BSc report Applied Mathematics

"Simplicial Neural Networks in a physical application"

Timon Badier 4925238

Technische Universiteit Delft

Supervisor

Dr. ir. D. Toshniwal

Overige commissieleden

Prof. dr. ir. A.W. Heemink	Dr. ir. D. Toshniwal

July, 2021	Delft

Abstract

Artificial Intelligence in the form of neural networks is becoming wide spread. This report focuses on a specific form of neural networks, Simplicial Neural Networks. After presenting their advantages and how they were implemented in Python by using the code of [1], they are tested in 2 experiments to explore their applicability in solving physical problems. The first experiment aimed to test the feasibility of the approach as well as to compare them to traditional neural networks. The second experiment aimed at testing the use of SNNs in predicting pressures through a Stokes Flow when the flow is known. Although the experiment was carried out incorrectly the network still produced accurate results in the context of the experiment, and SNNs could present an alternative to Finite Element Solvers.

Introduction

Formulating a physical problem into equations is an important part of a problem solving approach implicating physics. However solving these equations can be an equally challenging problem as not all equations can be solved exactly through calculus. To obtain a solution to such an equation, a computer and numerical methods are used. Numerical methods operate on a discretization of space, and often time, to obtain a solution. Some imprecision is however inherent to traditional numerical methods, and it can lead to incoherent results. When shrinking from surface or volume equations to local equations in order to solve them with traditional numerical methods, physical meaning can be lost, for example conservation laws [2].

One solution is to integrate surface and volume metrics into the numerical methods. [3] proposes such a solution, making it possible to calculate exact vector quantities such as the divergence or the curl of a field. In order to so, a mesh where the physical quantities from $\mathbb{R}^3$ are associated not only to points in space but also to edges, surfaces, volumes. The quantities are then linked between each other using fundamental theorems of calculus. Such a data structure is called a cell complex by [3]. Using matrix calculations representing the fundamental theorems of calculus, exact calculations can be made for example for the gradient or divergence. The calculation with matrix multiplications of the vector quantities can however become resource intensive as the number of vertices, edges, etc increases and can loose effectiveness for non linear problems.

That is where Simplicial Neural Networks [1] come into play. A specific type of cell complex called a *simplicial complex* can be defined. [1] then combines a machine learning algorithm with simplicial complexes to obtain a *Simplicial Neural Network*. The approach was tested by making a simplicial complex of co-authorship data, and then removing some of the citation numbers. Subsequently, the Simplicial Neural Network was trained to impute the missing data. It was able to exploit the structure of the data and performed well to predict the missing citation numbers.

However, formatting physical quantities into a simplicial complex, it could be possible to impute missing vector quantities from the data. Training a simplicial neural network to solve a physical problem could therefore present an alternative to large matrix calculations to predict physical quantities. It could even present an advantage over the alternative for non linear problems. This would also mean that edge-, surface- and volume-metrics are still integrated in the structure of the data itself, and physical meaning such as conservation laws could be learned to the Simplicial Neural Network. For example, a Simplicial Neural Network could try to predict pressure through a Stokes flow given the divergence and velocity of the flow.

How relevant are Simplicial Neural Networks in a physical application? Why use neural networks in this particular application? What are the possible applications it could replace?

Contents

Chapter 1

Relevance of SNNs

This chapter will focus on the relevance of Simplicial Neural Networks (*SNN*). SNNs denominate the use of simplicial complexes as structure for the data in combination with a machine learning algorithm. First we will look at how Simplicial Complexes are defined, before looking at how they integrate with a neural network along with their possible applications.

1.1 Simplicial Complexes

This section will outline the definition of a Simplicial Complex (*SC*) and its geometric interpretation. A SC is a collection of finite sets closed under taking subsets [1]. A member of a SC is of dimension p if it possesses $p + 1$ elements. Each of its elements is then of dimension $p - 1$.

The easier way to look at SCs is to interpret them geometrically. The cardinality of an element can be associated to a geometric object. For example, elements of dimension 2 possess 3 elements, each of cardinality 1. This means an element of dimension 2 can be visualized as a triangle, with 3 edges, each edge being dimension 1. Each edge then has 2 vertices, with each vertex of dimension 0. An example of such a construction is visible on figure 1.1, with the blue surface a triangle ABC, with for example AB as edge.

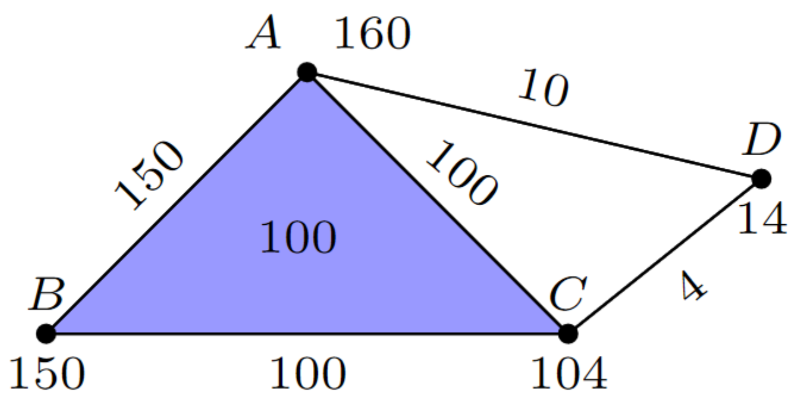


Figure 1.1: Simplicial Complex from [1]

Moreover, let's define a few mathematical objects. Let K_p be the set of simplices of dimension p from the simplicial complex K. Then we can define $C^p(K)$ the set of functions $K_p \to \mathbb{R}$ [1]. Such functions are called *p-cochains* and assign real values to each simplex, they are a means of encoding data. Going back to figure 1.1, these are the functions that attribute the value to each simplex. Predicting data numbers for the complex down the road will come down to having p-cochains as unknowns.

Without going into too much detail as [1] has the exact definitions layed out, *Coboundary* maps can be defined on $C^p(K) \to C^{p+1}(K)$ as a way of linking between the different dimensions of the simplicial complex. The coboundary maps use the coboundary matrices between each dimension, and using the transpose of these matrices they can also link the other way around between dimension $p+1$ and p. *Coboundary computational nodes*, called coboundaries in the rest of the paper, δ_i^j can then be defined, which take as input cochains from dimension i to output a cochain of dimension j. They will be used as nodes for the SNN and their exact computation depend on parameters that will come into play during the training of the neural network. *Simplicial Laplacians* are built on coboundary maps and carry information over the topology of the complex.

Finally one can define a *Simplicial Convolution*, a computational node on the cochains that will be used in the neural network. One important characteristic of the simplicial convolution is its low computational complexity: $\mathcal{O}(|K_p|)$. Secondly, it possesses a number of learnable weights that can be changed, as well as a parameter N that removes interactions between two simplices that are more than N-hops apart during the simplicial convolution. For example, if a simplicial convolution was to be defined, for dimension 1 and thus the edges, and had a parameter N = 1, that would mean that edges that are more than 1 edge apart would not affect each other directly. If applied to figure 1.1, the cochains for the edges BC and AD would not influence each other.

