University of Strasbourg

M1 in Applied mathematics and Computer science

Internship

Graph convolutional networks and some applications

Corentin MENGEL

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

Contents

1 Introduction

This internship is a continuation of the work done in a project earlier this year. During this project, we saw that Graph Convolutional Networks can achieve good results even when the underlying graphs are modified by local refinements or global deformations. We were in fact able to construct a robust classification model on the MNIST dataset. Moreover, we made a model able to detect the border between two areas in a signal, in simple cases.

In this internship, we focused on two problem. The first one is improving the Burgers' equation resolution by dynamic refining. To do so, the previous work done will be reused and improved.
The second problem is to define an interpolation operator on unstructured mesh in order to solve the linear transport equation by using the semi-Lagrangian method.

This internship was realized at the UFR de Mathématiques et Informatique, and was supervised by Emmanuel Franck, Laurent Navoret, Vincent Vigon and Laurène Hume. I would like to thank them for their guidance and availability throughout this internship.

I will present in this report the different results we achieved, and the possible improvements that could be performed.

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 $[n_i, n_j]$.

2.2 Convolutional ayers

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 $X_i^l \in \mathbb{R}^d$ denote the feature vector of the node n_i at the layer l. Then the updated features $X_i^{l+1} \in \mathbb{R}^{d'}$ 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].

2.3 Pooling Layers

The previous convolutional layers are layers that compute new node features, but never change the graph structure. However, we might want to consider the same graph but at different resolutions. To do so, pooling layers are needed. They can be used to enlarge receptive fields, thereby giving rise to better generalization and performance.
In this section I will present two graph pooling layers that I used during this internship.

2.3.1 Top-k pooling

This first layer is the Top-k pooling layer and was proposed in [3]. This layer selects a subset of nodes to form a smaller graph. To do so, a trainable vector $p \in \mathbb{R}^d$ is used. All the node features are projected in 1D using this vector. More precisely, if n_i is a node and $X_i \in \mathbb{R}^d$ is its node features, then the projection will be $y_i = X_i \cdot p/\|p\| \in \mathbb{R}$. This scalar measures how much information of the node n_i is retained when projected onto p. By down-sampling the graph, we wish to preserve as much node information as possible. This is achieved by selecting the nodes having the largest scalar projection on p to form the new graph.
The layer realizes the following operations:

$$\begin{aligned} y &= X \cdot p/\|p\|, \\ \text{idx} &= \text{rank}(y,\, k), \\ \tilde{y} &= \text{sigmoid}(y(\text{idx})), \\ \tilde{X} &= X(\text{idx},\, :), \\ A' &= A(\text{idx},\, \text{idx}), \\ X' &= \tilde{X} \odot (\tilde{y}\mathbf{1}_d^T), \end{aligned}$$

where k is the number of nodes in the resulting graph and $\text{rank}(y,\, k)$ returns the indices of the k largest values in y. The matrices $X' \in \mathbb{R}^{k \times d}$ and $A' \in \mathbb{R}^{k \times k}$ are the output of the pooling layer. The output of this layer only depends of the inputted node features X and the trainable vector p.
On Figure 1 we can observe example of the Top-k pooling layer applied to a regular mesh of the unit square. The node features of each node of the mesh are its position, so $d = 2$. The projection vector p used for this pooling was (1, 1).

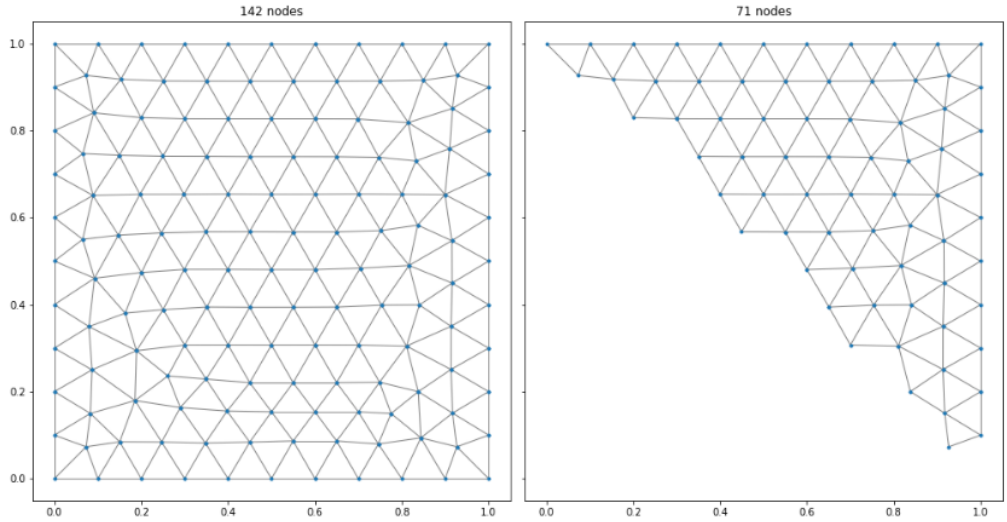

Figure 1: Initial mesh (left) and pooled mesh (right).

The initial mesh had 142 nodes, and the pooled mesh had 71 nodes, so we reduced its size by half. As we could have expected, only the nodes n_i of position $(x_i,\, y_i)$ verifying $x_i + y_i \geq 0.5$ were conserved in the pooled mesh.

2.3.2 k-Means pooling

During this internship, we also developed a second pooling layer. This time, the pooling results only depend on the nodes' positions in the mesh, and not the node features.
The k-Means pooling layer is based on the k-mean clustering algorithm. Its only parameter is k indicating how many nodes are parts of the pooled mesh. The pooling process is described as follow:

- we compute k clusters $(C_i)_{1 \leq i \leq k}$ of the inputted nodes using the k-means clustering algorithm,
- the k centers $(\tilde{n}_i)$ of those clusters are the nodes of the new mesh,
- we construct the adjacency matrix $A' \in \mathbb{R}^{k \times k}$ of the new mesh by using the Delaunay triangulation,
- we need to give node features to the centers $(\tilde{n}_i)$. To do so, each center $\tilde{n}_i$ will have the node feature

  $$X'_i = \text{gather}(X, C_i),$$

  where $\text{gather}(X, C_i)$ combines the node features of all the nodes (n_j) in C_i into a new node feature $X'_i \in \mathbb{R}^d$. For example X'_i can be the element wise maximum or average of all the node features of the nodes in the cluster C_i.

On Figure 2 we can observe a simple example of the k-Means pooling layer. The input mesh is a regular mesh of the unit square. The number of nodes is reduced to 1% compared to the inputted mesh. The input node features $X \in \mathbb{R}^{N \times 1}$ are simply $X_i = x_i + y_i$ where (x_i, y_i) are the positions of the node n_i.

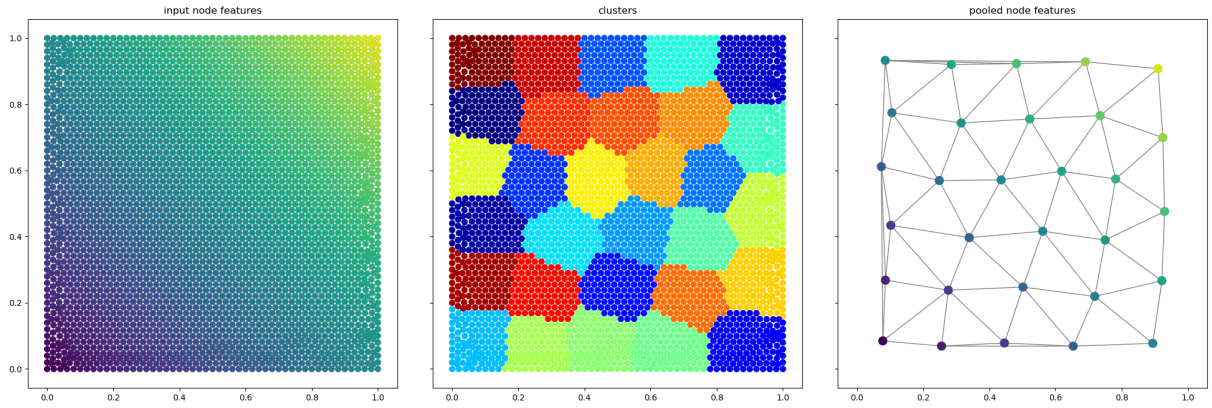

Figure 2: Initial mesh (left), clusters (center), and pooled mesh (right).

In this example, the node features of the cluster center is the maximum of all the node features of the nodes in the cluster.

2.3.3 k-Means unpooling

After reducing a mesh by applying a pooling layer, it can be interesting to up-sample this pooled mesh into the original mesh.
Let say that we have an original mesh with N nodes, and node features $X \in \mathbb{R}^{N \times d}$. After

applying a k-Means pooling layers, we obtain clusters $(C_i)_{1 \leq i \leq k}$ and new node features $X' \in \mathbb{R}^{k \times d}$. To reconstruct node features $X'' \in \mathbb{R}^{N \times d}$ for the original mesh, we simply define:

$$X''_j = X'_i \text{ for all } j \in C_i.$$

By doing so we obtain the following result:

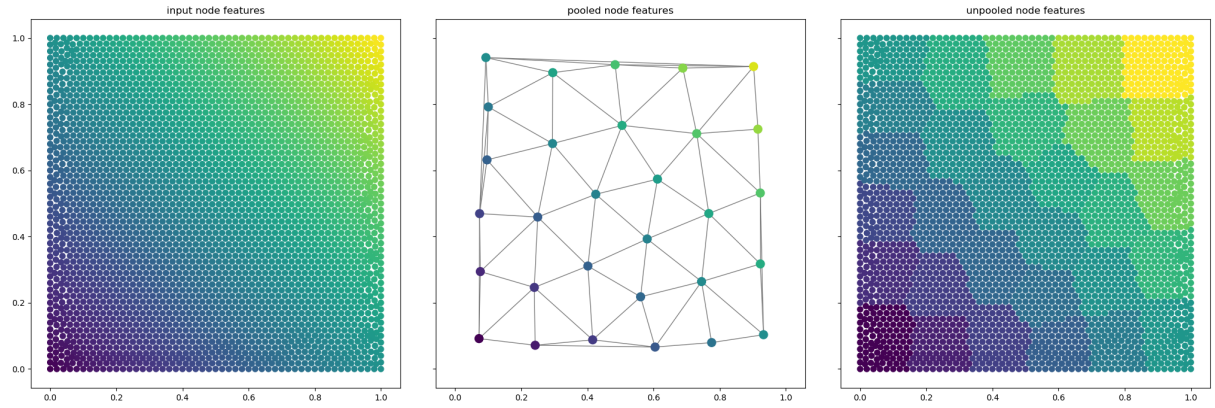

Figure 3: Initial node features (left), pooled node features (center), and unpooled node features (right).

