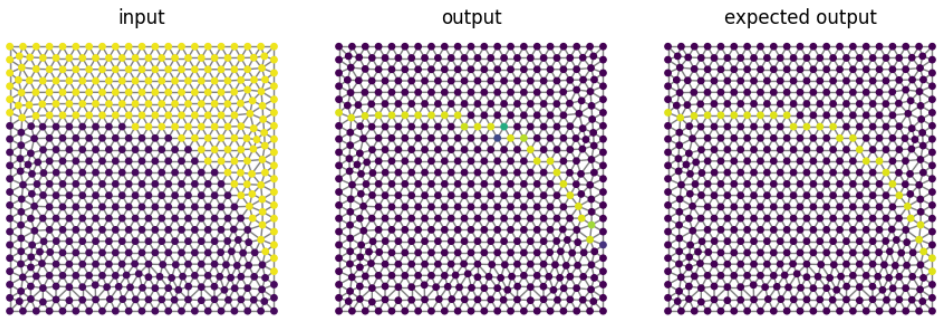

Figure 23: Comparison of the model output and the expected output on the mesh used during training.

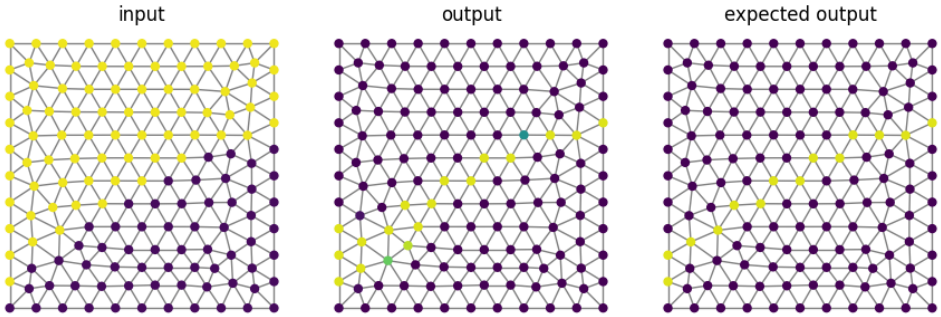

Figure 24: Comparison of the model output and the expected output on another mesh.

4.4 Results on the islands dataset

For this dataset the model used was composed of the following layers:

`GCN(32)-GCN(64)-GCN(128)-GCN(256)-GCN(128)-GCN(64)-GCN(32)-GCN(1).`

All the layers have a relu activation, except the last one having a sigmoid activation. The training of the model took 30 minutes. This time we did not get results as conclusive as in the trivial dataset. In fact the model struggles to detect the boundary between the two area. Some predictions made by the model can be seen on Figure 25.

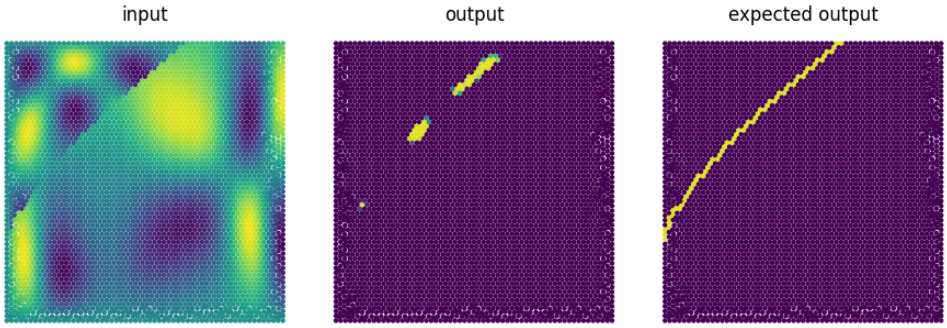

Figure 25: Prediction of the model for the islands dataset

As you can see, the model does not detect the frontier at all, but only detects large discontinuities in the image.
We concluded that the problem is to hard to solve for a model only composed of convolutional

layers. Some pooling like in the Vnet architecture might be necessary to achieve good results.

However we can try simplifying the problem by modifying the dataset. To do so we modified the islands dataset into another simpler dataset, named semi-trivial dataset. The images of the semi-trivial dataset are the same as in the islands dataset, to which we added a positive offset on a region of the images, and a negative offset to the region on the other side of the frontier. See Figure 26 for examples.