1.2 Simplicial Neural Networks

Now that SCs are defined, this section will motivate the use of SCs with machine learning. First the general workings of the machine learning algorithm will be explained before pointing out the advantages of this combined approach.

1.2.1 The Machine Learning Process

Machine learning needs three different components to function: a dataset, a neural network, and an algorithm to train the network with. The dataset will be used to train the network, as it gives a variety of inputs and desired outputs to the network. The algorithm is there to tweak the hidden weights of the network so that the input and output match as best as possible throughout the dataset.

The dataset is used to train the network. The larger the dataset is, the better the end accuracy and adaptiveness are after training at the cost of the training time. The data needs to include the output that the network should have after its computations. In our case, that means datasets will be constituted of different p-cochains, thus various vertex, edge and triangle values, like in [1]. The data will be passed on to the neural network.

The neural network is composed of *layers*, with each layers containing *nodes*, with a variety of input and outputs. The nodes themselves are a succession of basic mathematical operations with hidden parameters, called *weights*, with possibly multiple inputs and outputs. An example of such a structure is visible on figure 1.2. There the first layer possesses 3 nodes, the second one 2 nodes, and from 3 different inputs the network also computes 2 different outputs. In the case of SNNs, successive layers will be composed of simplicial convolutions, and coboundaries to link simplices of different dimensions.

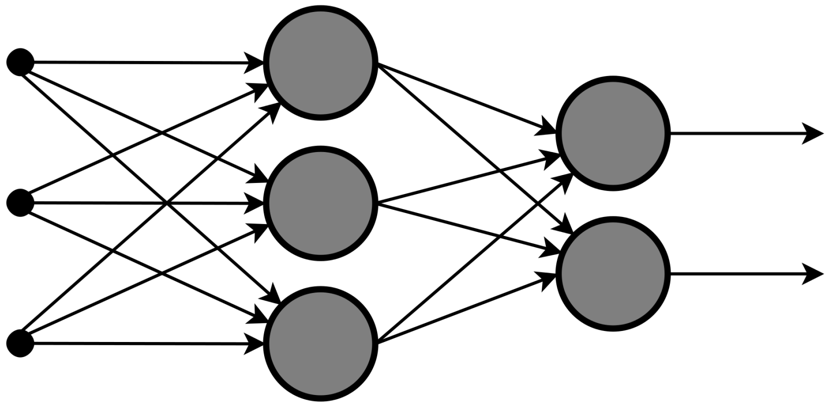


Figure 1.2: Example of a structure of a neural network [4]

In order to train, and thus adjust the weights of the neural network, an algorithm is needed. The neural network is fed the input and processes the output. Through a loss function, for example the mean squared error, the output is compared to the desired output. A *backward propagation* is then operated by a given *optimizer* from the last layer of the network to the first, and aims to adjust the weights of the nodes in order to minimize the loss. This operation is repeated over many iterations [5].

Without delving into details, admitting that the structure of the network is appropriate and complex enough, the output of the network should converge towards the output of the dataset. Once the network has been sufficiently trained, it is possible to feed the network new inputs and expect a reliable answer, as we will see in the case of simplicial neural networks in the next subsection.

1.2.2 Machine Learning exploiting Simplicial Complexes

Simplicial Neural Networks (SNN) are the use of simplicial complexes in combination with machine learning. This combined approach presents different advantages over traditional numerical methods operated on cell complexes and over traditional neural networks. As mentioned before, our data will be constituted of p-cochains, with the nodes of the network being simplicial convolutions and coboundaries.

Topology of the Simplicial Complex

First, as coboundaries and simplicial convolutions are used as nodes in the neural network, the network can be given a sense of the topological structure of the simplicial complex. How the topology should be exploited exactly is then learned to the network through training. For example, a network trained to impute missing data in [1] performed accurately to predict the entire data set when given only a fraction of it.

This property can be worth pursuing in an applied physics context, where different physical or vector quantities are present in different dimensions throughout the simplicial complex. It could also be possible to teach the network some physical laws such as conservation to obtain coherent results.

Adaptiveness of SNN

Another benefit of using SNNs is the adaptiveness of neural networks. First by building a general structure for a network, many different objectives could be set for its training and a user could save time by not having to meddle with the implementation of the network. Secondly, a trained network still shows a good adaptivity, as highlighted by [1]. In their experiments they trained a network to impute data on specific SC, called CC1. They then computed the output of the network on a different SC, CC2, and observed similar performance in accuracy. What is remarkable here is that CC2 has a different structure. In a physical context that could mean that a trained network could work with different spatial meshes, and thus not only different p-cochains but also different simplicial complexes, while keeping its accuracy. This would be once again a big time gain, and would remove the need to train the network for each specific spatial mesh.

Small Computational Complexity of a trained neural network

One last big advantage of SNNs is their low computational complexity once the network has been trained. Where traditional numerical methods using cell complexes could struggle with non linear equations, the network can perform low complexity convolutions to obtain predictions that would supposedly be accurate. This could be useful in real time applications and visualization where the uttermost precision is not needed. That means trained networks could possibly be replacing Finite Element Solvers referenced in [3] in these use cases.

Chapter 2

Implementation in Python

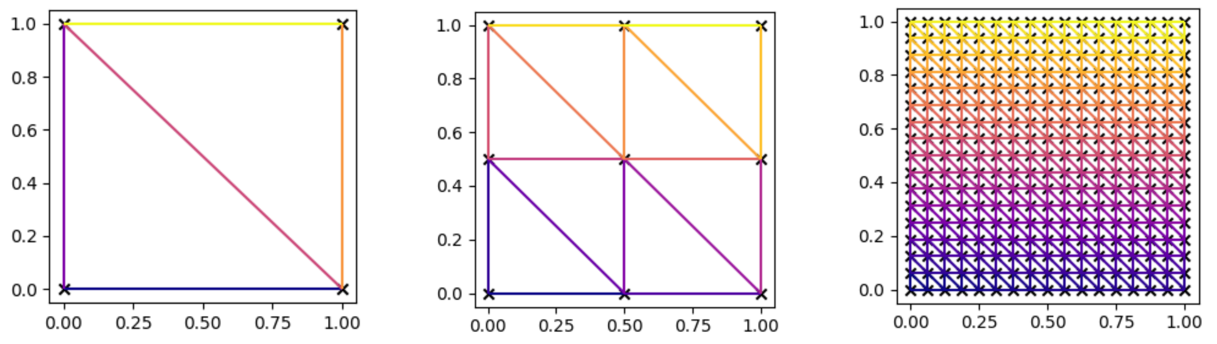
With the concept of Simplicial Neural Network defined, this chapter will focus on one possible implementation in Python. This is done by building on the code that [1] made available. The code was made in such a way that there would be different parameters that could be adjusted as the testing grew closer to testing in a physical context. This chapter will outline the main variables in the code.

2.1 Creating the Simplicial Complex

The first step of implementation was to generate the simplicial complex. Using a Jupyter notebook that [1] had prepared, with the python library `gudhi`, that was rather straightforward.

