Imperial College London

Department of Earth Science and Engineering

MSc in Environmental Data Science and Machine Learning

Independent Research Project
Final Report

Machine Learning models for more Green, Sustainable, Clean Renewable Energy: application to Wave Energy Convertors (WECs)

by

Angela Maria Castañeda Oviedo

Email: angela.castaneda21@imperial.ac.uk
GitHub username: edsml-amc121
Repository: https://github.com/ese-msc-2021/irp-amc121

Supervisors:

Dr. Rossella Arcucci
Dr. César Quilodrán Casas

September 2022

Abstract

Climate change has pushed us to research and invest in renewable energy sources, one of which could be Wave Energy Convertors (WECs). These devices design requires Computational Fluyd Dynamics (CFD) simulations, and their respective analysis. These simulations usually produce huge unstructured meshes, which are also graphs, and in which any classification and prediction model would be time-consuming to run again and again. Therefore, it would be ideal to effectively reduce the size of the data using AutoEncoders. This project aims to produce the best possible AutoEncoder for a given dataset by evaluating the two following methods: a first approach on how to combine Graph Convolutional layers with traditional AutoEncoders on graphs that share the same adjacency matrix; and testing and analysing different combinations of activation functions commonly used in Neural Networks. Moreover, two additional datasets would be studied and the combined results would be discussed.

1 Introduction

1.1 Context (Wave Energy Converters)

The ocean is a huge source of renewable energy. There are three main ways to obtain energy from the ocean: wave energy, tidal energy and thermal energy. Even though all these energies share the same source, the cause for each one of them is different: wave energy harvests the energy from the movements that the wind induces in the ocean, tidal energy is caused by the gravitational effects that the moon induces in the ocean and thermal energy comes from the sun's heating in the ocean's surface [KKAH17]. Many technologies are currently under research and development for each form of energy; and despite that all of them don't come without their challenges [AL18], various devices are almost ready to be commercially deployed [DPS09]. For Wave Energy, these devices are called Wave Energy Converters (WEC).

Given that any power generation device is only considered commercially viable depending on its tradeoff between power generation and costs [CDH+10], it is important to produce an optimal design for WEC devices. Various Computational Fluid Dynamics (CFD) simulations are carried out, whose purpose is to determine the device performance, but also to assess the survivability of the device under extreme marine conditions [Ste]. In these simulations, careful estimation of the environment variables [JPG18] is important to produce the best possible WEC design. For example, considering the location where the WEC device will be located, estimating the dynamic viscosity variable is important to establish the width of the flap of a point-absorber WEC device [WRHD15], since this width is highly related both to energy production and costs [CDH+10].

1.2 Project's objective

Imperial College London, City University, and many other universities, are currently working in a macro-project called WavE-Suite (or WaveSuite here) that targets the task of discovering accurate and efficient numerical tools for assessing the survivability of WECs under extreme marine conditions [Ste].

One of the mini-tasks of this macro-project involves applying Machine Learning methods to the CFD simulations produced so far. Given the large size of the data, it is desirable to find effective compression methods that allow the user to use classification and prediction models in a much easier to deal with compressed space. For this reason, the current project's objective is to produce an effective AutoEncoder design that effectively compresses and decompresses the WaveSuite data provided for the project.

Two methods were considered to produce the final model. The first one represents a first approach on how to combine Graph Convolutional layers (GCN) with the classic AutoEncoder. The second one evaluates how the different combinations of activation functions between the layers of the model determine the model's performance. Using these two methods, different AutoEncoder designs were proposed and tested on an environment variable mentioned previously: the dynamic viscosity.

2 Literature Review

2.1 AutoEncoders

An AutoEncoder is a Deep Learning method that, through the use of Neural Networks, takes some inputs and learns to compress and decompress them, thus reducing the size of the data. A classical AutoEncoder is comprised by two components: an Encoder and a Decoder. The Encoder component learns to produce representations of the original data in a so-called latent space, which has a much smaller dimension when compared to the original; and the Decoder learns to recover the original data using the latent space representations [BKG20] .

These techniques that seek to reduce the size of the data by creating representations in a latent space are called dimensionality reduction techniques, and the main objective of the creation of this latent space is to allow the implementation of classification and prediction models in this space, where the much smaller dimension will allow them to perform much more efficiently than in the original space [WHWW14].

The components of the AutoEncoder: the Encoder and the Decoder are both fully connected neural networks comprised by many layers. As a design recommendation, a Decoder 'structure should mirror the structure of the Encoder [Mic22]. Also, as a convention adopted by the Machine Learning community, the number of neurons used in each hidden layer tends to be a power of 2, and some sources even claim that doing this increases the performance of the models [PRSS$^+$22, TK93].

2.2 Graph Neural networks

As the name suggests Graph Neural networks (GNN) are Neural networks whose inputs are graphs. They originate from the desire to apply Machine Learning models to graphs and they have been used in many applications; i.e. drug discovery since drugs can be seen as molecular graphs; traffic prediction since roads and its intersections can be represented as graphs; and social media analysis; among many others [Les21b].

But just saying that GNNs are Neural networks whose inputs are graphs, does not explain their whole complexity. While powerful to represent many world interactions such as the ones previously mentioned; graphs lack structure. In comparison, images have a rectangular grid structure that easily allows traditional Neural networks architectures to incorporate concepts like Convolutional layers and pooling layers.

Convolutional layers take advantage of the grid structure by using shared weight filters, this allows a Convolutional layer to extract localized spatial features while also using a small number of parameters when compared against a linear layer in the same model. Additionally, a convolutional layer usually outputs a bigger number of channels when compared to the number of channels in its input, which expands the information available for the image [ON15].

Subsequently, the results of a convolutional layer are usually passed through a pooling layer, which summarizes the information in each channel of the previous step [GK20]. This combination of successive convolutional and pooling layers is usually called a feature extraction module in the literature [ZMW$^+$19, Nav19]. Applying this feature extraction module just before a fully-connected linear layer, significantly reduces the number of parameters to tune in that following linear layer of the model, and hence less computational resources are required [GK20].

Both convolutional and poling layers can be redefined so that similar concepts can be applied in Graph Neural Networks [Ham, MJK20].

2.2.1 Traditional Graph Convolutional layers

Unlike images, graph's nodes don't have any defined order or reference points, and in consequence taking advantage of its structure is significantly harder. For this reason, the nodes data from its neighbours must be added using permutation-invariant functions such as sum or mean [Ham] .