On a less trivial example we obtain such results:

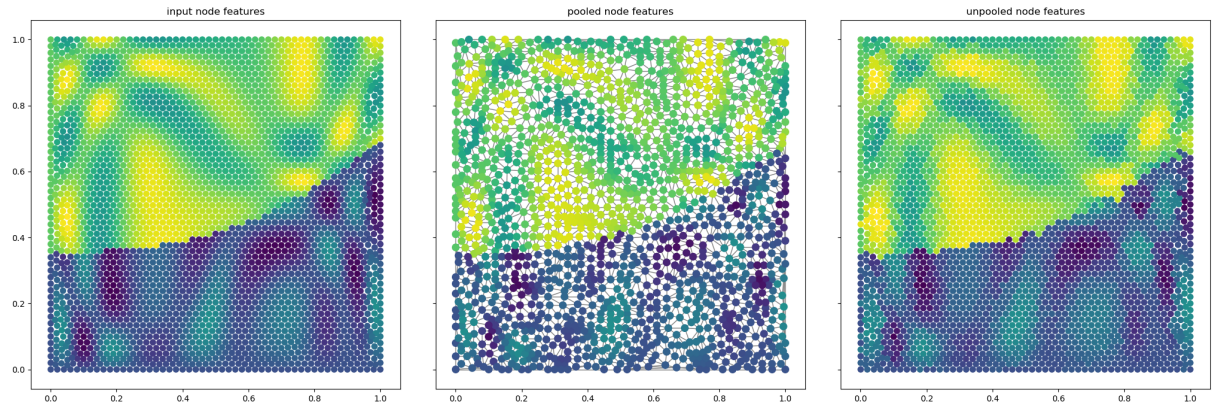

Figure 4: Initial node features (left), pooled node features (center), and unpooled node features (right).

Here the pooled mesh has 50% less nodes than the original mesh.

3 Frontier detection

3.1 Frontier dataset

During my previous project on graph convolutional networks and their applications, I studied a frontier detection problem. The problem consists of detecting the position of the frontier between two areas on a mesh.
There are 3 different types of areas, so the frontier dataset is subdivided in 3 parts:

- Island dataset: the two areas are two randomly generated oscillating signals.
- Semi-trivial dataset: same as the island dataset, but there is an offset between the two zones so that the frontier is more distinguishable.
- Trivial dataset: the two areas are different constant values.

Here are some examples from each dataset:

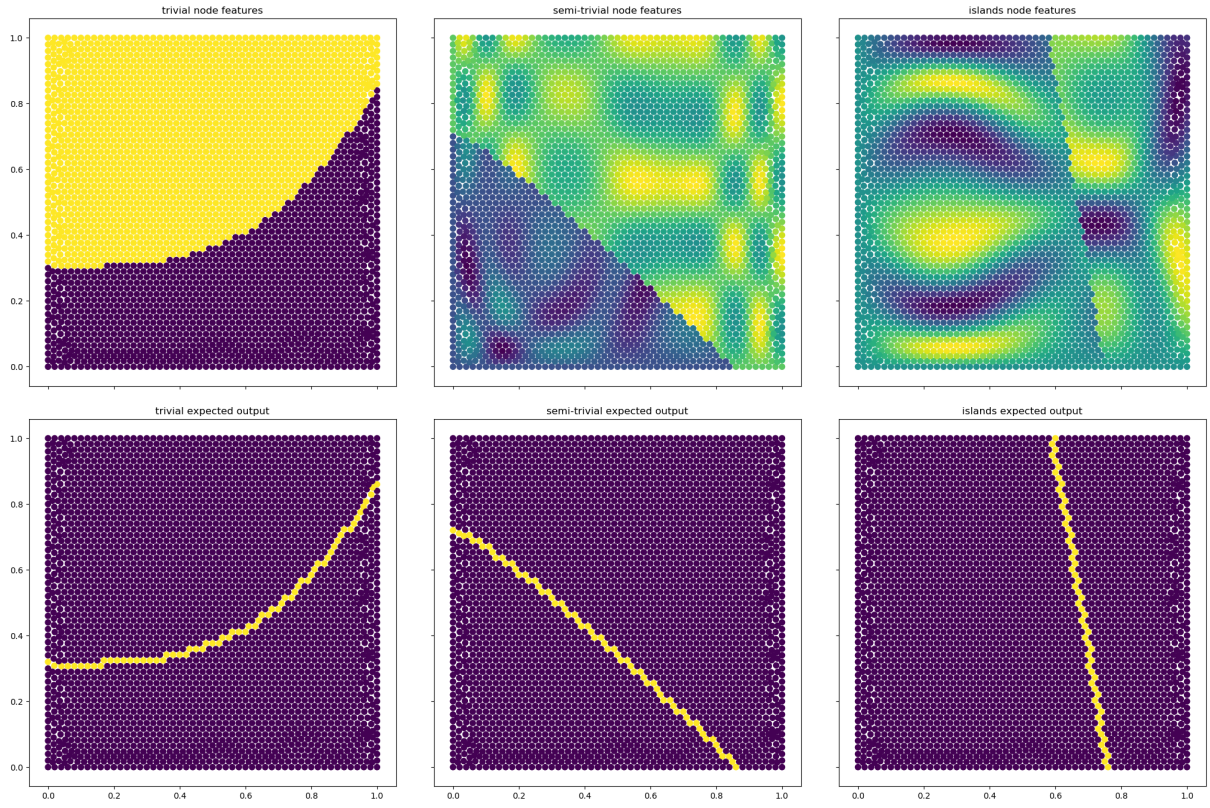

Figure 5: Input node features (top) and expected output (bottom).

3.2 Previous results

During my previous work, the network I used was simply a sequence of convolutional layers put one after the other. The mesh was not modified via pooling or unpooling.
Such a simple model was complex enough to solve the frontier detection on the trivial dataset.

However as we can see on Figure 6, the network struggled to detect the frontier on the islands and semi-trivial dataset.

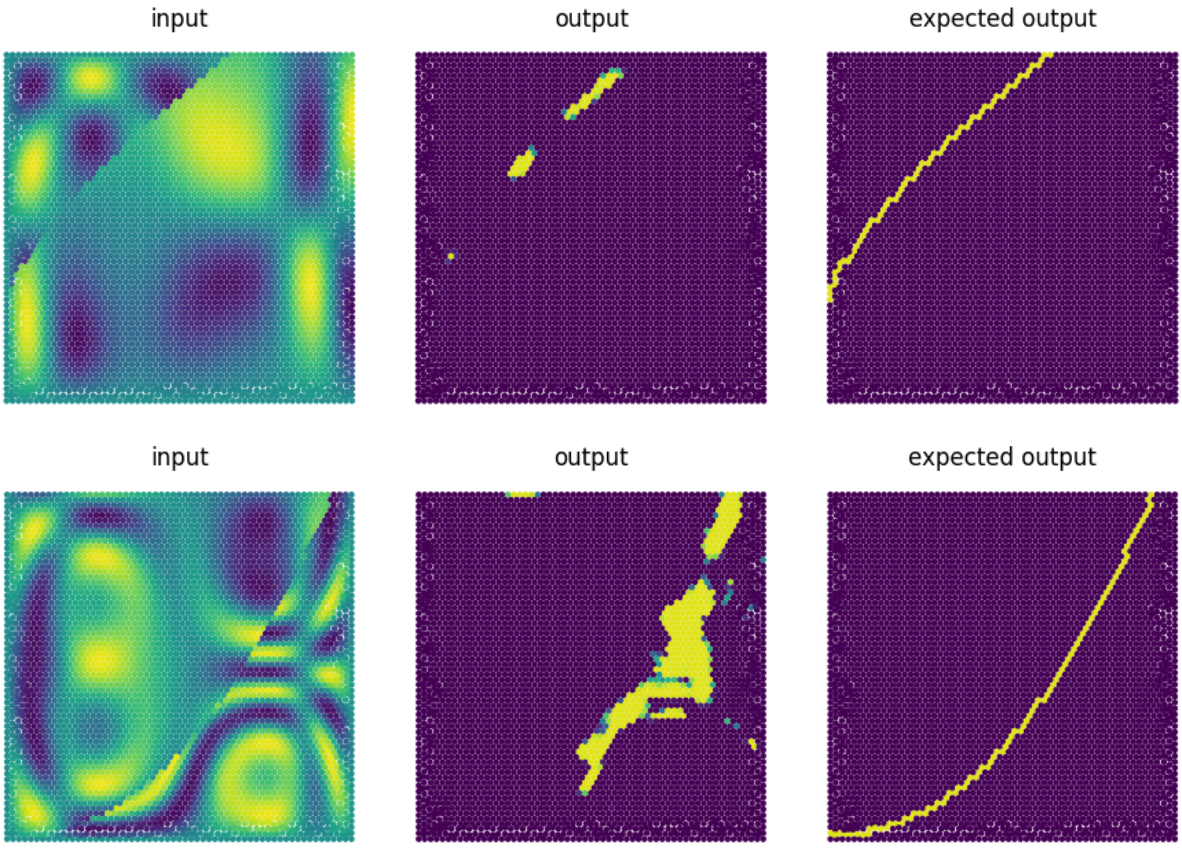

Figure 6: Old model results on the islands dataset.

The conclusion was that the model was not complex enough, and that a change in its architecture was necessary.