A mesh of points on $(0,1)\times(0,1)$ similar to one that could be used in a Finite Element Solver was generated, depending on a parameter `subdivisions`. For example the simplicial complex in figure 2.1a has 4 vertices, 5 edges, and 2 triangles. The `subdivisions` parameter would be used to increase the grid resolution, like in figure 2.1b and 2.1c. This way, training and testing the network on a small number of subdivisions could be possible before the time-consuming task of training it for a higher amount of subdivisions was tackled.

Python was then used to generate edges and triangles around this mesh (this project did not use more than 3 dimensions in the simplicial complex). To store p-cochains, 1-dimensional `numpy` tables were used. Using code from [1] it was possible to know how the indexing from the p-cochains tables to the corresponding vertices or edges or triangles was made.



(a) Simplicial Complex generated with 0 subdivisions

(b) Simplicial Complex generated with 1 subdivision

(c) Simplicial Complex generated with 4 subdivisions

Figure 2.1: Different Simplicial Complexes generated with different amounts of subdivisions. For visibility's sake the edges have different colors, the triangles are not colored.

2.2 Implementing the Neural Network

To implement and train the neural network, the library `PyTorch` will be used. The neural nodes are either simplicial convolutions or coboundaries, nodes that have both been implemented in python and made available by [1].

Simplicial convolutions need 3 parameters to be initialized, `N` as referenced in Section 1.1, the dimension of the input array and the dimension of the output array. That means that in successive layers of the network, instead of working for example with 5 simplices, the network can actually work with 5 times "`num_filters`", before bringing back to 5 the number of simplices on the last layer. To compute the output of a convolution, the simplicial laplacian of the corresponding simplex dimension needs to be given along with the cochain. Coboundaries need the same parameters for the exception of `N`, and need the corresponding coboundary matrices instead of the corresponding simplicial laplacians.

A layer is then built out of these nodes. For vertices, a simplicial convolution is operated on the vertex cochain and the output is summed with the output of a coboundary taking as input the edge cochain. For edges, we have a similar structure but with 2 coboundaries, one having as input the vertices, one having the triangles as input. Finally, for the triangles like for the edges, there is only one coboundary, having as input the edges. This structure for a simple layer is visible on figure 2.2, and is summed up in pseudo equations from 2.1 to 2.3. C_i's are the convolutions on their respective dimension, δ_i^j's the coboundaries from dimension i to j. $V/E/T_{in/out}$ are the input and output cochains. The network is then built on a succession of these layers.

$$V_{out} = C_0(V_{in}) + \delta_1^0(E_{in}) \tag{2.1}$$
$$E_{out} = C_1(E_{in}) + \delta_0^1(V_{in}) + \delta_2^1(T_{in}) \tag{2.2}$$
$$T_{out} = C_2(T_{in}) + \delta_1^2(E_{in}) \tag{2.3}$$

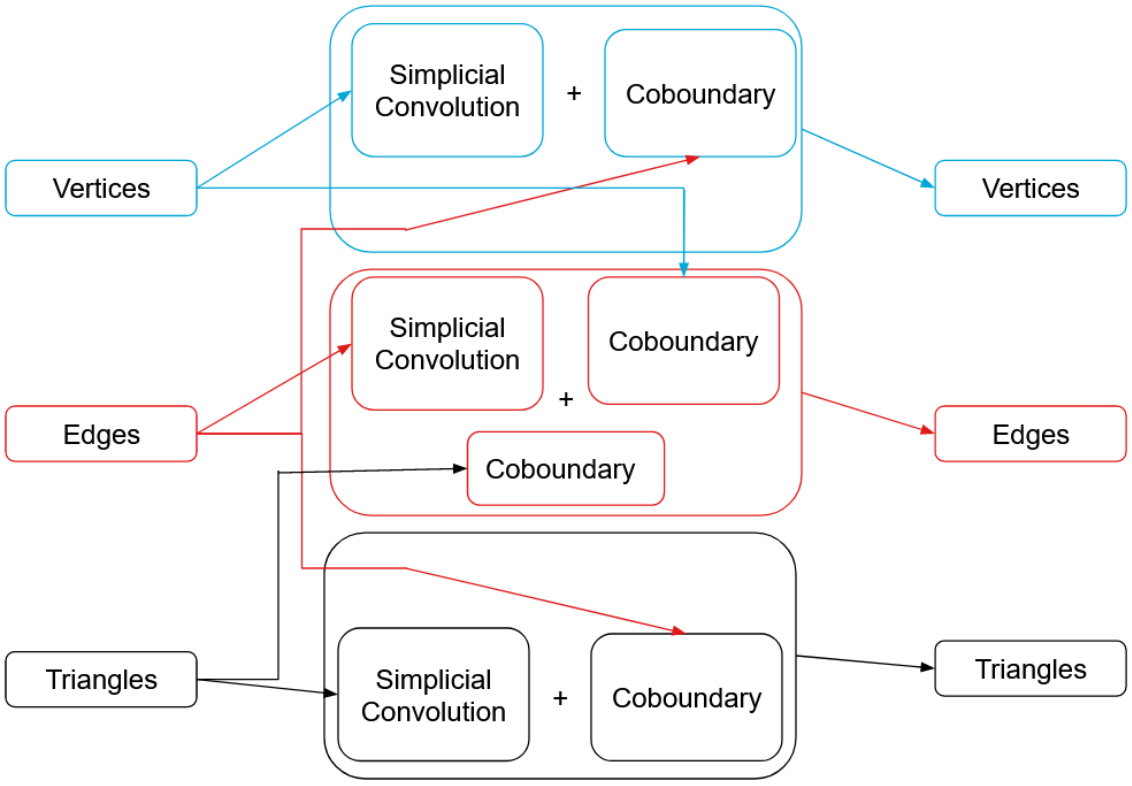


Figure 2.2: General structure of a layer in the network

Chapter 3

Experiments

Now that SNNs have been defined and their implementation in Python has been outlined, this section will present the two main experiments. The first one as a proof of concept tested if the network could predict simple relationships between vertex cochains and edge cochains and compared the SNN to a traditional neural network. The second experiment would test the performance of this approach in predicting the pressure (face cochains) in a divergence-free Stokes flow when given the flow (edge cochains). Due to an error in the way the data was generated and was being fed to the network, the last section of this chapter (section 3.3) goes over what changes to the process were made last minute but were not carried out in training a network.

3.1 Simple vertex-edge cochain relations

The first step in testing the approach was to create simple relationships between vertex and edges cochains. A linear relationship was used to test the most basic proof of concept, before adding some non-linearity to it. To test if the approach was valuable compared to traditional neural networks, the performance of the SNN will be compared to a traditional neural network with an equivalent number of adjustable parameters.

Simple relationship

For this test, random vertex cochains were generated using Python, with values between 0 and 1. Using the coboundary matrix, the edge values were calculated manually to be $q_i - q_j$ (with q_x the value of the cochain for the point x and $i < j$). The triangle cochains were the flow along the edges of each triangle, thus 0. The network was fed the vertex cochains, and the loss function (Mean Squared Error) was evaluated on all 3 dimensions. Testing was done on a special cochain, where the vertex value was equal to its distance to the origin.