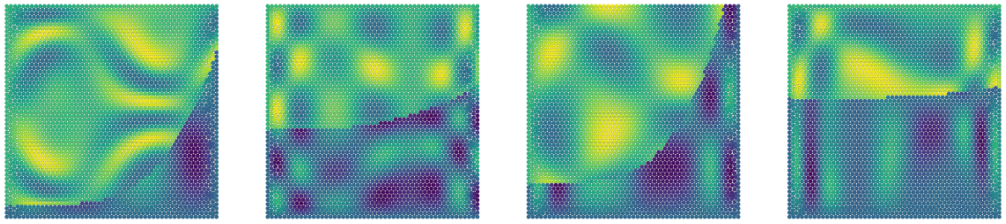

Figure 26: Example of meshes in the semi-trivial dataset.

This way the discontinuity at the frontier is bigger than in the islands dataset, and as a result it might be easier for our models to detect the frontier.
Sadly, our models did not achieve any better results than in the islands dataset. It confirms the fact that they are not complex enough to solve this simplified problem. A change in the models architecture is necessary.

4.5 Dynamic refining of the meshes

We saw previously that our model on the trivial dataset was able to detect frontiers in meshes of different resolutions. So in this section we will put this to our advantage and iteratively detect the frontier on mesh by refining it. This allows us to detect the position of the frontier more and more precisely as we refine our mesh.
To be able to achieve this is to have a function refining a mesh around the detected frontier. We denote by M the mesh on which we detected the frontier and M' the refined mesh. Then the nodes of M' are the nodes of M plus the barycenter of every triangle of M having a node on the frontier. Finally to reconstruct the triangles of M', we use a Delaunay triangulation. Here is an example of such a construction:

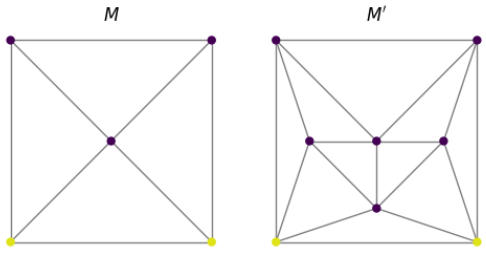

Figure 27: Example of mesh refining using Delaunay triangulation. The yellow nodes are on the frontier

We can now describe the process used to dynamically refine the mesh around the frontier:

1. Detect the frontier on a given mesh M.
2. Refine M into a more detailed mesh M' around the detected frontier.

3. Restart at step 1 using M' the base mesh for the detection.

We stop this process after 5 iterations. An example of this process is available on Figure 28. As you can see, the model is able to correctly predict the position of the frontier, even when the mesh structure is highly modified after multiple iterations.