3.3 U-Net architecture

The U-Net architecture was first developed for biomedical image segmentation. The network consists of a contracting path and an expansive path, which gives it the u-shaped architecture.
The contracting path consists of convolutional layers followed by pooling layers. After each pooling, the number dimension of the node features is increased to compensate with the reduction of the mesh. The expansive path reconstructs the initial mesh using convolutional layers followed by unpooling layers. Moreover, the expansive path reuses previously computed node features during the contracting path via concatenation.
The models I used in this internship were U-Net networks with a depth of 3, i.e. 3 pooling layers and 3 unpooling layers. A simplified representation of the model can be seen on Figure 7.

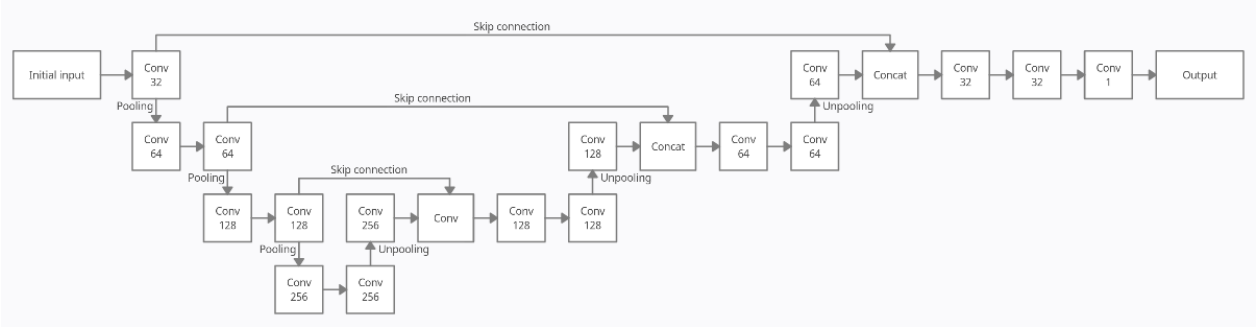

Figure 7: Architecture of the model used.

3.4 Results

For the U-Net architecture, we have the liberty of choosing what type of pooling and unpooling layers we will use. Same for the convolutional layer.
My first try was to use the Top-k pooling and unpooling layers, with the Vanilla GCN convolutional layer. The model was implemented using the KGCNN library [5]. However the model was not able to give satisfying results on the trivial dataset. In fact, as outlined in the paper [2], the Top-k pooling layer has a major issue. The nodes that are connected usually share similar node features, and their similarities further increase after the convolutional layers. As a result, those nodes will have a similar score and when selecting the top k nodes, entire portions of the mesh are discarded, making it hard to recover the information during the unpooling layers.
As a result I changed the pooling and unpooling layers from Top-k to k-Means. Each pooling layer reduced its inputted mesh by 75%. The model was then able to detect the frontier in the trivial case. Moreover, it was able to accurately detect the frontier in the semi-trivial case, which was a breakthrough from the previous sequential model:

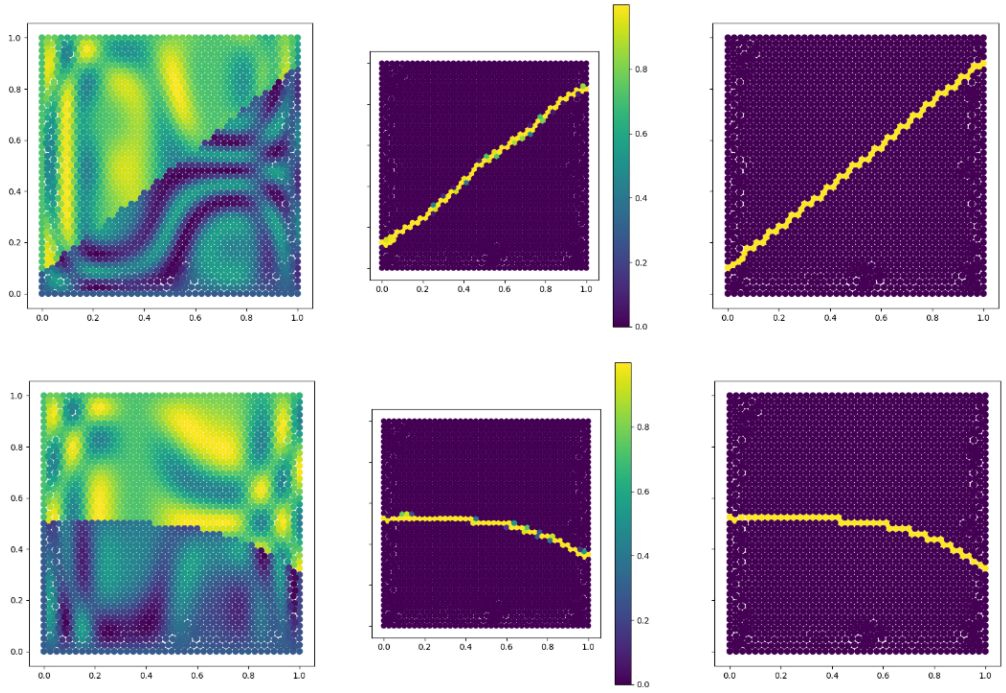

Figure 8: Input (left), model prediction (middle), expected output (right).

However, the model still was not able to detect the frontier in the islands dataset. Therefore, I changed the convolutional layer used in the U-Net architecture from the Vanilla GCN layer to the ChebConv layer, with the parameter $K = 3$. It resulted in the model having more trainable parameters (about 1 million) and being more complex. With this modification, the model was finally able to detect the frontier in the three cases, including the islands dataset:

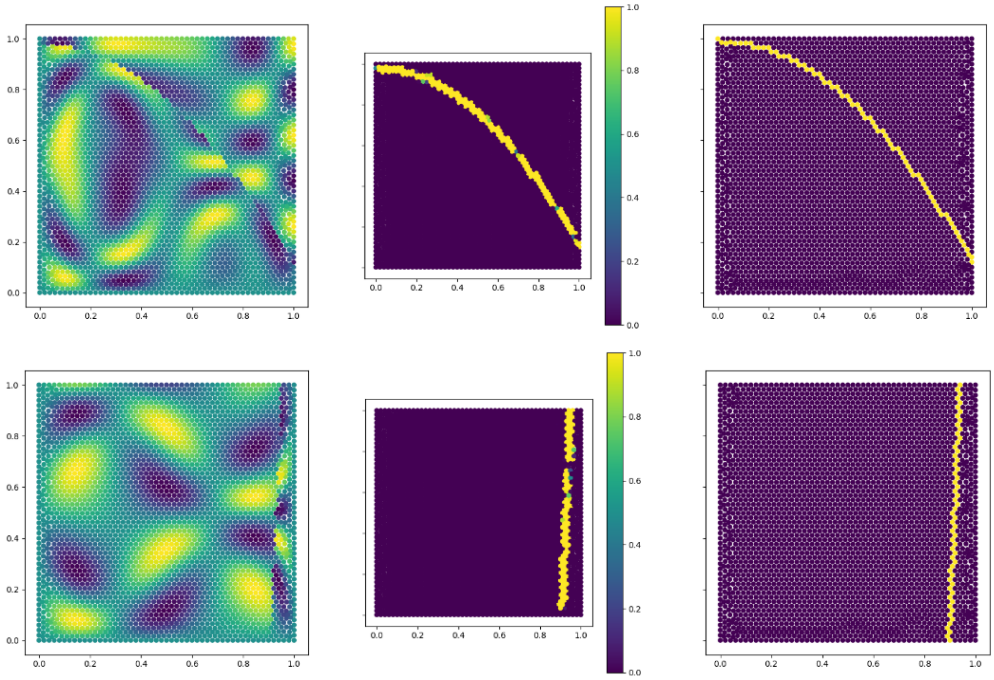

Figure 9: Input (left), model prediction (middle), expected output (right).

As we can see on Figure 9, the model is even able to locate the discontinuity when it is near the border of the mesh.
The model was trained 1500 signals, for a duration of about 2.5 hours. I used the v100 GPU made available to me by the university. The loss function used during training is the dice loss:

$$d(p,\, q) = 1 - 2\frac{\sum_i p_i q_i}{\sum_i (p_i + q_i)}.$$

The numerator is the sum of nodes being properly predicted as being on the frontier, and the denominator is the sum of total nodes of both prediction and ground truth. The model achieved a final loss of 0.29 on the test set.

4 Burgers' equation and dynamic refining

The Burgers' equation is a partial differential equation used in multiples fields such as fluid mechanics or traffic flow. In my internship I considered the 2-dimensional case of the Burgers' equation, which can be written as:

$$\partial_t \rho(t,\, x) + \nabla \cdot \left(a \frac{\rho(t,\, x)^2}{2} \right) = 0 \tag{1}$$

where $\rho : \mathbb{R}_+ \times \mathbb{R}^2 \to \mathbb{R}$, $a \in \mathbb{R}^2$ and $t \in [0,\, T]$.
To solve this equation numerically, multiple methods are available. For example, we can use the kinetic relaxation method or the finite volume method. I personally used the finite volume method as it was easily implementable with my current mesh data structure.

4.1 Finite volume method

The finite volume method allows us to transform partial differential equations into algebraic equations. To do so, the volume integrals that contain a divergence term are converted to surface integrals by using the divergence theorem.
Let us suppose we have a triangle mesh Ω composed of the triangles Ω_j, $j = 1, \ldots, M$, and let us denote by $(t_n), n = 1, \ldots, N$ a finite discretization of the interval $[0,\, T]$. By ρ_j^n we define:

$$\rho_j^n = \frac{1}{|\Omega_j|} \int_{\Omega_j} \rho(t_n,\, x)\, dx$$

where $|\Omega_j|$ is the area of the triangle Ω_j. Moreover, let E_j be the set of indices k such that the triangle Ω_k has a common edge with the triangle Ω_j. Let k be in E_j. By d_{jk} we denote the length of the common edge between the triangles Ω_j and Ω_k, and by n_{jk} we denote the outward normal of the edge. See Figure 10:

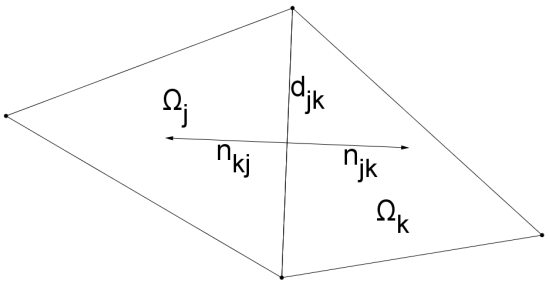

Figure 10: Notations used for the finite volume method.