Taking inspiration from CNN, a Graph Convolutional layer also called GCN layer, aims to summarize for each node the data features of the immediate neighbours of each node [Ham]. That purpose is achieved by the following equation:

$$H^{(k)} = AH^{(k-1)}W^{(k)}_{neigh} + H^{(k-1)}W^{(k)}_{self} \tag{1}$$

Where [Ham]:

- H^k is the feature matrix at step k. Hence, H^0 are the initial features of the nodes.
- A is the adjacency matrix of the graph
- W matrices are the weights or trainable parameters of the neural network.

Note here that different weights are assigned to different features of each node, but every neighbour is considered equal when added, which represents a permutation-invariant function. Also, multiplication by the inverse of a diagonal matrix D that contains node degrees can be added, which in this case would be the equivalent of normalization [Ham] . Another form of normalization was proposed by [KW16a] in which the original normalization matrix is split in two squared roots and each one of those squared roots is applied in each side of the feature matrix.

One GCN layer will summarize information about a node and its neighbours, or in other words its 1-hop neighbourhood. By combining various layers, the new features in each k^{th} iteration will summarize the original data features of the k-hop neighbourhood of each node. It is worth mentioning here that contrary to conventional NN, in GNN adding more layers is not always beneficial due to a problem called over-smoothing [Les21a].

2.2.2 Alternatives to Graph Convolutional layers

GCN layers are not the only approaches proposed to redefine Convolutional layers in a GNN architecture. Here we mention two other famous approaches to the problem of summarizing k-hop neighbourhood information in updated node embeddings:

- **Message Passing Neural Networks (MPNNs):** Instead of tuning a weights matrix, a function is obtained that computes how much each node feature should influence the reference node we are currently updating [GSR$^+$17]. While this approach usually produces good results, computationally speaking it is very expensive and therefore it is only used for small datasets .
- **Graph attention networks (GATs):** Instead of assigning equal weights to every neighbour, different weights (here called attention weights) are assigned to each neighbour, and these are usually computed based on how similar the neighbours are according to the associated node embedding [VCC$^+$17]. GAT layers use less resources than GCN layers and usually produce better results, but don't use as much computational power when compared against MPNNs, and hence GATs are seen as a middle ground when it comes to feature extraction layers in a GNN.

2.2.3 Graph Pooling layers

Just as GCN layer, generalizing the concept of pooling layers from grids to graphs is not straightforward. Many strategies have been proposed for this task, and most of them try to perform the pooling by dividing the input graph into clusters and then applying a permutation-invariant function such as sum, mean or max to each cluster [MJK20]. However, the focus of this project is to combine GCN layers with AutoEncoders, while graph pooling layers are mostly a nice add-on. For this reason we will limit ourselves to use the simplest form of graph pooling which consists of taking every channel of the input as a cluster. This way we get an output of at most 3 times (if we use sum, mean and max together) the number of channels in the input graph.

2.2.4 Graph AutoEncoders

As expected, Graph AutoEncoders are models that compress graph data in a latent space and then attempt to reconstruct the original graph using the information in the latent space. However, this task is extremely difficult given that reconstructing an entire graph usually requires not only predicting the nodes features, but other characteristics of the graph such as the adjacency matrix; and in some extreme cases even the number of nodes must be predicted [RNS20].

Most Graph Autoencoders in the literature aim to reconstruct the adjacency matrix of a graph [KW16b, TYL22]. They do this by generating node embeddings and then use this embeddings to predict link existence between nodes.

The most famous framework proposed in the literature [KW16b] uses a Graph Convolutional Network as an Encoder. This Encoder is composed by successive GCN layers connected by RELU functions that provide non-linearity to the model. For the Decoder part, this framework uses an architecture called inner-product decoder. By using the node embeddings produced by the Encoder for each node, the Decoder simply computes the inner product between every pair of nodes and uses the result as the probability that a link exists between those two nodes. By selecting a threshold, the Decoder decides if those two nodes are connected and therefore estimates the adjacency matrix of the graph.

3 Methodology

The principal dataset of the project is the WaveSuite dataset. It is part of the WaveSuite project and was produced by CFD simulations. This dataset consists of various unstructured meshes or graphs with the following characteristics:

- All graphs share the same adjacency matrix.
- Each graph contains approximately 850000 nodes.
- Each node contains data of different environment variables, but only the value of the dynamic viscosity will be used, which leaves only one feature per node.

As explained before, this project aims to produce the best possible AutoEncoder design for the WaveSuite dataset, by using two methods: GCN and different combinations of activation functions.

First, a classic AutoEncoder will be implemented. Then, three alternative AutoEncoder designs that incorporate Graph Convolutional layers (GCN) will be proposed. The goal is to check if combining GCN layers with the classic AutoEncoder design will improve its results. However, it is important to note that the usual approach to Graph AutoEncoders aims to reconstruct the adjacency matrix of the graphs [KW16b, TYL22], but given the unique characteristic of our dataset in which all graphs share the same adjacency matrix, this project will instead aim to reconstruct the node feature values.

Later, each model will be tuned by testing different combinations of activation functions. Since the RELU activation function is the most frequent function used when it comes to connecting different layers of a model [1], it will always be used to connect successive layers of the same type. Furthermore, a group of successive layers of the same type within a model, will be called a module. The testing of different activation functions will be done in the locations that connect different modules within a model [2]. Every combination of activation functions in each model will be called a variant of that model.

Lastly, given the large size of the WaveSuite data and the limited time and computational resources of the project, it was decided that all the models and their variants will be first tested on two smaller datasets. After that, conclusions will be drawn and the most promising models will be tested in the WaveSuite dataset.

3.1 Datasets used in this project

The number of nodes in the WaveSuite dataset makes the evaluation and tuning of the proposed models very slow. For this reason, two smaller datasets were used initially: the air-pollution dataset and MNIST dataset. The distribution of the three datasets used is shown in Figure 1. The air-pollution dataset and the MNIST dataset both have values between 0 and 1, while the WaveSuite dataset has values between 0 and $7,2$ .Hence, the WaveSuite dataset will be standardized by dividing it by $7,2$ before passing its values through the AutoEncoders.