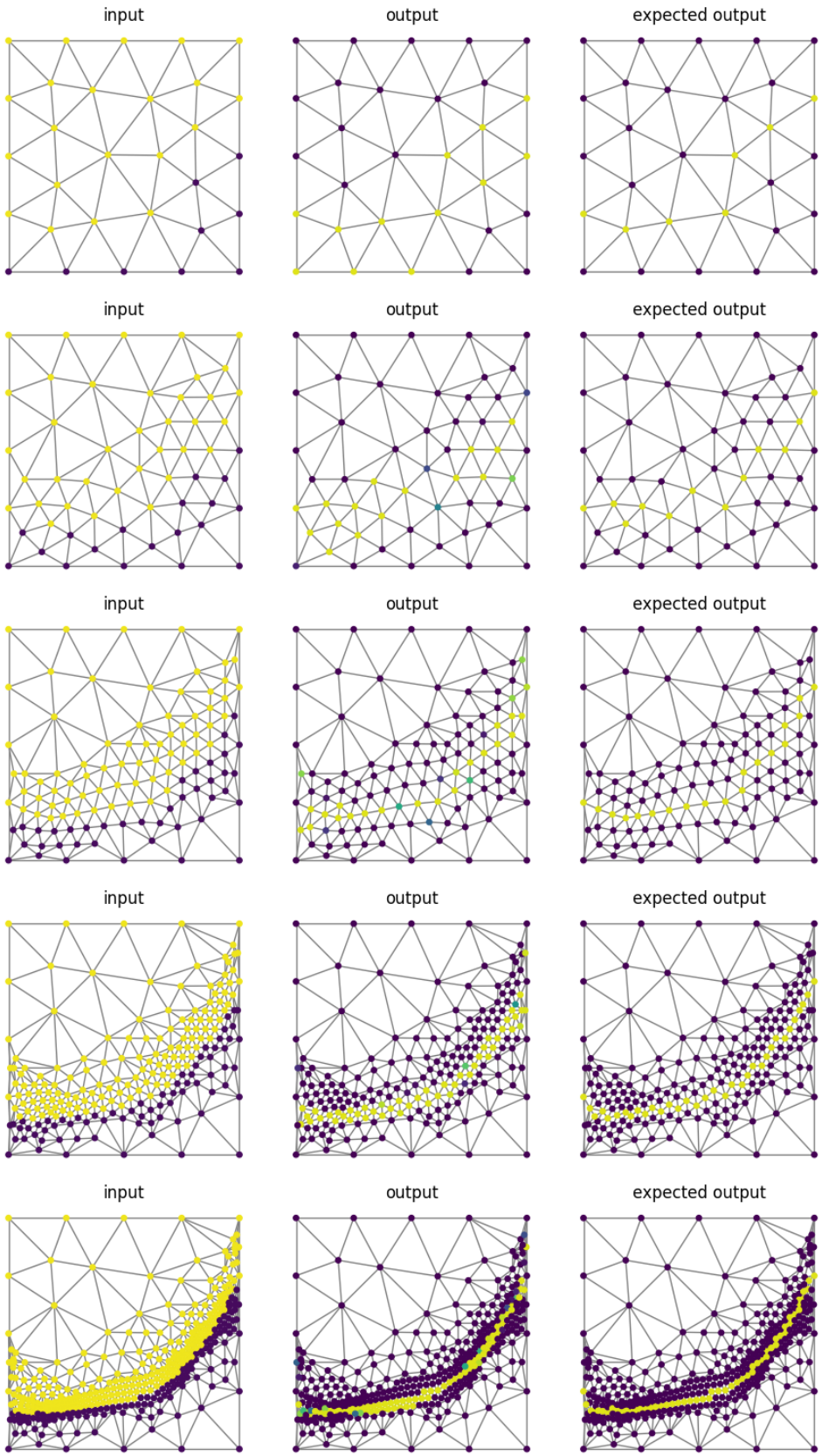

Figure 28: Iterations

5 Tools used and Implementations

The totality of this project was coded in Python 3.8. Moreover the following libraries were used:

- Tensorflow/Keras [1]: a deep-learning Python library. It allowed us to construct models and use the already implemented training and evaluation pipelines.
- Spektral [4]: an extension of Keras which implements the graphs neural network layers talked about before. It also helps us set up a pipeline for preprocessing the images into graphs.
- Github: used to share the code, open issues, and see our progression in the project.
- PyGMSH: a GMSH python interface allowing us to create various geometries and generate meshes directly in our Python pipelines.
- PyGSP 0.5.1 [2]: a graphical library helping us visualize graphs and signals.

There are multiple Python files on my repository, each having a specific usage:

- `mnist_training.py` and `frontier_training.py` are used to respectively train the models on the MNIST and Frontier datasets. The trained models are saved on the hard drive after training.
- `datasets.py` contains two classes inheriting the Spektral `Dataset` class. They permit us to extract node features from the datasets and construct graphs for our models to train on.
- `frontier_generator.py` contains the code necessary to generate the Frontier dataset images. It was written by V. Vigon.
- `mnist_evaluation.py` and `frontier_evaluation.py` are used to evaluate saved models on new data.
- `mnist_measures.py` allows us to make multiple measures on a model trained for the MNIST dataset.
- `frontier_refining.py` contains the code to dynamically refine a mesh using Delaunay triangulation and implements the refining process for the Frontier dataset.
- `plots.py` contains functions to plot results and data generated by our code. These functions where the ones used to generate the figures in this report.

6 Conclusion

We saw in this project that Graph Convolutional Networks can achieve good results even when the underlying graphs are modified by local refining or global deformations.
In fact we were able to construct a robust classification model on the MNIST dataset. Moreover, we made a model able to detect the frontier in the trivial dataset, regardless of the mesh considered.
This project will be followed by an internship. A part of this internship will be to solve a transport equation on a rough mesh, to detect discontinuities on this mesh with the help of GCNs, and refine the mesh where the discontinuity was located in order to improve the resolution of the transport equation. This project makes us confident that this will be achievable.
This project was limited in time. We could have gone more in-depth in some points:

- The current dynamic refining for the frontier dataset gives very badly structured meshes. We could have for example refined a triangle into 4 smaller ones while preserving the smallest angle. This way the refined mesh can still be used to solve equations.
- We saw that our model was not complex enough to detect frontiers in the islands dataset. With more time we could have implemented, or use an already implemented version of the Unet architecture but for graphs. We believe it would have given us convincing results.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Michaël Defferrard, Lionel Martin, Rodrigo Pena, and Nathanaël Perraudin. Pygsp: Graph signal processing in python. *URL https://github. com/epfl-lts2/pygsp*, 2017.

[3] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[4] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral. *arXiv preprint arXiv:2006.12138*, 2020.

[5] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.