Then, using the divergence theorem, *Equation* (1) can be rewritten:

$$|\Omega_j| \frac{\rho_j^{n+1} - \rho_j^n}{\Delta t} + \sum_{k \in E_j} d_{jk} F(\rho_j,\, \rho_k) = 0,$$

where:

$$F(\rho_j^n,\, \rho_k^n) = \frac{1}{2} \left[a \cdot n_{jk} \left(\rho_j^{n2} + \rho_j^{n2} \right) + \max \left(|a \cdot n_{jk} \rho_j^n|,\, |a \cdot n_{jk} \rho_k^n| \right) (\rho_j^n - \rho_k^n) \right].$$

This gives us the relation:

$$\rho_j^{n+1} = \rho_j^n - \frac{\Delta t}{|\Omega_j|} \sum_{k \in E_j} d_{jk} F(\rho_j^n,\, \rho_k^n).$$

For the method to converge, we must impose a CFL condition on the time step Δt. In fact, at every time t_n, the time step must verify:

$$\Delta t < \mathrm{CFL} \cdot \frac{\|\Omega\|}{\|a\| \|\rho^n\|_\infty},$$

where $\|\Omega\|$ is the length of the smallest edge in the mesh Ω, and CFL is a small constant strictly smaller than 1. In my programs, CFL = 0.08.

4.2 Boundary conditions

In general, the mesh's triangles have 3 neighboring triangles, except for the triangles on the boundary of the mesh. Those triangles need to be treated separately depending on the value of ρ^n. Let us suppose that Ω_j is a triangle on the boundary of the mesh Ω, i.e. has only 2 and not 3 neighboring triangles. Let n_j be the outward normal of the edge of Ω_j not having any neighboring triangles, and d_j its length.

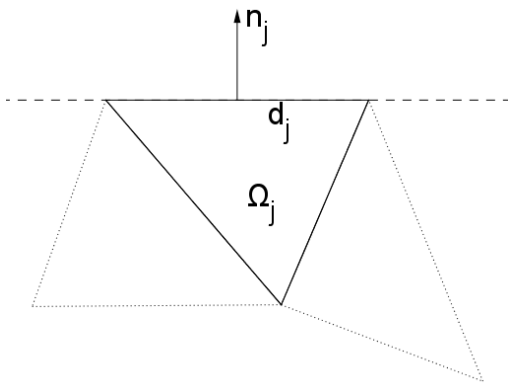

Figure 11: Notations used for the finite volume method on the boundary triangles.

We need to consider two cases:

- If $a \cdot n\rho_j^n < 0$, which means that the quantity ρ_j^n is entering the mesh, we impose $\rho_j^{n+1} = \rho_j^n$.
- If $a \cdot n\rho_j^n \geq 0$, which means that the quantity ρ_j^n is leaving the mesh, we subtract

  $$\frac{\Delta t}{|\Omega_j|} d_j F(\rho_j^n,\, \rho_j^n)$$

  to ρ_j^{n+1}. It is equivalent to adding a ghost triangle outside of the mesh with the value ρ_j such that Ω_j has 3 neighboring triangles.

4.3 Examples of solutions

In the following section the mesh Ω will be a triangle mesh of the unit square. Let us look at a first example, where ρ^0 is equal to:

$$\rho(0,\, x,\, y) = \sin(5\pi x) \sin(5\pi y),$$

and $a = (1, 0)$. Using the method described in the previous sections, we obtain the following results at $t = 0.05s$:

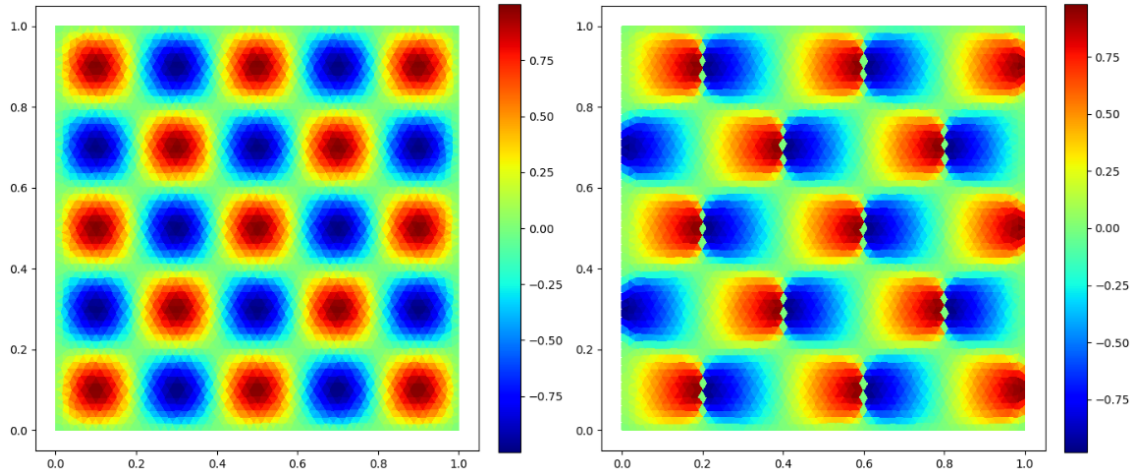

Figure 12: Initial solution (left) and final solution (right) at $t = 0.05s$

As we can see on Figure 12 the positive parts of the sine wave moved in the direction given by a, and the negative parts of the sine wave moved in the opposite direction. Moreover, we can observe oscillations at the intersection between the positive and negative parts of the sin wave at $t = 0.05s$.
Let us observe another example, this time using $a = (1, 1)$, and where the initial solution is more random. Here are the initial solution and the final solution:

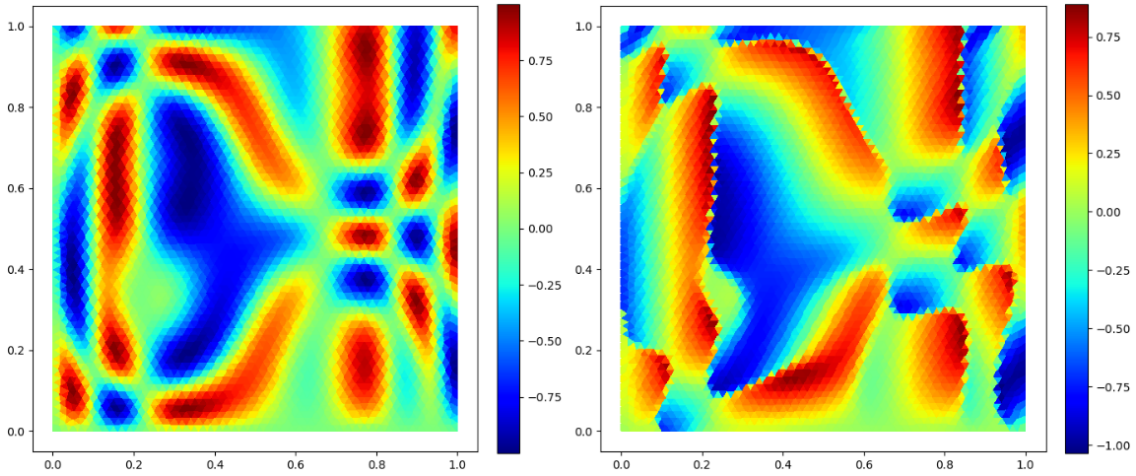

Figure 13: Initial solution (left) and final solution (right) at $t = 0.05s$

The same observations can be made on Figure 13: discontinuities appeared at the intersection of the positive and negative parts of the solution. As a result the solution computed has oscillations at those positions, and is not as precise as we would like.

4.4 Dynamic refining

To try and solve this problem, we can use a much finer mesh, i.e. a mesh with more triangles. However this solution is inefficient because the discontinuities appear only at precise

small parts of the mesh. Another solution is to refine the mesh while we are computing the final solution. The refining will occur when we are observing discontinuities in the current solution. By doing so we can hope that the final solution will be more accurate.
To detect the discontinuities, I used the model trained in Section 3. But in order to use the model, we need to transform the Burgers' solutions into node features. Indeed, currently the Burgers' solutions consist of an array of values for each triangle of the mesh. To get node features, I simply give to a node n_i the average value of every triangle n_i is a part of.
On Figure 14 I display different predictions made by the model on Burgers' solutions. The first prediction is made on a solution similar to the training dataset of the model. As we can see, the model correctly predicts the position of the discontinuities.
However we can ask ourself how good the model will be able to generalize on solutions significantly different than the ones in its training set. The second and third predictions on Figure 14 show that the model is still able to accurately detect the discontinuities, which is reassuring.
Another question that we can ask ourself is how the model behaves when given input solutions without any discontinuities. In fact, the model was only trained on signals containing discontinuities, so some unexpected behaviors can be expected. To be sure, I evaluated the model on initial Burgers' solution which are continuous. The results can be seen on Figure 15. As we can observe, the model does not detect any discontinuities. Once again this is reassuring, and indicates that this model will be usable to dynamically refine our mesh during the calculation of Burgers' solutions.
Now let us describe the pipeline used for dynamic refining. I suppose that we have a starting mesh Ω and an initial solution ρ^0. Then:
While $t < T$**:**

- update Δt and compute the solution for this time step on the mesh Ω,
- transform the triangles features of the solution to node features via averaging,
- use the model to get a prediction on the discontinuities positions if they exist,
- if discontinuities are detected:
  - replace Ω by refining it around those discontinuities. To do so, we refine the triangles around the nodes detected as being on discontinuities by subdividing them into 3 sub-triangles.
  - update Δt and compute the solution a second time on the new mesh.

However, a problem emerges from the CFL conditions. In fact, every time a mesh Ω is refined into a mesh Ω', we have $\|\Omega'\| \simeq \|\Omega\|/2$. As a result, if we refine the mesh every time the model detects discontinuities, the time step Δt will tend to 0, and the calculations of the Burgers' solution will take an enormous amount of time. To solve this problem I fixed a maximal amount of refining. Moreover, the mesh will be refined only after discontinuities have been detected for a certain amount of time. In practice, the maximum amount of refining is fixed to 4 and the time to wait between refinements is set to 0.008 seconds.