At first the network performed really badly with 0 subdivisions, and during training the loss would not converge to 0 no matter the learning rate and size of the dataset. That was the case because the coboundary node wasn't being used in the structure of the network. That was changed and layers were given the structure presented in Section 2.2. It then performed well and testing was scaled up to a 5 subdivision problem. The structure of the network was kept very simple, with only 2 layers. The first layer multiplies the length of the cochains used in the computation by `num_filters = 10`, and the second layer brings it back to its original length. The network was trained over 2500 iterations with a learning rate starting at 10^{-3} and ending at 10^{-6} for random batches of size 200 to 2000 as the learning rate lowered. Training stopped when the largest absolute errors on the testing cochains were 2 orders of magnitude smaller than the exact values.

The prediction on the test cochain after the training is visible on figure 3.1, with a biggest absolute error of $1.2 \cdot 10^{-4}$. The *Rooted Mean Square Error* (RMSE) on the prediction was $7.25 \cdot 10^{-5}$ for an average absolute value of $9.21 \cdot 10^{-3}$. The exact distribution of the error is visible on figure 3.2.

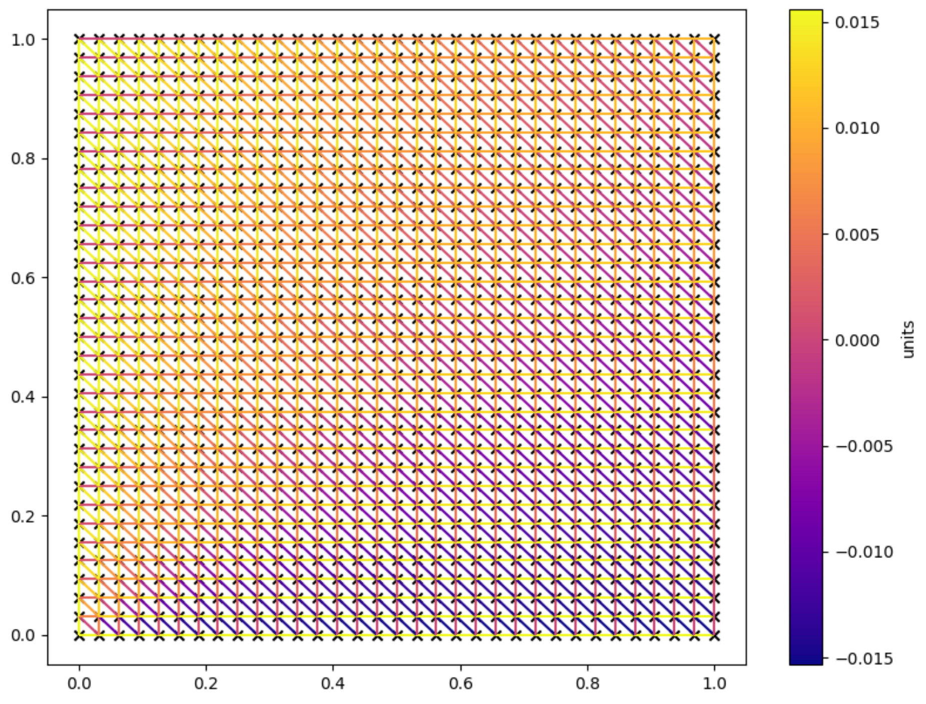


Figure 3.1: Prediction of the trained network of the edges on the testing cochain in the linear relationship test case.

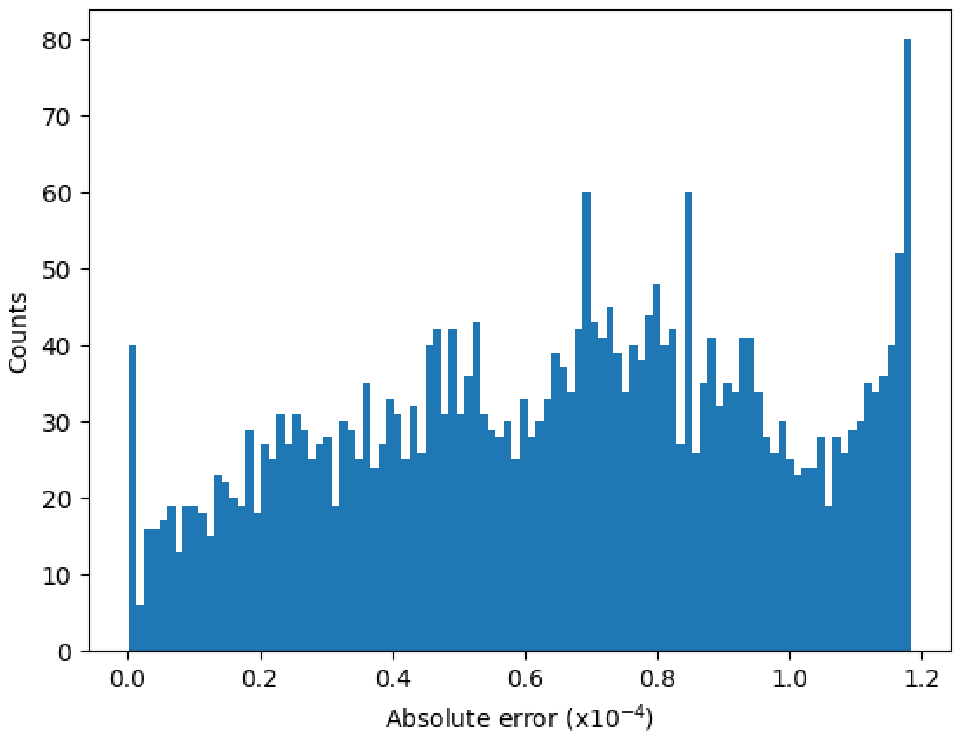


Figure 3.2: Histogram of the absolute error on the prediction of the edge cochain for the test cochain. RMSE : $7.25 \cdot 10^{-5}$

This accurate prediction, although on a simple problem, was a good sign that this approach could work.

Non-Linear relation

Since the linear relation prediction was accurate, the next test was to add some non linearity to the problem. Expanding on the problem exposed in the previous paragraph, the edge data that the network would need to predict was changed to be $f(e_i)$ with e_i the previous edge cochains, with $f(e_i) = \sin(e_i) + e_i^2$. This non linear function was arbitrarily chosen on the sole fact that it summed a trigonometric function and a square and it would be interesting to see if the network could predict this relation. The simplicial complex was also scaled back to a 0 subdivision problem for the training time gain from the 5 subdivision problem from the previous paragraph.

The network possesses 3 layers, and the loss was evaluated only on the edges. Data for the vertex values was generated at random for a fixed seed to reproduce results, but only a small amount of dataframes was used due to training time constraints. The batch size was thus of 50 dataframes. The optimizer was the ADAM optimizer, and the MSE loss function was used. The history of the loss during training is observable on 3.3.