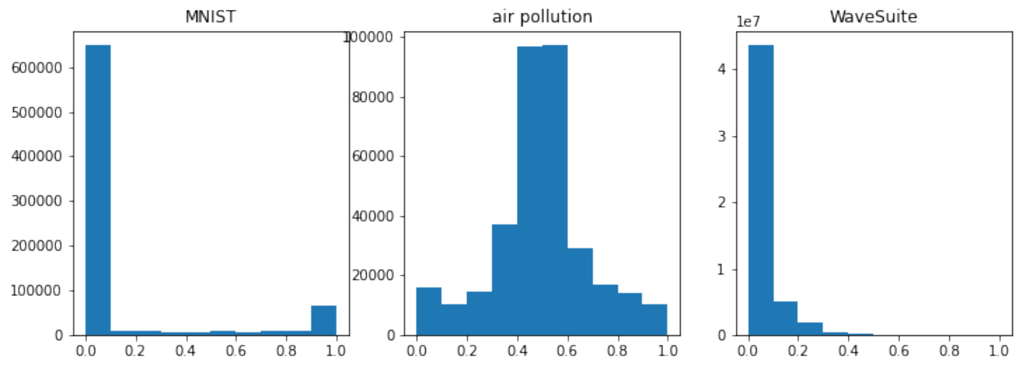


Figure 1: Datasets distributions

Once everything was running and preliminary results were obtained on these two smaller datasets, the top models were tested on the WaveSuite dataset.

3.1.1 WaveSuite dataset

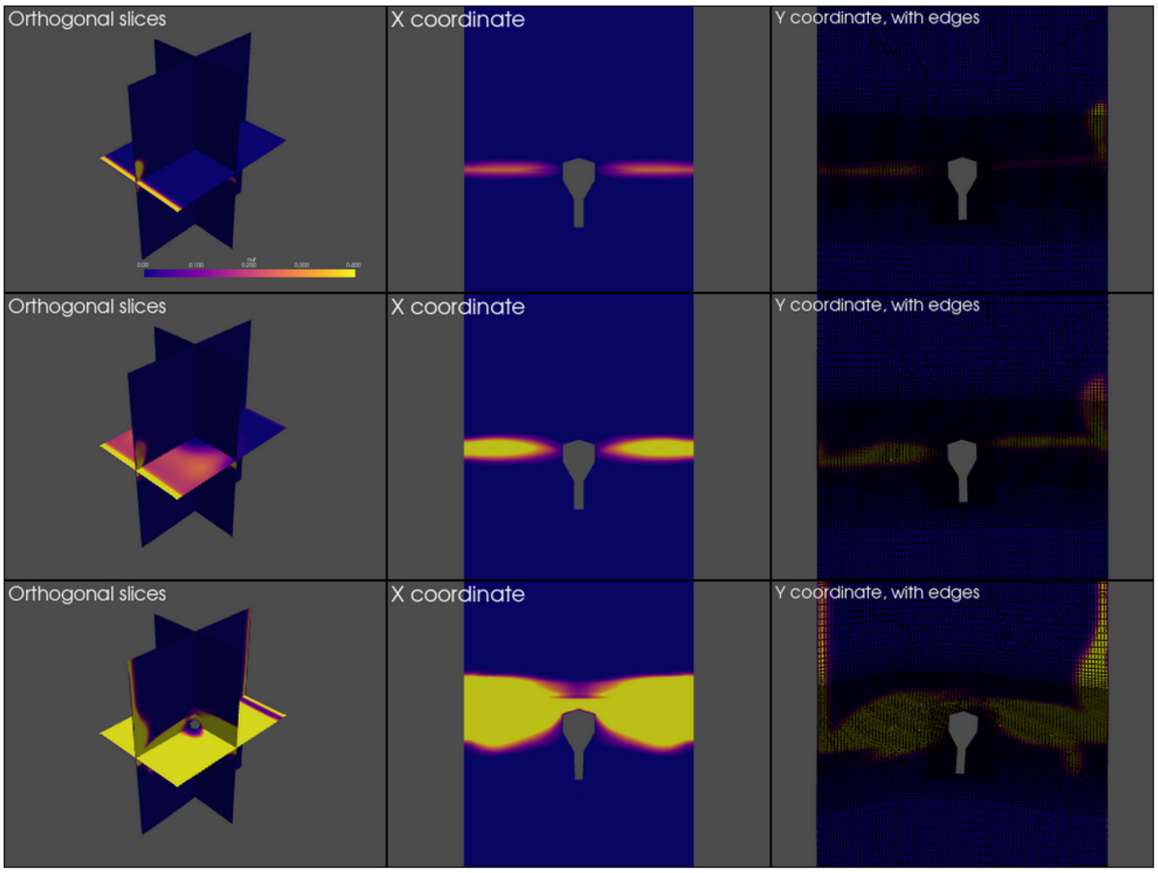
This dataset is comprised of 60 vtu files, each one containing a graph with 851101 nodes and 10552231 edges. Each graph was generated using an unstructured triangular mesh and it is contained inside a 3D box. All graphs share the same adjacency matrix and only the node features change. Since the only environment variable kept was the dynamic viscosity, each node has only one feature.

Three samples of the WaveSuite dataset are shown, one in every row of Figure 2a. Note that for each sample, there lies a buoy at the centre, which represents the WEC device. The colours around the buoy represent the values of the dynamic viscosity in its surrounding environment. For each sample it is possible to see:

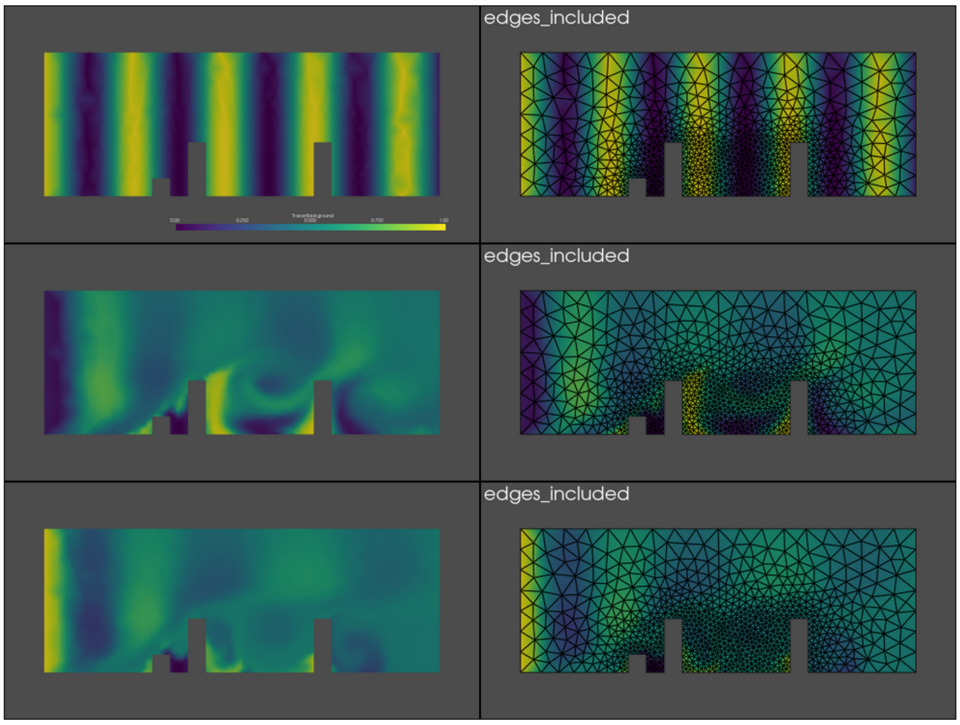
- The orthogonal slices of the 3D box that contains the graph