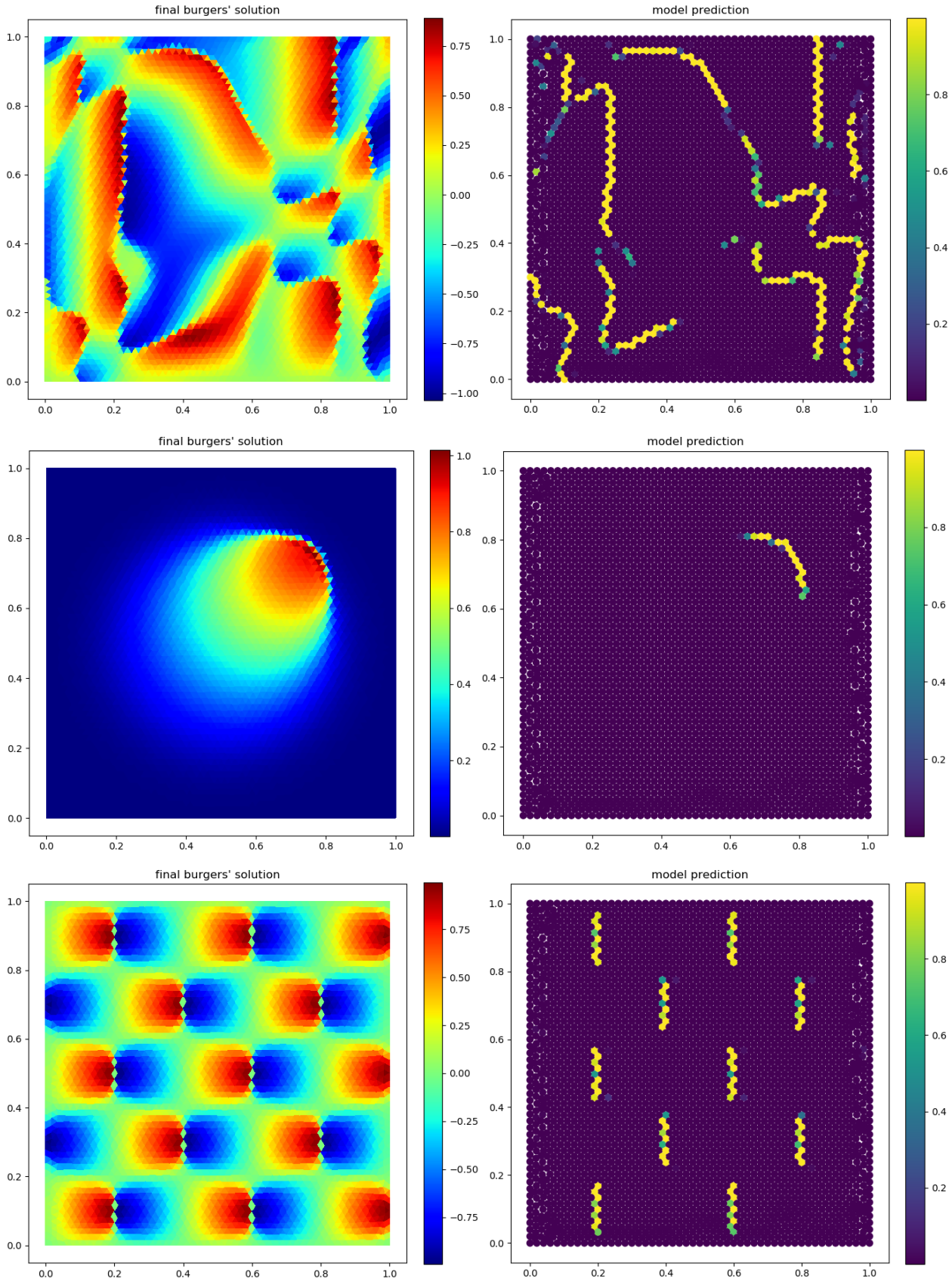

Figure 14: Final Burgers' solutions (left), and model predictions (right).

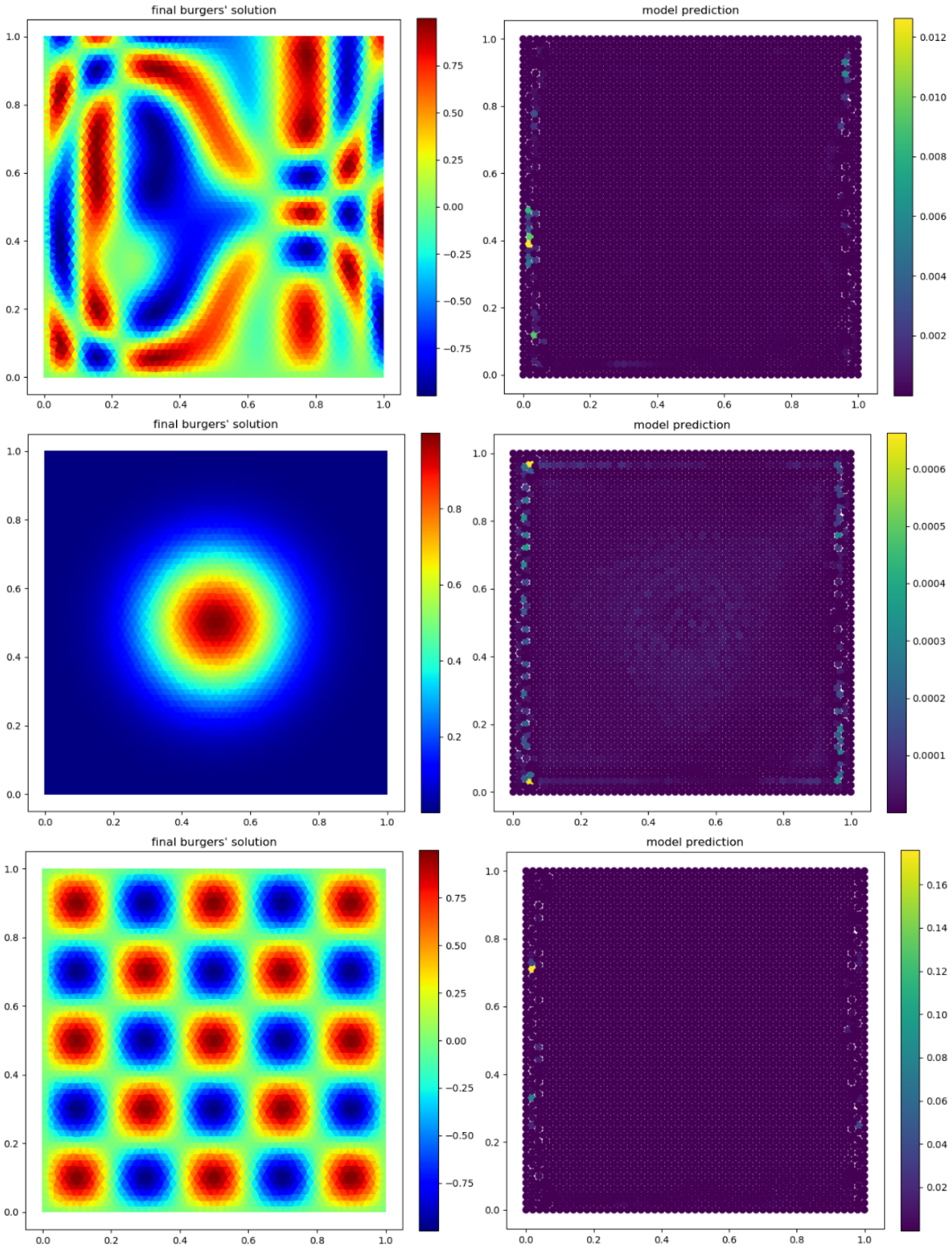

Figure 15: Initial Burgers' solutions (left), and model predictions (right).

4.5 Results

In this section I will compare the final Burgers' solution obtained with or without dynamically refining the mesh. We denote ρ as the solution computed without refinements and ρ_{dyn} the one compute with refinements.
To be able to numerically compare the two solutions, I need to have a solution that I deem being as accurate as possible. This third solution will be obtained by computing the Burgers' solution a third time, but on a much finer mesh Ω_{ref}. We denote this solution by ρ_{ref}. This way, the discontinuities observed previously will be less pronounced on ρ_{ref}.
Once this solution ρ_{ref} is computed, I will project the two other solution ρ and ρ_{dyn} on the finer mesh. These projections will be computed using the `griddata` function from the SciPy library [6].
Now that I have all three solutions on the same mesh, I can compute the L^1 error:

$$\|\rho_{\text{ref}} - \rho_{\text{dyn}}\|_{L^1} = \sum_j |\,(\Omega_{\text{ref}})_j\,|\,|(\rho_{\text{ref}})_j - (\rho_{\text{dyn}})_j|,$$

and the same error for ρ. This will tell us if the dynamic refining of the mesh allowed us to achieve better results.
The initial solution used is the one presented on Figure 13, and $a = (1, 1)$. Here is the solution ρ obtained at $t = 0.04s$ without any refinements:

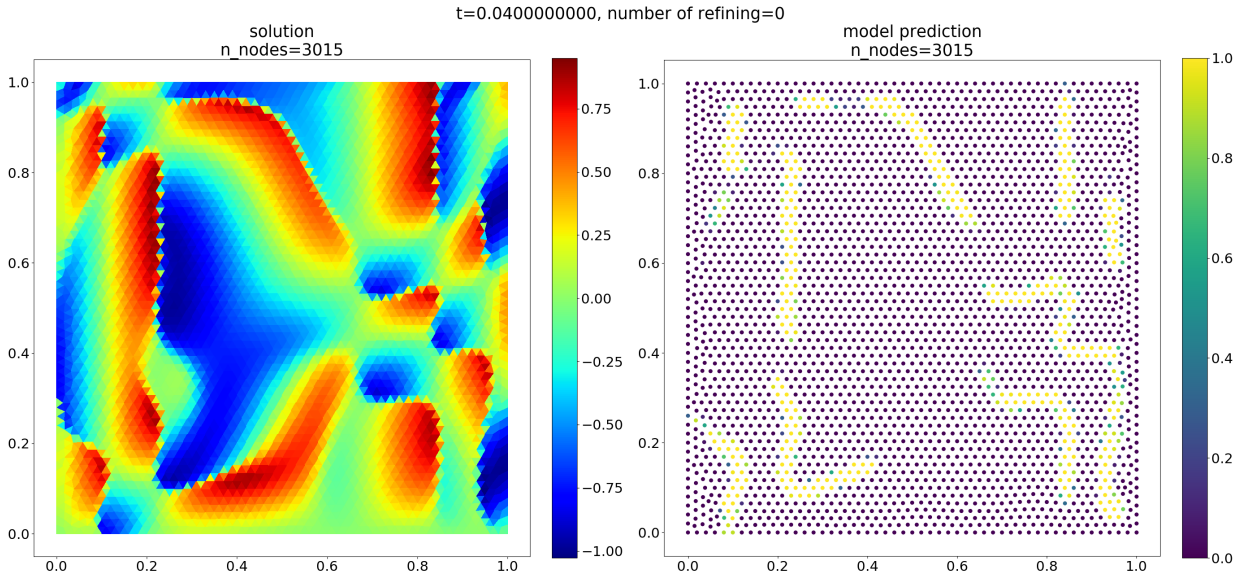

Figure 16: ρ (left), and the model prediction (right).