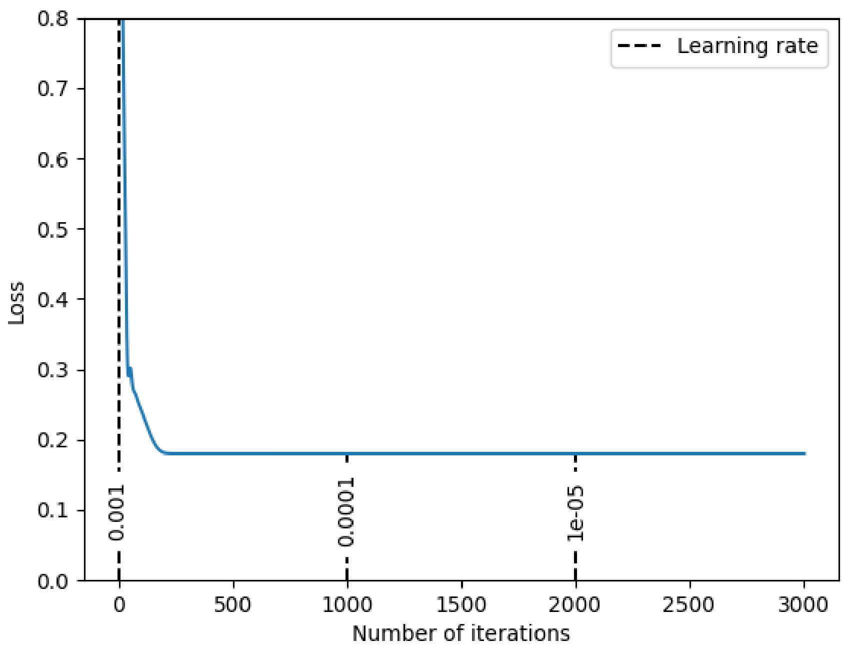


Figure 3.3: Evolution of the loss function for this network

After 3000 iterations, it was clear that the loss would not lower anymore than it did, and the batch size or the complexity of the network should have been increased. However, since the goal was to compare its performance to a traditional neural network, the training was stopped. The absolute error distribution on the batch of 50 randomly generated dataframes is visible on figure 3.4. Although the network's predictions could not be considered accurate, this network will be used to benchmark the performance of an SNN next to a traditional neural network in the next paragraphs.

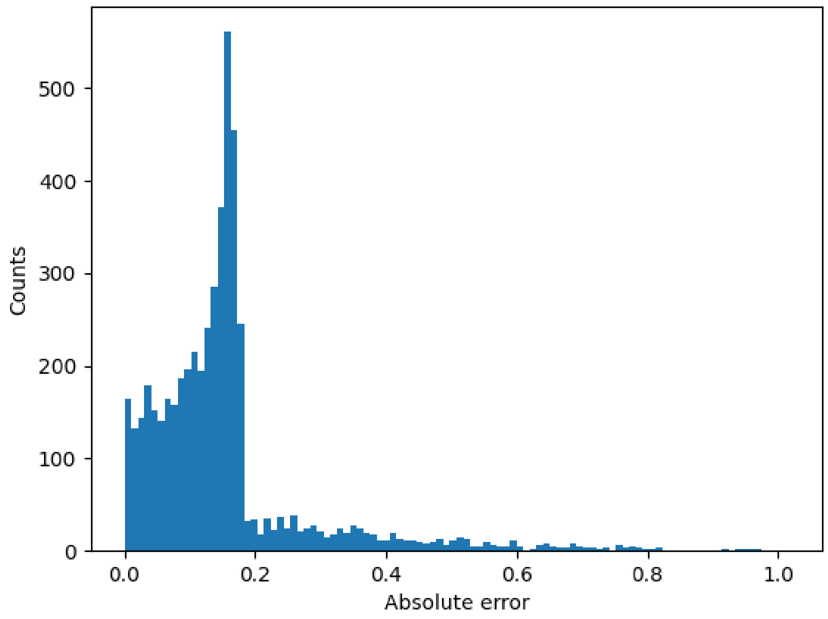
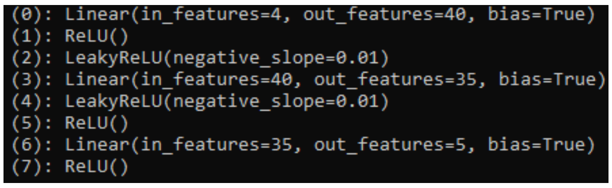


Figure 3.4: Distribution of the error after training on predicting edge value. Average exact edge value : 0.32, RMSE of the prediction : 0.46.

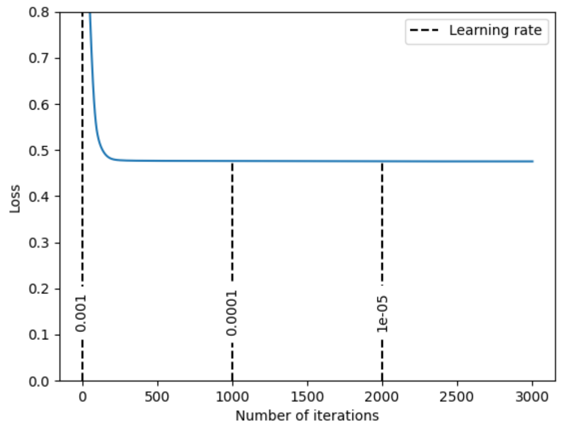
Comparison to a traditional Neural Network with the same amount of parameters

To compare the performance of the previous SNN to a traditional neural network, a traditional neural network was setup and trained on the same dataset. The previous SNN possesses 1707 adjustable parameters, and this network 1815. Its basic structure is visible on figure 3.5a. The network underwent the same amount of iterations in training and the learning rate was changed after the same amount of intervals, see figure 3.5b. Although the computer was faster to compute the iterations, the loss and precision of this network after the same training was worse than the equivalent SNN (figure 3.5c).

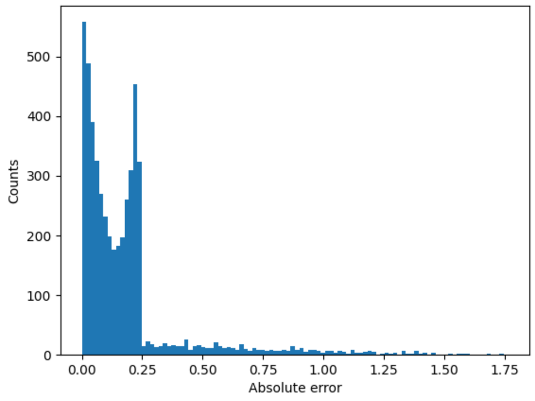


```
(0): Linear(in_features=4, out_features=40, bias=True)
(1): ReLU()
(2): LeakyReLU(negative_slope=0.01)
(3): Linear(in_features=40, out_features=35, bias=True)
(4): LeakyReLU(negative_slope=0.01)
(5): ReLU()
(6): Linear(in_features=35, out_features=5, bias=True)
(7): ReLU()
```

(a) Chosen structure for the traditional neural network.



(b) Evolution of the loss for the traditional network



(c) Absolute error on the prediction of the edges on the same dataset as on figure 3.4. RMSE of the prediction : 0.68.

Figure 3.5: Structure, training and performance of the traditional neural network with slightly more parameters than the previous SNN.

Although both networks performed pretty poorly in this experiment, at an equal training and slightly less parameters the simplicial neural network showed a slight edge in performance over the traditional neural network (RMSE of 0.46 vs 0.68). To draw conclusive results however, the experiment would need to be repeated with different relationships between the cochains and more dataframes, possibly at a number of parameters that is different. Another structure for the traditional neural network could also be more appropriate.