[1]The use of the RELU function is justified by the need to add non-linearity to the otherwise purely linear models

[2]Note that the locations that connect different modules are the ones that connect different types of layers



(a) WaveSuite examples



(b) Air pollution examples

Figure 2: Samples of the two datasets that contain unstructured meshes

- The X slice of the box
- The Y slice of the box, along with its portion of the triangular mesh

3.1.2 First preliminary results dataset: air-pollution dataset

The air-pollution dataset is comprised of 401 vtu files, each one containing an unstructured mesh, or a 2D graph in this case, with 852 nodes and 2419 edges. Just like in the WaveSuite dataset, all graphs in this dataset share the same adjacency matrix and every node has only one feature. In this case, the pollution concentration was selected as the feature of interest.

Three samples of the air-pollution dataset are shown, one in every row of Figure 2b.The similarities with the WaveSuite dataset make it possible to implement a code specifically for this dataset, that will also work on the WaveSuite dataset by just changing the input files.

3.1.3 Second preliminary results dataset: MNIST

In order to have another small dataset to make the preliminary analysis, the famous MNIST dataset was selected. MNIST is comprised of 10000 images of 784 $(28x28)$ pixels each. Since MNIST just contains one feature per node which is the values of the pixels, it is necessary to define its corresponding adjacency matrix. To do this, a list of edges is defined by connecting every pixel to all of its neighbours (including diagonals, which makes them at most eight).

Aside from that, the two previous datasets had an amount of samples bigger than their number of pixels per sample. While this is not ideal, it is a frequent issue in real-life datasets. In contrast, MNIST has a significantly bigger number of samples than pixels, which is a desirable characteristic in a dataset. To compensate this flaw [3] only the first 1000 samples of the MNIST dataset were used in this project.

In principle, the air-pollution dataset has more similarities to the WaveSuite dataset than MNIST has. However, as can be seen in Figure 1, the MNIST distribution is more similar to the WaveSuite distribution, when compared against the air pollution distribution.

3.2 AutoEncoder designs

As mentioned previously, the AutoEncoders implemented aim to reconstruct the node features of each graph; and in every dataset used each node of each graph has only one feature value. To input the information into the models, a node ordering was assigned to the nodes and the node feature data was then flattened into a 1D vector. Additionally, the adjacency matrix is presented in a sparse-representation called coordinate list where instead of a matrix the edges information is stored in a list of pair-connected nodes [QLW21]. The first model does not use the adjacency matrix but all others do. The architecture of the models is shown in Figure 3.

3.2.1 Classic AutoEncoder

The first model of the project is a classic AutoEncoder. The Encoder is a fully-connected Neural Network with input size the number of nodes [4] and output size the latent space dimension. Apart from the input and output layer, the Encoder has one hidden layer that doubles in size the output layer. This hidden layer uses RELU as activation function. Following the recommendations mentioned in the literature review, the Decoder is a mirror of the Encoder. However, there is no activation function between the Encoder and the Decoder. Since this model is composed by all layers of the same type, this model will be seen as entire module.

[3]It is obviously not a flaw, but it is a flaw in the sense that it makes this dataset different from the previous ones.

[4]Just as when working with images, the input layer size is the number of pixels.