The solution ρ_{dyn} obtained with 4 refinements can be observed on Figure 17. We can already remark that the oscillations are less present on ρ_{dyn}. Indeed, on the right part of the figure we can see that the mesh has been refined at the discontinuities locations as desired. The solution computed on a much finer mesh, ρ_{ref}, is shown on Figure 18.
The two projected solutions are displayed on Figure 19 along with the respective errors relative to ρ_{ref}. As expected, the errors are smaller in the case of the dynamically refined solution ρ_{dyn}.

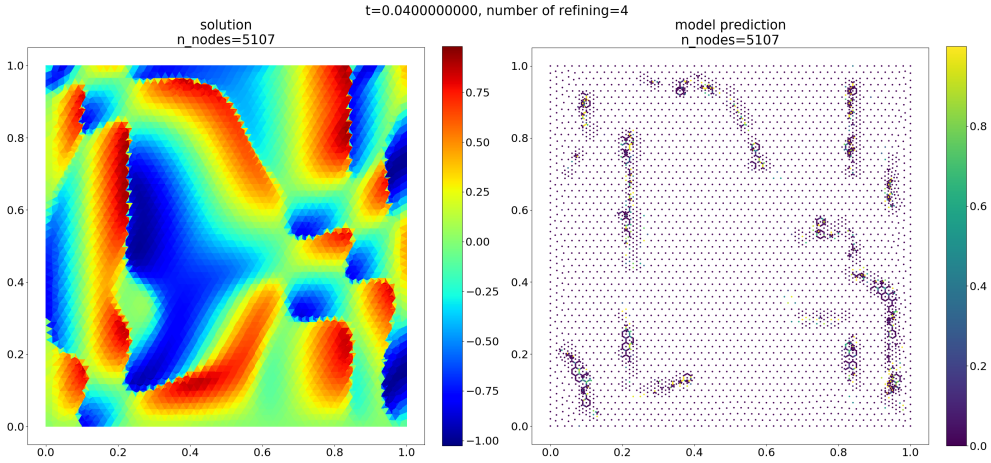

Figure 17: ρ_{dyn} (left), and the model prediction (right).

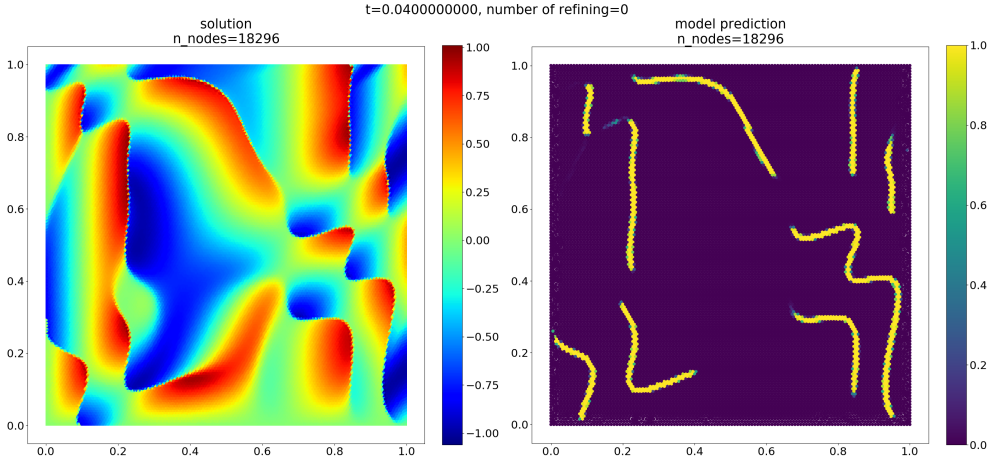

Figure 18: ρ_{ref} (left), and the model prediction (right).

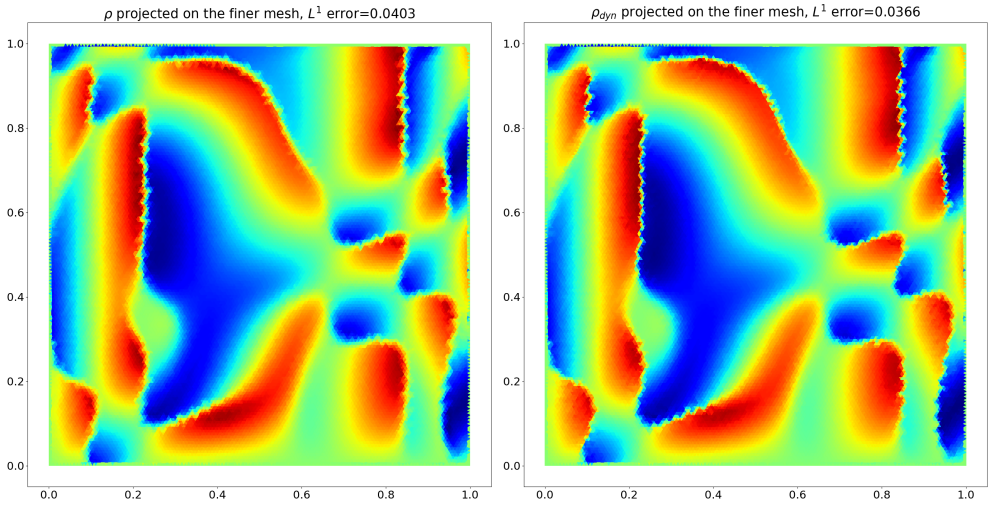

Figure 19: Projections of ρ (left) and of ρ_{dyn} on the finer mesh.

5 Transport equation and interpolation problem

A transport equation is an equation describing the displacement of some quantity. More precisely it can be written as:

$$\partial_t u + a(x) \cdot \nabla_x u = 0 \tag{2}$$

where $u : \mathbb{R}_+ \times \mathbb{R}^2 \to \mathbb{R}$ is the quantity and $a : \mathbb{R}^2 \to \mathbb{R}^2$ is the direction.

5.1 Semi-Lagrangian method

We can compute a solution to Equation (2) by introducing characteristic curves. A characteristic curve, denoted by $X_{s,y} : \mathbb{R}_+ \to \mathbb{R}^2$, is defined as the solution of:

$$\begin{cases} X'(t) &= a(t,\, X(t)), \\ X(s) &= y. \end{cases}$$

Suppose we have a discretization of $t_n = n\Delta t$, $n = 1 \dots M$ of the interval $[0,\, T]$, where $\Delta t > 0$ and $t_M = T$. Moreover, let Ω denote a triangle mesh of the unit square, with its nodes denoted by $(x_j)_{j=1\dots N}$. The solution u on Ω can then be computed as:

$$u_j^{n+1} = u(t_{n+1},\, x_j) = u(t_n,\, X_{t_{n+1},x_j}(t_n)).$$

However, the point $X_{t_{n+1},x_j}(t_n) \in \mathbb{R}^2$ is not necessarily a node of the mesh Ω. The semi-Lagrangian scheme consists of approaching $u(t^n,\, X_{t_{n+1},x_j}(t_n))$ by:

$$u(t^n,\, X_{t_{n+1},x_j}(t_n)) = (\Pi u^n)\left(X_{t_{n+1},x_j}(t_n)\right),$$

where Π is an interpolation operator.
In my internship, I only considered a simplified form of Equation (2) where a is a constant in $\mathbb{R}^2$. This way, the characteristic curves are explicitly known:

$$X_{s,y}(t) = y + (t - s)a.$$

As a result the semi-Lagrangian scheme can be written as:

$$u_j^{n+1} = (\Pi u^n)\,(x_j - a\Delta t)\,.$$

5.2 Interpolation problem

As seen in the previous section, we need to have an interpolation operator in order to solve the transport equation. In fact, let Ω^T be the mesh Ω but translated, i.e. the mesh whose nodes are $(x_j - a\Delta t)_{j=1\dots N}$. We can reconstruct a new mesh $\tilde{\Omega}$ of the unit square by joining the two meshes Ω and Ω^T. The nodes of the mesh $\tilde{\Omega}$ are the nodes of Ω and Ω^T who are in the unit square, and the triangles are computed using Delaunay triangulation. The initial solutions I considered are functions with compact support, so the nodes of Ω^T which are outside the unit square will have a fixed value equal to zero. An example of the mesh $\tilde{\Omega}$ can be seen on Figure 20.
The yellow nodes are the nodes of Ω while the purple nodes are the nodes of Ω^T. The interpolation operator Π will need to be able to interpolate values from the yellow nodes