3.2 Flawed approach to predicting pressure cochains in a divergence-free Stokes Flow

Now that the proof of concept had been established as seen in Section 3.1, the next step was to test the network in a physical context. A 2-dimensional divergence-free Stokes flow has been chosen as problem, and the goal was to predict the pressure when given the flow. This could have direct applications as measuring the speed of a liquid in different parts of a flow is fairly easy compared to measuring its pressure.

As a disclaimer, there was a problem in my code, the data that ended up being used to train the network was not the right one. When the network was tested on new data, this new data was not generated as it should have been. However, due to time constraints

I was not able to get a network working on the correct data. The corrections to the experiment are present in the section 3.3

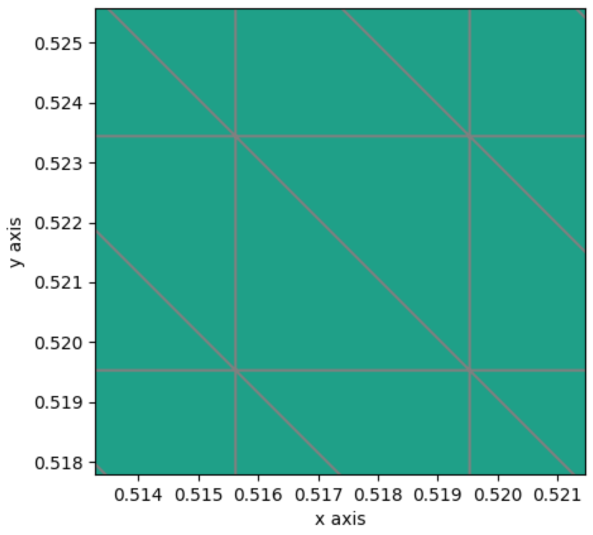
3.2.1 Generating the data

To generate data to train and test the network, my project supervisor D. Toshniwal wrote a python script that could solve a Stokes flow (Equation 3.1) on the same meshes as on figure 2.1. It uses the finite element solver FeNiCs, and is parametrized with the same parameter `subdivisions` to tweak the size of the mesh. The other important parameter is the boundary condition for the flow : the speed of the liquid. Vorticities of the liquid are associated to vertices (`"Lagrange"` finite elements), speeds of the liquid to vectors normal to the edges (`"RT"` finite elements), pressures are points in the center of the triangles (`"DG"` finite elements). The boundary conditions are the tangential speeds of the liquid to the boundary along the 4 different walls of the domain. I however first thought the boundary conditions were the constants for the speed of the liquid inputted in the Dirichlet equations at the borders.

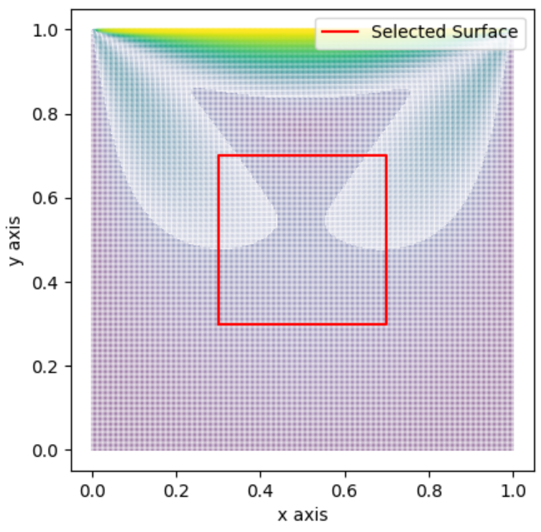
$$\mu \Delta \vec{v} = \vec{\nabla} p \tag{3.1}$$

The first approach was to generate data with what I thought were different boundary conditions on a specific number of subdivisions (2 to 4) that allowed the script to solve the stokes flow problem and also train the network directly on the same number of subdivisions. However this low number of subdivisions created unreliable results from the solver, and this already relatively high number of edges and triangles made the training long for the network. Closeness to the boundaries also produced unreliable results that rendered the network very unreliable when tested.

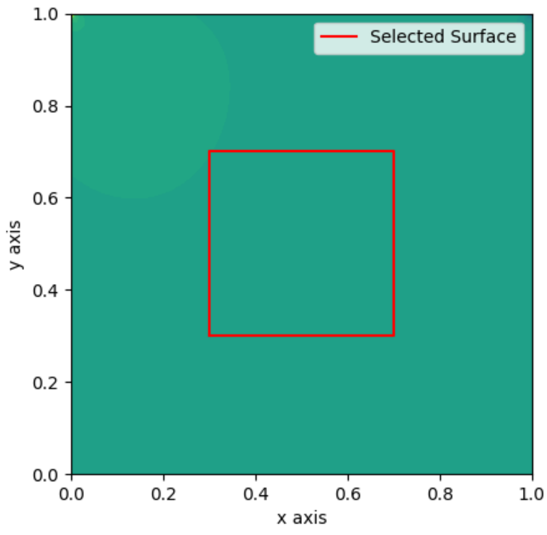
Thus the approach changed to choosing $(0, 0)$ for boundary conditions. The script would then solve the problem for 8 subdivisions, for a total of 66 049 vertices, 197 120 edges, 131 072 triangles. Once solved, each group of 4 vertices forming a square (with 5 edges and 2 triangles) was isolated from the mesh to be considered one element of a dataset from a 0 subdivision problem as visualised on figure 3.6a. To remove boundary induced incoherence, the square had to be $\epsilon = 0.3$ away from the closest boundary, see 3.6b and 3.6c. For this high number of subdivisions, this ϵ value was arguably too large and could have been reduced to increase the size of the dataset while still removing boundary induced incoherence, but the size of the dataset was already large enough and was kept at 0.3 for the rest of the experiments. This approach lead thus to having 10404 sets of 0 subdivision cochains.



(a) Zoom on part of the mesh to what is brought back to a 0 subdivision simplicial complex starting from a 8 subdivisions mesh.



(b) Flow velocity



(c) Pressure

Figure 3.6: Figures obtained when solving this Stokes Flow problem for 8 subdivisions and a $(0,0)$ Dirichlet boundary condition for the flow of the liquid. Figure (b) displays the flow velocity, and not the vorticities that were wrongly saved as edge data.

When saving the data, the tools of FeNiCs should have been used but instead the program went through the mesh and evaluated manually the value of the different components. That is where another problem is located with this experiment, vorticities were saved as flow of the liquid and flow as vorticities. Instead, edge cochains should have been the flow of the liquid normal to the edge between 2 vertices. For triangles, the average between the 2 pressures of the triangles was calculated and substracted to both values before being saved as the triangle cochain. This was motivated by the large negative numbers for the pressure that the solver was producing compared to edge values, as a way to somewhat normalize the values that the SNN would have to predict.

3.2.2 Training the Simplicial Neural Network

Now that data had been generated, it was possible to train the network, since it was realised only later that the wrong data was being fed. The network was given the edge cochains, and thus the vorticities instead of the flow, and was evaluated during training on performance on predicting the edges and triangles cochains.