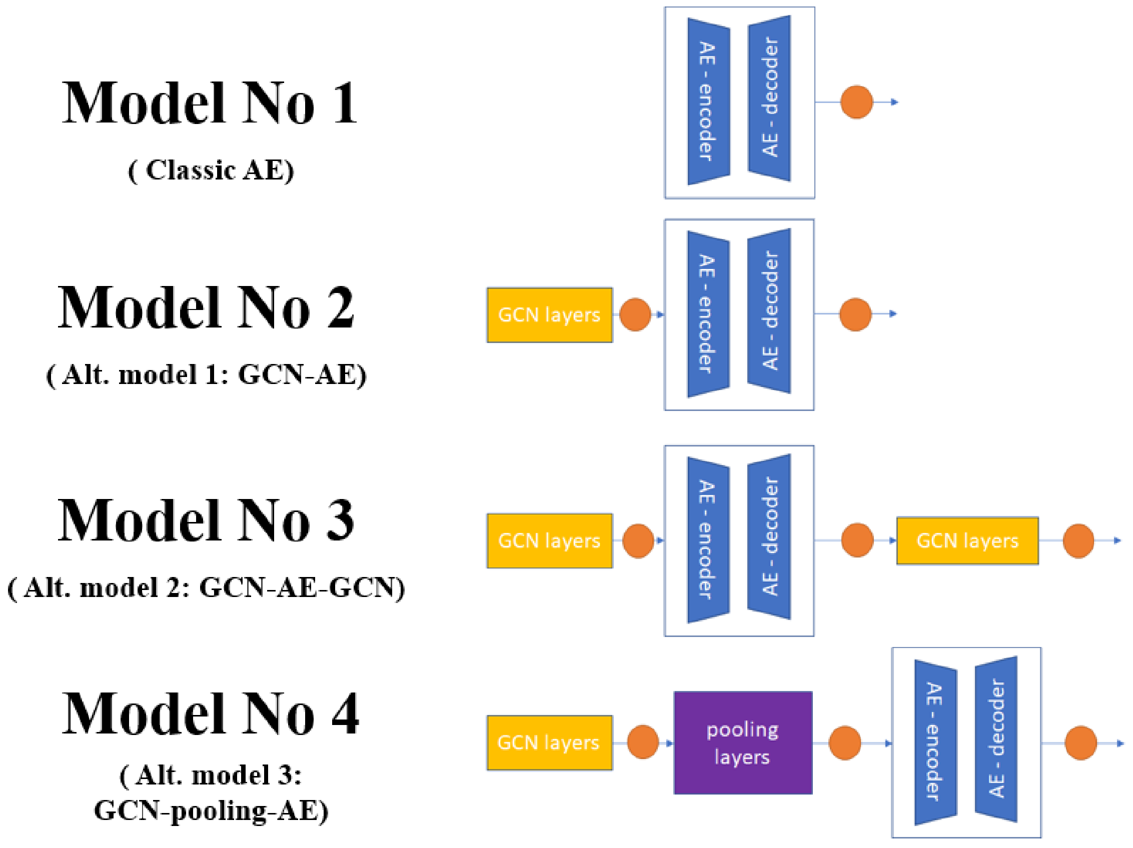


Figure 3: Proposed AutoEncoder designs

3.2.2 GCN-AE

The second model adds a module of successive GCN-layers before the classic AutoEncoder module. For this project, it was decided to use 3 GCN-layers where the hidden layers use RELU as activation function. The GCN-layers summarize the k-hop neighbourhood information for each node in various channels. This expands the information contained in the node features and the goal is to check if this allows the second module of the model to make better predictions. From now on, the sequence of numbers of channels of the GCN-layers will be called the embedding sequence of the GCN-layers.

Unfortunately, the number of points inputted to the second module ends up being multiplied by the number of channels of the last GCN-layer , which largely increases the parameters of the model. A preliminary analysis carried in this project determined that to improve the model's results from the classic AutoEncoder, the number of channels of the last GCN-layer must be at least 2. For this reason the used embedding sequence of the GCN-layers was: $[8, 16, 2]$.

3.2.3 GCN-AE-GCN

Inspired by the mirror principle of AutoEncoders, the third model takes the second module and adds an additional 3 GCN-layers module at the end of it. Consequently, in this new module the embedding sequence of the hidden GCN-layers was: $[2, 16, 8]$.

3.2.4 GCN-pooling-AE

The last model represents a first approach on how to incorporate graph pooling layers with AutoEncoders. This model takes the second model and between the two existing modules, adds a graph pooling-layers module. Ideally, the nodes would have been clustered in a smart way before applying the pooling layers, but due to time restrictions this step was skipped.

3.3 Tuning the models with different activation functions

In addition to proposing different AutoEncoder designs, the project also aims to test the effect of different activation functions in the results produced by the models. Different sequences of activations functions, located in the connection of two modules within the same model, were tested. To illustrate this, the locations of these activation functions are presented as orange dots in Figure 3. For each model, each combination of activation functions will be called a variant.

Initially, to construct the variants to test, three activation functions were allowed at the output of the models: None (i.e. no activation function at all); RELU; and Sigmoid. At the other locations only RELU and None were allowed. This combinations would make a total of 3, 6, 12 and 12 variants respectively for each model in that order.

Apart from the previous variants and exclusively guided by intuition, the following combinations were also implemented for Model_3: [None Sigmoid None], [None Sigmoid RELU], [RELU Sigmoid None], [RELU Sigmoid RELU]. Therefore, 37 variants were tested in total in the two small datasets.

4 Code metadata

All the code implemented in this project was made in python using jupyter notebooks. To read and plot the vtu files that contained the unstructured datasets, the packages pyvista and vtktools were used. Also, to implement the models described previously the torch-geometric package was used, which is an extension of the torch package.

All the jupyter notebooks were run on Windows 10. The device used to run these notebooks was a PC [5] with 32 GB of RAM CPU and a 8 GB graphical processing unit (GPU). The packages torch and torch-geometric can be configured so that they perform their computations using either the CPU or the GPU. To use the GPU, it is necessary to install a platform called CUDA.

The models and their variants finished training on the two small datasets about three times faster when the GPU was used instead of the CPU. However, the memory available in the GPU is smaller than the memory available in the CPU, and it was not possible to train the models on the WaveSuite dataset using the GPU. Additionally, it was necessary to implement a special torch-geometric class that allowed the execution to better manage large datasets such as WaveSuite; and even then the models training had to be partitioned in different notebooks.

5 Preliminary Results for the small datasets

The MSE loss will be used to rank the proposed models and its variants. It is worth mentioning that in a preliminary analysis the ADAM optimizer and the SGD optimizer (with various momentum and learning rates) were tested, and the ADAM optimizer performed significantly better.

5.1 Air-pollution dataset

The variants were trained on the air-pollution dataset using 50 epochs and a batch size of 32. The results are ranked using the MSE loss and they are presented in Table 1. Note that for every model the best variants are the ones that use the sigmoid activation function at the output of the model. The variants that have no activation function at the output produce similar results, but the ones that

[5] Alienware M15, GPU: NVIDIA GeForce RTX 2070 with Max-Q Design

use RELU have considerably worse results. Apart from the output activation function, varying the others activation functions seems to have a rather negligible effect on the model's results.

For every model, its variant that uses sigmoid activation function at the output and does not use any activation in the connections between modules is either at the top, or close to it with a negligible difference when compared to the one in the top. Therefore, these variants are plotted for each model in Figure 4. In this case, the best results were achieved by Model_2 and the worst results were achieved by Model_4.