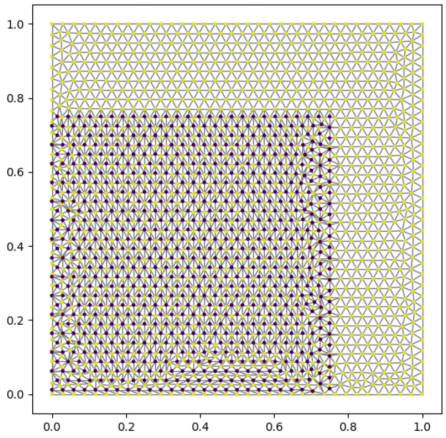

Figure 20: Mesh $\tilde{\Omega}$ on which we will apply the interpolation operator Π.

to the purple nodes. To do so, I trained a graph convolutional network having the U-Net architecture described on Figure 7. In fact, the U-Net architecture is adapted to interpolation problems. The dataset used for training the dataset consists of a modified version of the frontier dataset. The input signals to our model are frontier signals to which I have fixed 50% of the node features to zero. The expected output is the signals but without any node features missing. Some examples of signals in the modified dataset are available on Figure 21

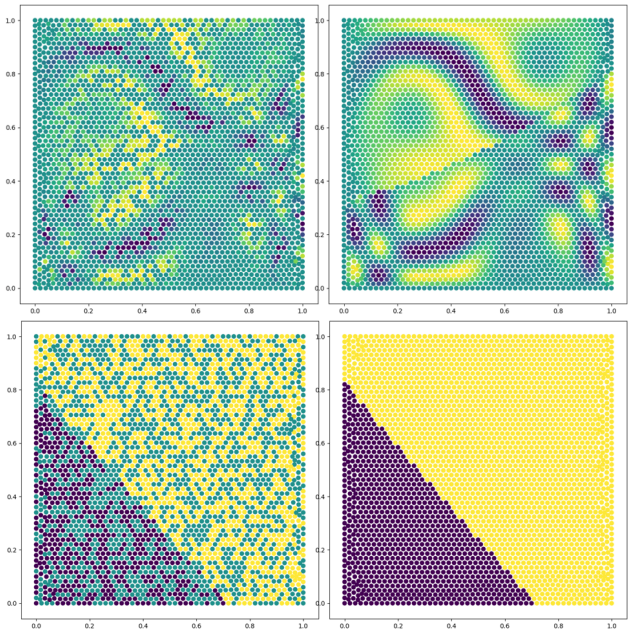

Figure 21: Input (left) and expected output (right).

After training the model, I compared it to the interpolation operator `griddata` from the library SciPy [6]. The comparison can be found on Figure 22.
As we can see, the U-Net and the `griddata` interpolation operators give similar results.

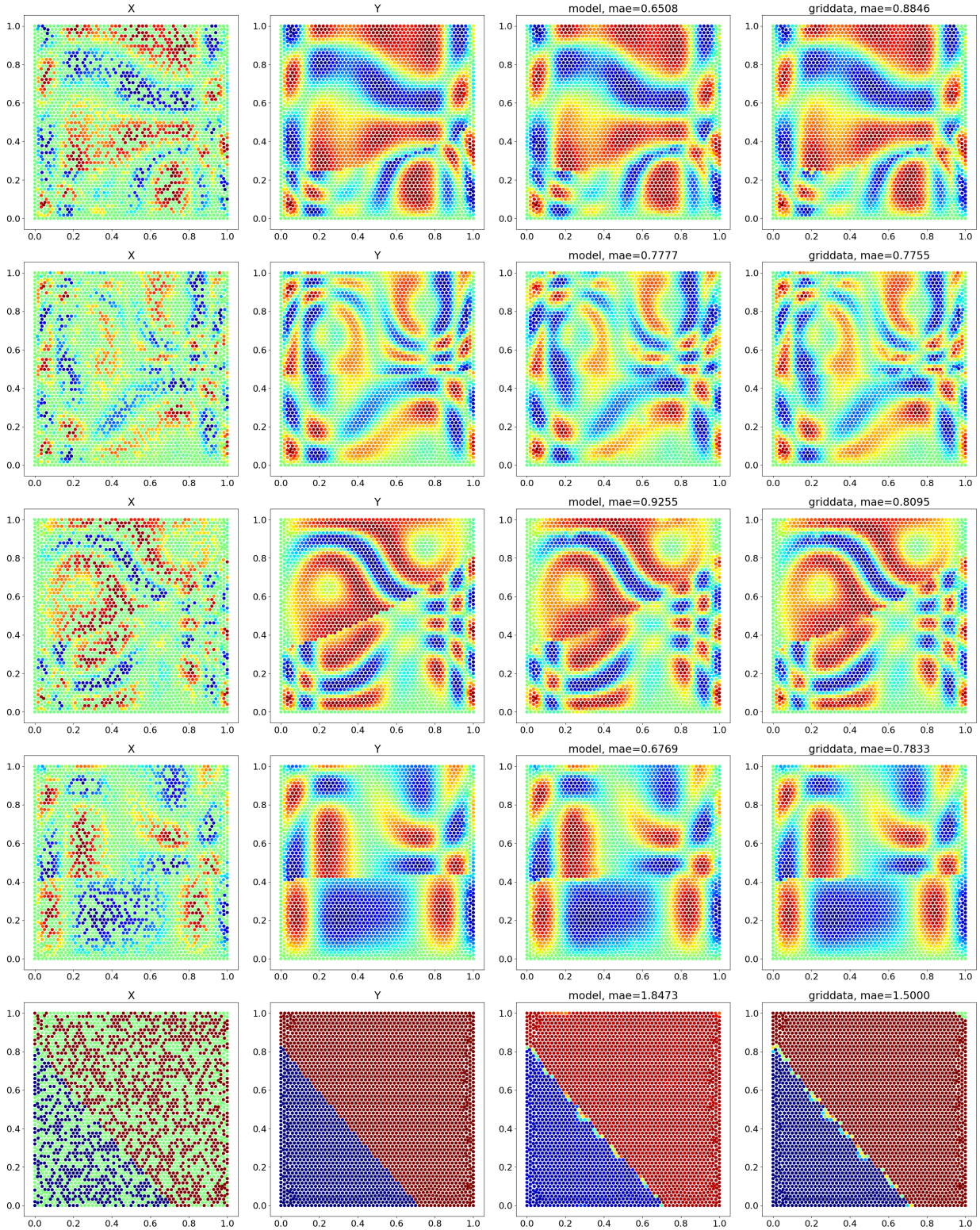

Figure 22: Input (left) and expected output (right).

5.3 Results

Now that we have our interpolation operator, we can solve the transport equation. In all our examples, we have $a = (1, 1)$. The first example, seen on Figure 23, is the displacement of a square with sharp edges. I plotted different solutions, each computed using different Δt. The smaller Δt is, the more interpolation steps are done. As we can see, the more interpolation steps are computed, the less precise our solution becomes, and the smoother the edges are. To be able to compare those solutions, I recomputed them using the `griddata` interpolation operator from Scipy. Those new solutions are displayed on Figure 24. As we can see, those solution are a bit more precise, but the square edges are still smoothed.
In the next example, I used a Gaussian function as initial solution. The results using the U-Net interpolation model can be observed on Figure 25. We can notice the same phenomenon than in the first example: the more interpolation steps are done, the less precise our solution is. As a result, the last solution is heavily degraded. On the other side, the solutions computed using the `griddata` interpolation operator shown on Figure 26 are precise and do not show any degradation after multiple interpolation steps.
Another observation is that for example on Figure 23 or Figure 25, the model outputs values smaller or larger than the ones on its input, indicating that the model is instable.
The differences observed between those two operators could be explained by the training dataset of the U-Net model. In fact, during training the U-Net model is only trained on regular triangle meshes. However, as it can be seen on Figure 20, the mesh on which the interpolations steps are done is very different than the one used during training. As a result, the model does not behave as good as expected. To improve the interpolation, I could have trained the model on different meshes similar than the one on Figure 20.
A second possible improvement to the model could be made by modifying the way it is trained. Indeed, we could train compositions of the model and not the model itself. This way the training would force the model to be stable by multiple successive composition.

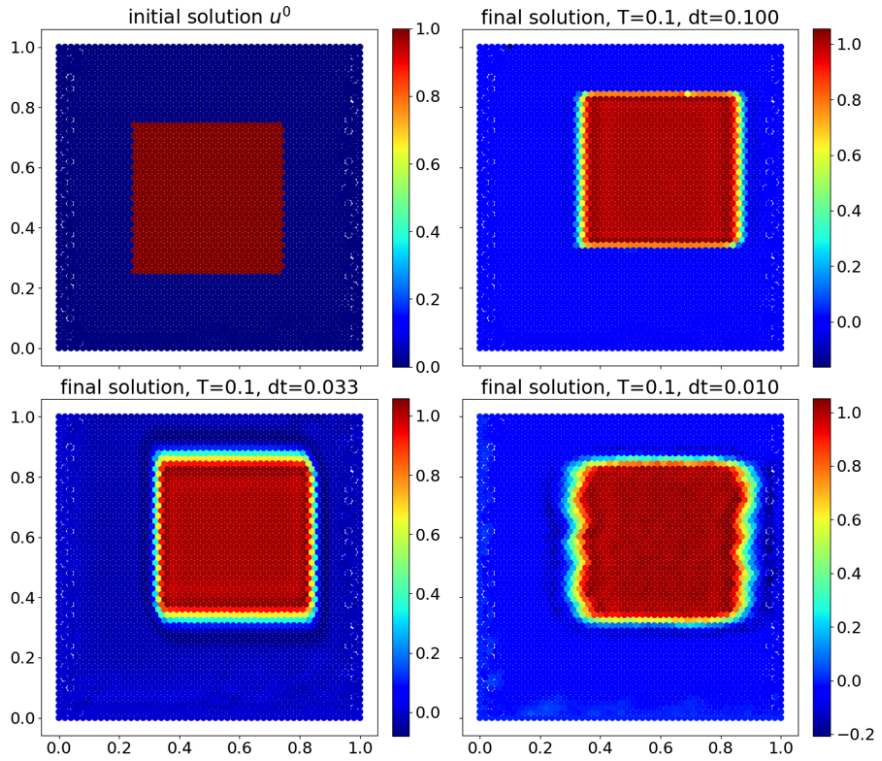

Figure 23: Solutions computed using the U-Net interpolation model.