The built-in *Mean Squared Error* (MSE) criterion was used for the loss calculation. Since triangle values were the main interest and were order of magnitudes smaller than edges values, their error was weighted by 100 000 but the criterion still evaluated the edge prediction. The optimizer was the built-in ADAM optimizer, with different learning rates depending on the stage of the training. Trial and error was used to determine the number of layers and the value of `num_filters`, but 4 layers were used in the results presented in the next section. `N` was left at its default value 5 (from [1]'s code), but it could have been lowered as in our case the maximum number of hops possible in this simplicial complex was 2. The exact parameters can be accessed in the code.

With 10404 different cochains the dataset to train on was relatively big, and the 32GB of RAM of my computer was not enough to train on the entirety of the dataset. The network was thus trained on fractions of the dataset. Training started on a learning rate of 10^{-4} with $10404//100 = 104$ cochains selected uniformly throughout the dataset. The number of cochains was then raised to $10404//70 = 148$ from the learning rate 10^{-6} and on. The evolution of the loss and the learning rate in function of the iterations is present on figure 3.7. The fact that the network was trained over multiple time spans explains the sudden spikes in loss that are visible on the figure, for example at 6500 iterations.

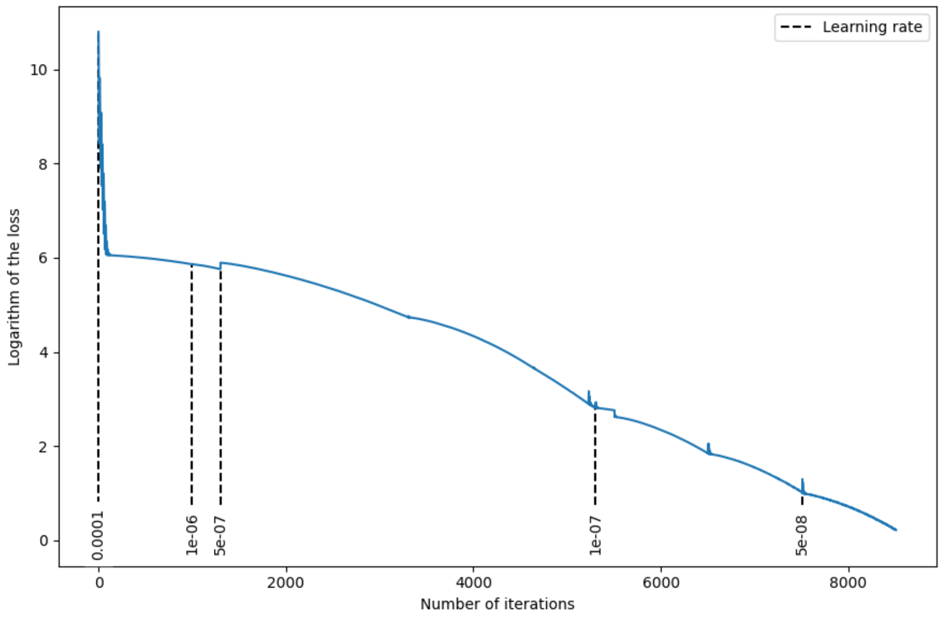


Figure 3.7: Logarithm of the loss in function of the iterations during training

After this training phase, the network produced accurate predictions for triangle cochains as we will see in the next section, but not for edge cochains. This was understandable due to the low amount of training and the fact that triangle loss was weighted. It is expected that if training was to be continued, the network would gain accuracy on the edges as it would just need to send back untouched the edge values that it was fed.

3.2.3 Prediction Results and Testing on New Cochains

The network trained, its performance was tested. The results of this section are made with all the 10404 cochains, as computing just the network's output was possible. The lack of RAM of the computer was a problem only for the back propagation of the training. They are also obtained with the same approach as earlier in the section, thus with vorticities instead of flows for edges values.

Accuracy on the training data

First let's look at the accuracy of the prediction for the dataset it was trained on. The average absolute value for the triangle cochains is $3.70 \cdot 10^{-3}$. The network produced predictions with a RMSE over all triangle values of $1.53 \cdot 10^{-4}$. The histogram of the absolute error is visible on figure 3.8. The prediction is thus arguably accurate, with the RMSE being an order of magnitude smaller than the average absolute value, which is surprising since the network wasn't working with the data it was supposed to.

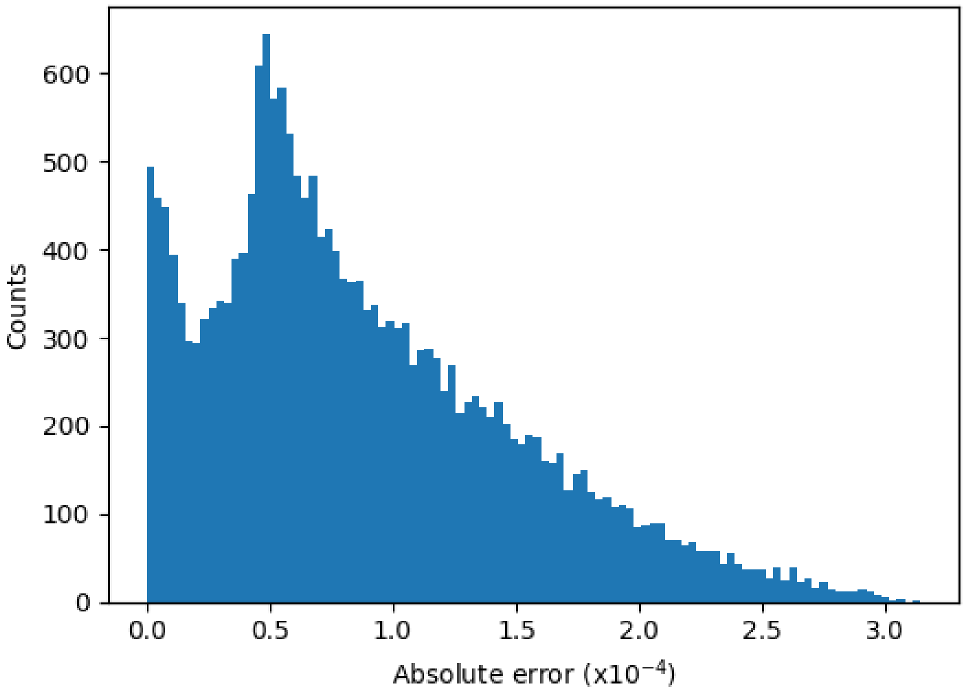


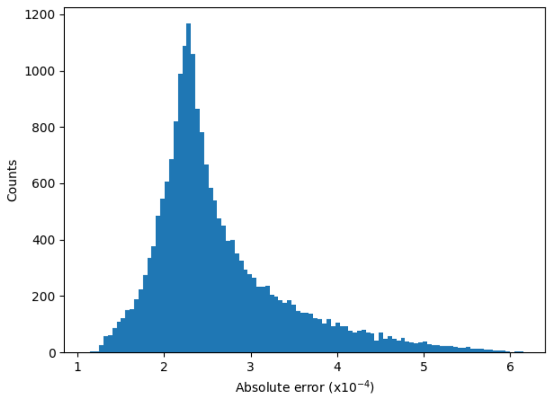
Figure 3.8: Histogram of the absolute error over the whole dataset of triangle cochains when making a prediction using the trained network.