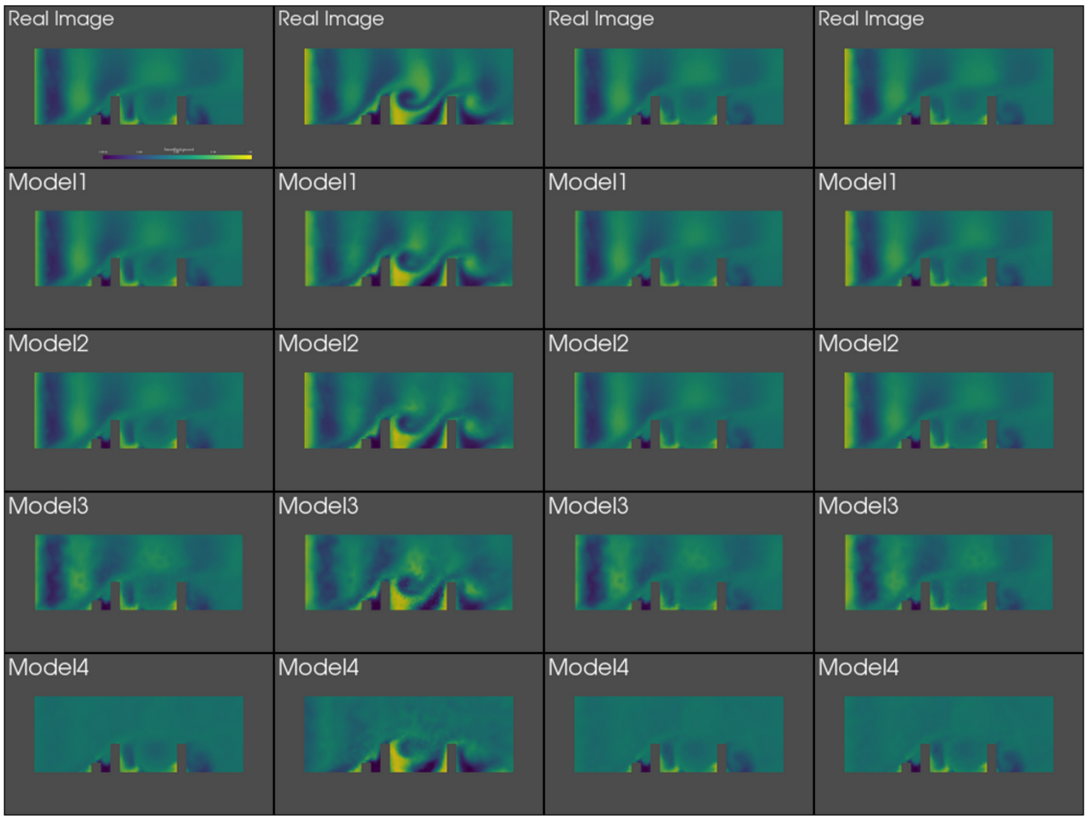


Figure 4: Comparison between the decompressed results for the proposed AutoEncoder models using the air-pollution dataset

5.2 MNIST dataset

The variants were also trained on the MNIST dataset using 80 epochs and a batch size of 32. The results are ranked using the MSE loss and they are presented in Table 2. In this case, the top variants are the ones that do not use any activation function, either at the output or between the modules. This means that contrary to popular belief for MNIST AutoEncoders, it is better to not use a sigmoid activation function at the output. However, using RELU or Sigmoid at the output also achieves very similar and good results and Sigmoid in particular guarantees that the decompressed values end up in the $[0, 1]$ range [6].

The variants that use Sigmoid as the output activation function are plotted for each model in Figure 5. In this case, the best results were achieved by Model_3 and the worst results were achieved by Model_4 .

[6]This means that for the models that have no output activation function; the values out of the [0,1] range of the decompressed images could be forced to lie in this range and that would improve the results even more.

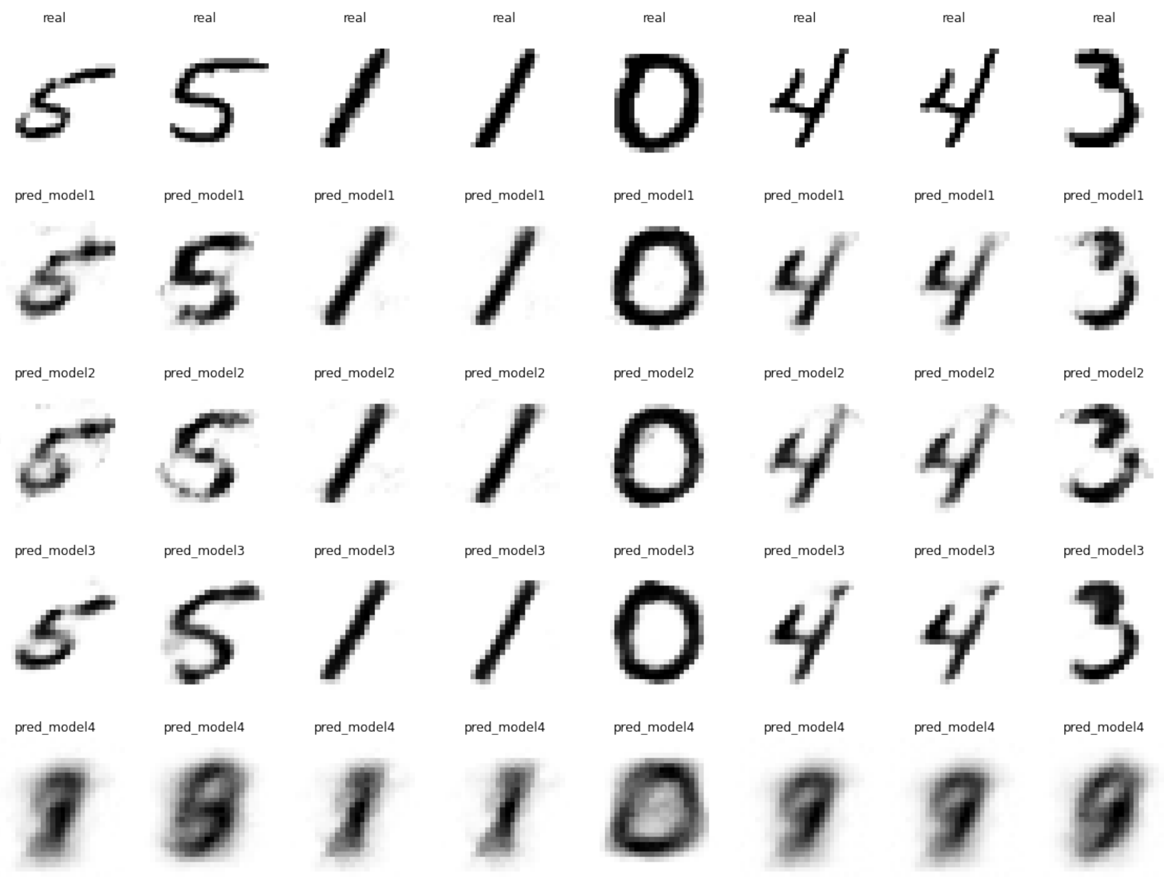


Figure 5: Comparison between the decompressed results for the proposed AutoEncoder models using the MNIST dataset

5.3 Discussion on the proposed models and variants

- In any model, the most important activation functions are the ones located at the output. Therefore, it is important to standardize the data between 0 and 1 if the Sigmoid output activation function wants to be tested.
- Incorporating GCN-layers makes the model perform a little better. Unfortunately, doing this significantly increases the number of parameters [7] and the execution ends up being considerably slower.
- The results of Model 4 were so bad in both datasets, that even the best variant for Model 4 performed worse than any other Model variant. Therefore, Model 4 will not be evaluated on the WaveSuite dataset.