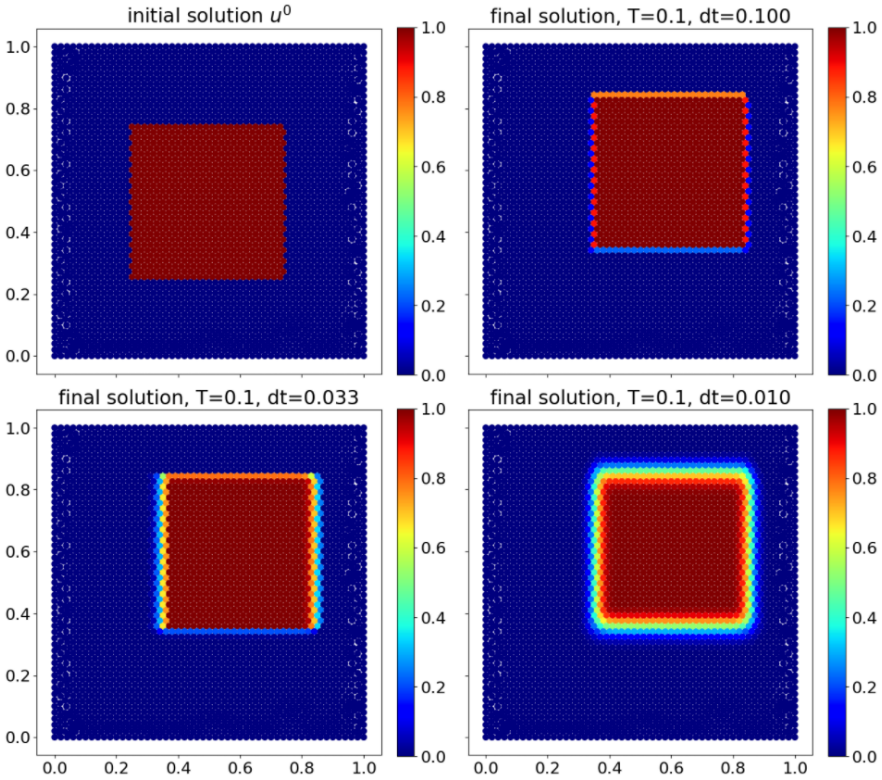

Figure 24: Solutions computed using the `griddata` interpolation operator.

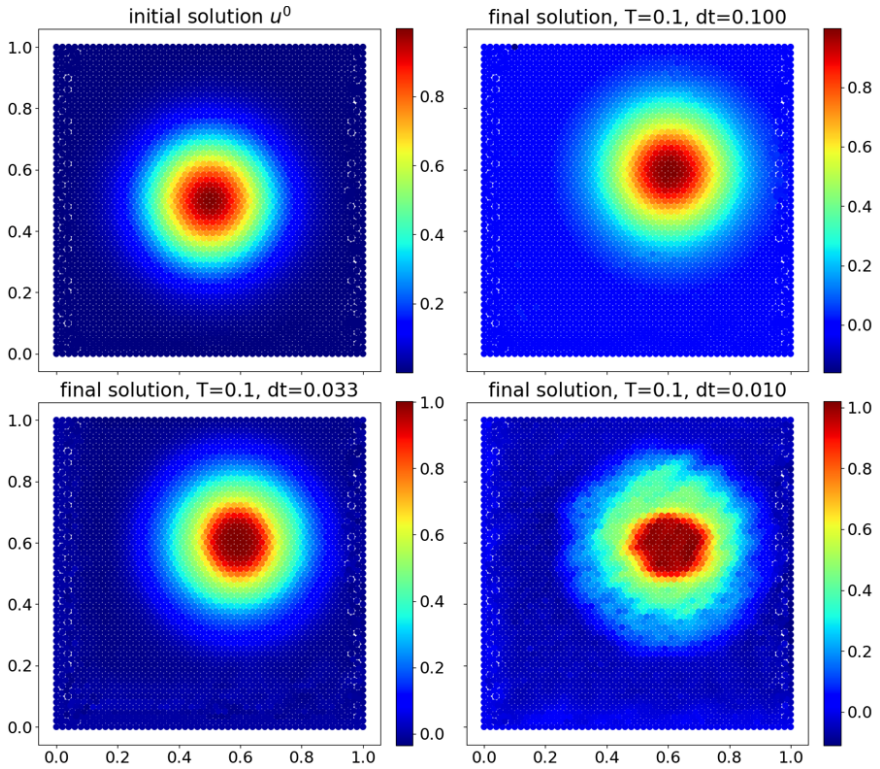

Figure 25: Solutions computed using the U-Net interpolation model.

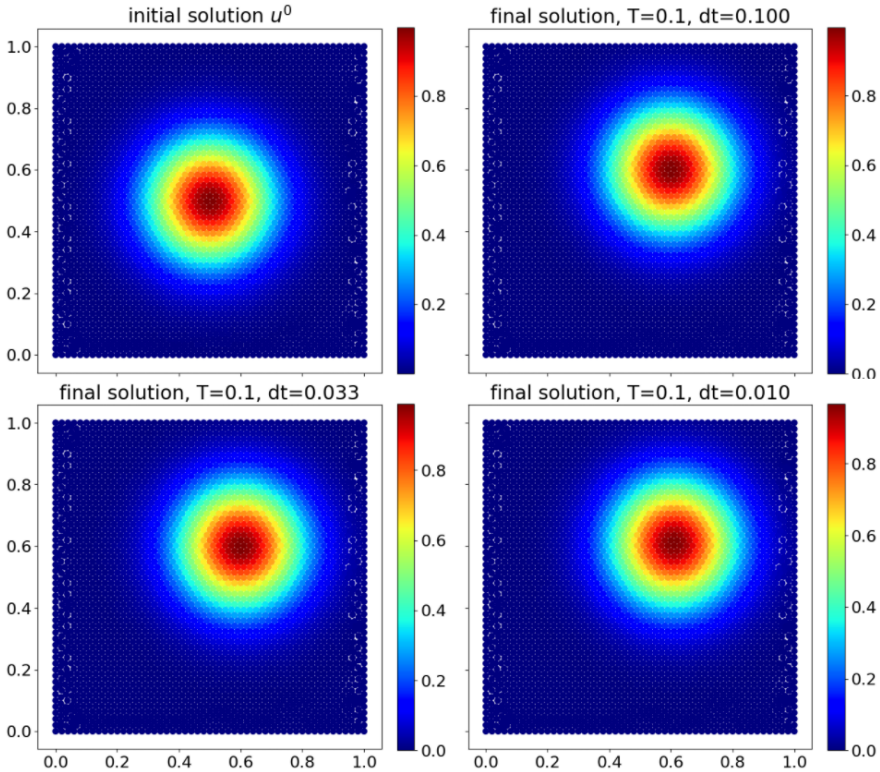

Figure 26: Solutions computed using the `griddata` interpolation operator.

6 Conclusion

We saw in this internship that the U-Net architecture, usually used by conventional convolutional neural networks, can be efficiently used in the context of graph convolutional neural networks. In fact, the U-Net architecture allowed us solve the frontier detection problem, problem that simple sequential GCNs were not able to solve.
This breakthrough allowed us to build an effective discontinuity detection model. This model was then used to compute precise Burgers' solution by using dynamic refining. This pipeline of dynamic refining could however be improved. In fact, a model indicating where the mesh could be unrefined would be a big improvement. Moreover, we saw that we had to limit the total number of refinement because of the time step constraints. Using another method than the finite volume method, without time constraints, would allow us to refine our mesh without limitations.
Finally, a variance of this U-Net frontier detection model allowed us to build an interpolation operator and to solve linear transport equations using the semi-Lagrangian method.
However, the model in its current state does not yield acceptable results. Indeed, we saw that the interpolation operator in unstable, resulting in bad results after multiple iterations. To try and solve this issue, we could change the training dataset. Moreover, we could change the way the model is trained so that it becomes stable by multiple successive compositions. To do so we would need to train the model based on its output after a fixed number of compositions, and not after only one iteration.

7 Internship and tools used

7.1 Internship

7.1.1 Organization

This internship was realized at the UFR de Mathématiques et d'Informatique at Strasbourg. Due to the pandemic, I mostly worked remotely from my home. Moreover, weekly video-conferences where organized with my supervisors. It allowed me to present the work I had done and the results I got during the week. Furthermore, it allowed us to talk about my problems and try to solve them.
The internship lasted 8 weeks. During the first few weeks, I mostly worked on the frontier detection problem. I tested and developed multiples convolutional and pooling layers, until I had satisfying results.
During the time left, I simultaneously worked on the Burgers' equation and the transport equation/interpolation problem.

7.1.2 Experience earned

This internship was enriching for me and allowed me to develop new skills and/or improve skills acquired during the M1, such as:

- Machine learning: I learned new machine learning algorithms and pipelines, which makes me now more comfortable in this domain.
- PDE solving methods: I also studied new methods for resolving various PDEs.
- Python development: I learned concepts about Python development that I was not aware of before. It made the programming during my internship easier and smoother.

7.2 Tools used

The totality of this project was coded in Python using the Visual Studio Code integrated development environment. This IDE allowed me to write and execute code directly on the v100 remote machine made available to me by the University of Strasbourg. This remote server contains a v100 GPU, which is fast and efficient for machine learning projects.
The machine learning models where created using the library Keras and Tensorflow [1]. It is a library created and maintained by Google facilitating the training of neural networks.
The library Spektral [4] gave me access to the VanillaGCN and ChebConv convolutional layers, and made me able to create pipelines for dataset generation and loading.
The meshes were created using the Python API PyGMSH of Gmsh (github).
The Frontier dataset generation code was written by Vincent Vigon. All the other features where coded by myself and are available on Github (link).

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] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Spectral clustering with graph neural networks for graph pooling. In *International Conference on Machine Learning*, pages 874–883. PMLR, 2020.

[3] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *international conference on machine learning*, pages 2083–2092. PMLR, 2019.

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

[5] Patrick Reiser, Andre Eberhard, and Pascal Friederich. Graph neural networks in tensorflow-keras with raggedtensor representation (kgcnn). *Software Impacts*, page 100095, 2021.

[6] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.