Accuracy on new data

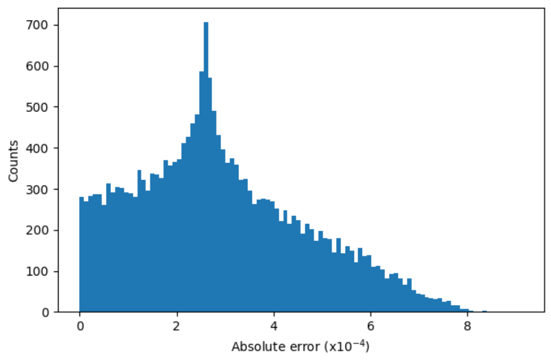
The accuracy of the prediction on the training data confirmed, new data was generated using the same process as in Section 3.2.1, but with (what I thought to be) new boundary conditions. 2 different boundary conditions were chosen at random: $(1,1)$ and $(-5,4)$ for the flow of the liquid outside the domain in the Dirichlet equations at the border.

For the boundary condition $(1,1)$, the average absolute value for the pressure cochains is $1.11 \cdot 10^{-2}$, and the RMSE for the prediction is $3.89 \cdot 10^{-4}$, thus 2 orders of magnitude smaller than the values it needed to predict. The histogram of the absolute error is visible on figure 3.9a.

For the boundary condition $(-5,4)$, the average absolute value for the pressure cochains is $1.21 \cdot 10^{-2}$, and the RMSE for the prediction is $4.80 \cdot 10^{-4}$, thus also 2 orders of magnitude smaller than the values it needed to predict. The histogram of the absolute error is visible on figure 3.9b.



(a) Histogram of the absolute error over the whole dataset of triangle cochains when making a prediction using the trained network on data generated using a boundary condition of $(1,1)$



(b) Histogram of the absolute error over the whole dataset of triangle cochains when making a prediction using the trained network on data generated using a boundary condition of $(-5,4)$

Figure 3.9: Histogram of the absolute error when making a prediction using datasets with different boundary conditions.

The network that has been trained using a constant of $(0,0)$ in the Dirichlet equation at the borders is thus performing well when trying to predict pressure data when given edges that are issued from data generated using different parameters. Testing this network while changing the proper border conditions would have been preferable, but the next subsection will outline which changes I was doing to do the experiment correctly, although I lacked time to finish it.

3.3 Correcting the approach of the pressure cochains prediction in a Stokes Flow

A few last minute changes were done to carry out a new experiment, described as in the previous section, but with the correct data this time. The changes were implemented but time was lacking to train and troubleshoot the implementation of the network.

Changes to the FeNiCs code

First off in the generation of the data, the built-in functions of FeNiCs were used. The call `valueDofs = u.vector().get_local()` was used to obtain a list of all the cochains mixed for the vertices, edges and triangles, instead of evaluating the value of the vector in the coordinates of the corresponding simplice. That meant FeNiCs is taking into account the structure of the data. For example this means that instead of obtaining the vector for the flow at the center of an edge, we obtain the norm of the flow normal to the edge, as was the original goal when generating the data.

This approach however meant that the list of the data was ordered using a special map, a `dofmap`. Each vertex or edge or triangle posses a DOF. In order to extract the data back to a 0 subdivision problem for training time, it was necessary to know the coordinates corresponding to the data values. Using the call `tabulate_dof_coordinates()` it was possible to obtain the coordinates corresponding to the DOF. For each coordinate, the code brute-forced its way through the DOFs until it found the right DOF number, and then it could know where to look for the data value in the `valueDofs` array. This was needed since the code processed by square of points, and thus knew the coordinates of the target vertices/edges/faces but not their DOF number. This brute force approach meant a slow running time to generate data, and to gain time, the stokes flow was being solved on a 5 subdivision problem instead of 8, and produced less data frames.

These changes made sure the correct data was now being saved. The next step was to adjust the code for the SNN.

Troubleshooting the SNN code

Now that the data was being correctly generated, the neural network would need to be trained again. However due to the time constraint it was not possible to train the network correctly. To train the network faster, less training dataframes were used, and the quality of the prediction was evaluated only on the pressure cochain it was producing and not the edges. That lead however to the loss converging to a number > 0. To try and avoid that, the pressure was averaged over each 0 subdivision frame, but that also did not work to train the network accurately.

As last approach the SNN was trained on a whole dataframe with 3 or 4 subdivisions instead of 0 subdivisions dataframes. The goal was simply to predict the pressure of a single dataframe when it was always given the same one, but the loss also didn't converge to 0, whether working with 3 or 4 subdivisions. The problem here could be linked to the very small datasize, or parameters of the network, or the structure of the network self. Time was lacking to properly diagnose the issue and carry out multiple tests.

The results of this testing with new data was thus not conclusive at all.

Chapter 4

Conclusion

With neural networks becoming wide spread, one can look at their relevance in solving physical problems. Simplicial neural networks are a special form of neural networks, that present advantages over traditional neural networks, as well as properties similar to Finite Element Solvers that can make them useful in a physical application.

In this project, we explored their relevance in solving a physical problem, and tested a simple implementation in Python by building on the code made available by [1]. Although the main experiment with Stokes flow is very flawed, since it was surprisingly still able to produce predictions when given vorticities as edges instead of flows, it could indicate that SNNs present a relevant alternative to finite element solvers once they have been trained. After their training they are faster to compute a solution, and they could be used as realtime vizualization software or monitoring software. The error on the prediction is difficult to bound without testing for a trained SNN, but if its accuracy can be proved in a specific application a SNN could replace traditional solvers.

The experiments carried out were however very flawed, as outlined in the report. The testing of SNNs could also have benefited from an experiment where a different simplicial complex was used with a trained network, instead of testing only with new cochains. The results of this experiment could have let us know if an entirely different spatial mesh could be used with an already trained network. This possibility would be very useful to feed differently shaped physical problems to the same network.

Bibliography

[1] S. Ebli, M. Defferrard, and G. Spreemann. “Simplicial Neural Networks”. In: (2020).

[2] J. Blair Perot. “Discrete Conservation Properties of Unstructured Mesh Schemes”. In: *Annual Review of Fluid Mechanics* 43.1 (2011), pp. 299–318. DOI: `10.1146/annurev-fluid-122109-160645`. eprint: `https://doi.org/10.1146/annurev-fluid-122109-160645`. URL: `https://doi.org/10.1146/annurev-fluid-122109-160645`.

[3] R.R. Hiemstra et al. “High order geometric methods with exact conservation properties”. In: *Journal of Computational Physics* 257 (2014). Physics-compatible numerical methods, pp. 1444–1471. ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2013.09.027`. URL: `https://www.sciencedirect.com/science/article/pii/S0021999113006414`.

[4] *By Offnfopt - Cropped and removed text from File:Multi-Layer Neural Network-Vector.svg*. URL: `https://commons.wikimedia.org/w/index.php?curid=39533797`.

[5] A. Novikov et al. *Tensorizing Neural Networks*. 2015. arXiv: `1509.06569 [cs.LG]`.