5.4 Analysis on hyperparameters

Given the limited computational and time resources, it is not possible to perform a proper hyperparameter optimization [8]. To compensate, a similar but simpler analysis will be carried on instead. For this, a default set of parameters was established and then one parameter will be varied to check its importance. The default set of parameters is as follows:

[7]As mentioned previously, the number of parameters in the input layer of the AutoEncoder module ends up being multiplied by the number of channels in the GCN-layer that connects to it; and this number must be at least 2.

[8]Performing hyperparameter optimization in pytorch requires a package called Ray Tune, and the fact that the current models are based on torch-geometric instead of regular torch would add more difficulty to this project.

- Latent space dimension $= 32$
- Batch size $= 32$
- Number of channels in the GCN-layers that connect to the AutoEncoder module (Here called Middle Channels for simplicity) $= 2$

The results are presented in Table 3 and the following conclusions can be drawn to design the optimal model for the WaveSuite dataset:

- **Latent space dimension:** This parameter is really important no matter the model used. In the two small datasets, it was observed that there is a limit were it is not possible to reduce the latent space dimension without compromising the quality of the decompressed results. For this reason, it is not possible to extrapolate this value by looking at smaller datasets, and this hyperparameter must be tuned on its own for the WaveSuite dataset.
- **Embedding sequence of the GCN-layers:** Given the already discussed importance of a low number of Middle channels, and given that it has to be at least 2 to improve the basic design of the classic AutoEncoder; for the WaveSuite dataset this sequence will be set to: $[8, 16, 2]$ in the Encoder of the models.
- **Batch size:** It was observed that smaller batch sizes converge faster to a lower MSE error. However, after enough iterations, the MSE results tend to the same value despite of the batch size used. Therefore, tuning this parameter is not very important in order to get a low MSE error.

  This parameter is important however, when the time and computational resources are considered, since smaller batches take less computational resources but also longer times to run. To be able to run the models on the WaveSuite dataset, a batch size of 8 was be selected for this project.

6 Results for the WaveSuite dataset

6.1 First design choices

To design the final AutoEncoder, the first things to choose are the activation function at the output of the model and the latent space dimension. To make these decisions, only the classical AutoEncoder will be used. The activation functions tested were: None, RELU and Sigmoid; and the latent space dimensions tested were: 16, 32, 64, 128, 256, 512.

The models that did not use any activation function at the output were the ones that performed the best. On the contrary, the other models seemed to output a constant image no matter what was introduced. Therefore, only the models with no activation function at the output were considered and their results are shown in Figure 6, using different latent space dimensions. A bigger latent space dimension produces better results but requires a bigger number of parameters and since a latent space dimension of 256 points is enough to represent the data, this value will be chosen instead of 512.

Having taken these design choices for Model_1, Model_2 and Model_3 were also implemented using 256 points as the latent space dimension and using no output activation function at the output. All models were trained using 10 epochs, a batch size of 8; and $[8, 16, 2]$ as the embedding sequence of the GCN-layers for the Encoder. The MSE error results were as follows:

- **Model 1:** 0.001160
- **Model 2:** 0.00278
- **Model 3:** 0.000586

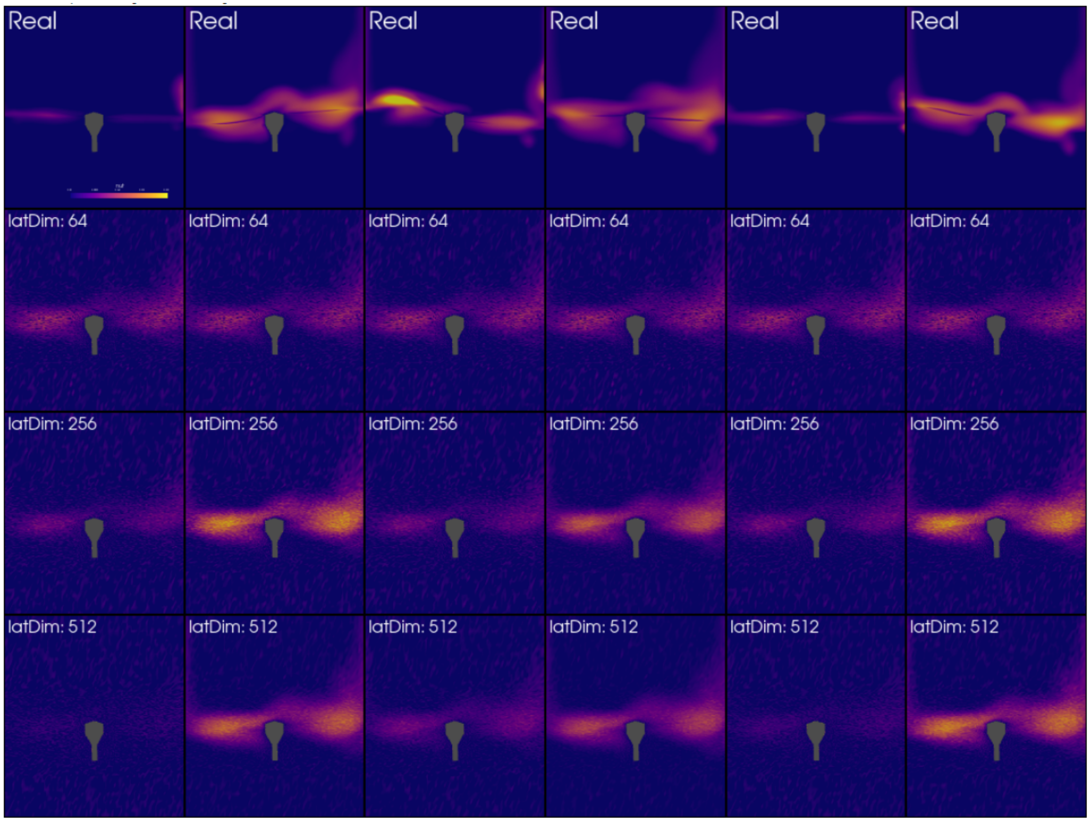


Figure 6: Comparison of the decompressed results for different latent space dimensions, when using a classic AutoEncoder with no output activation function

6.2 Discussion

For the WaveSuite dataset, Model_1 produced better results than Model 2. Given the large amount of nodes in the dataset and that the GCN-layers increase the number of parameters of the model [9], overfitting is a very likely cause. Regularization techniques such as dropout could be implemented but those are out of the scope of this project.

Model_3 produces even better results than Model_1 but given that it took 30 times the same time to execute [10], Model_1 was selected as the final AutoEncoder.

This final model was trained again using 50 epochs, getting 0.000798 as MSE loss [11]. The results are shown in Figure 7.

7 Conclusions

This project identified some key hyperparameters to tune when designing an AutoEncoder. It was determined that in general, using the ADAM optimizer generally produces better results for AutoEncoders than the SGD optimizer. It was also determined that the number of points in the latent space dimension is an important variable that should be tuned. Also, two major conclusions can be drawn from the two methods studied in this project:

The activation function used at the output of the AutoEncoder is another variable that heavily

[9]And in the WaveSuite dataset, the number of nodes and therefore parameters is already much larger than the number of samples

[10]Model 1 took 413 seconds to train while Model 3 took 12345 seconds to train

[11]This final model took just 2008 seconds to train

influences the results. The Sigmoid was the best choice for the air-pollution dataset but None was the best choice for the other datasets. This difference could be explained by the fact that the derivative of the sigmoid function is a normal distribution; and since the air-pollution dataset distribution is similar it was easier for the models that used sigmoid to adapt to this dataset.

In general, adding GCN-layers to an AutoEncoder improved the results but hugely compromised on number of model-parameters and training time. Two alternative more common approaches were implemented such as: adding an extra hidden layer that doubled the current hidden one in size; or just doubling the number of epochs in the classic AutoEncoder. Both approaches achieved good results in a significantly lower training time. Therefore, adding GCN-layers to an AutoEncoder isn't worth it, at least in the way it was done in this project.

As a future project, the current AutoEncoder design project could be expanded into a Variational AutoEncoder, and this would allow the user to produce new artificial WaveSuite samples.

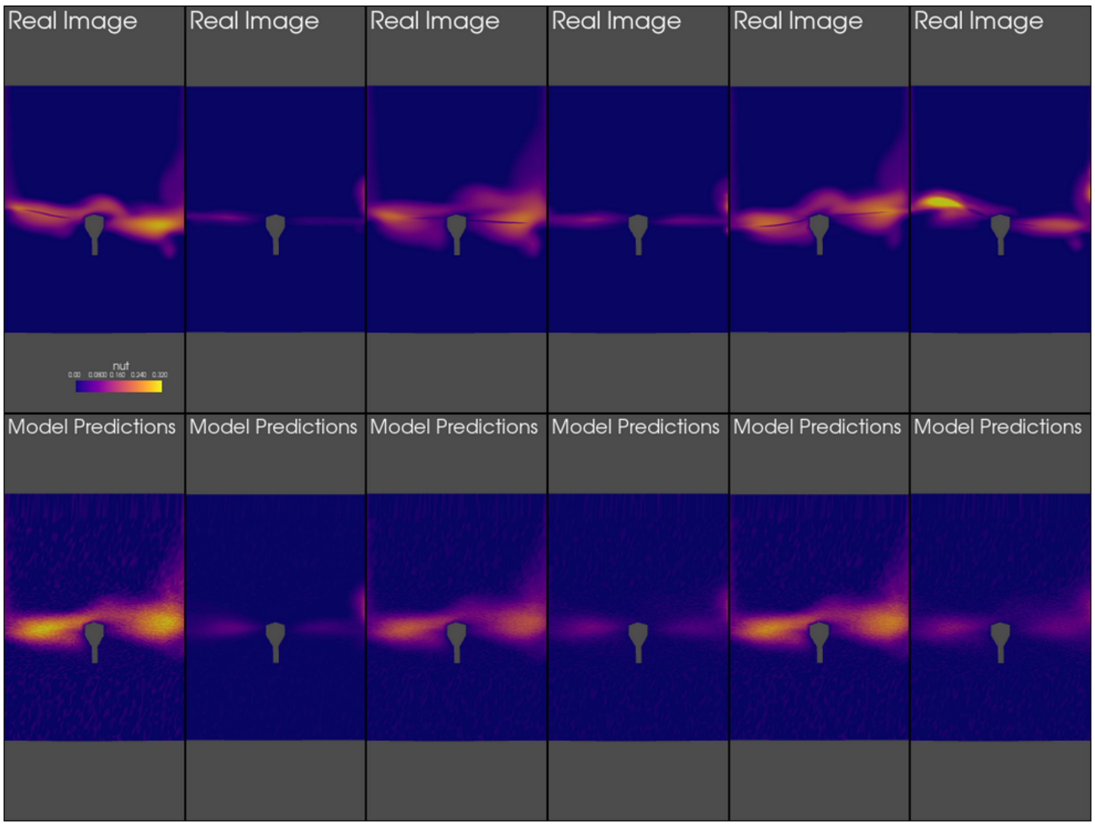


Figure 7: Decompressed results of the final AutoEncoder, a classic AutoEncoder with no output activation function and 256 latent space dimension

Table 1: Comparison of the MSE losses on the air pollution dataset for different models and variants usig 50 epochs

	Model name	Sequence of activation functions	Validation loss	Ranking	training_time
0	Model1	[Sigmoid]	0.0014	3	10.66
1	Model1	[None]	0.0022	7	16.35
2	Model1	[RELU]	0.0429	30	10.7
3	Model2	[None Sigmoid]	0.0008	1	13.97
4	Model2	[RELU Sigmoid]	0.0011	2	13.76
5	Model2	[None None]	0.0021	6	22.2
6	Model2	[RELU None]	0.0056	10	13.33
7	Model2	[None RELU]	0.0395	29	13.4
8	Model2	[RELU RELU]	0.0538	31	13.72
9	Model3	[None None Sigmoid]	0.0017	4	15.8
10	Model3	[None RELU Sigmoid]	0.0049	8	15.67
11	Model3	[None None None]	0.0089	11	17.65
12	Model3	[None None RELU]	0.0094	12	15.22
13	Model3	[None Sigmoid None]	0.0114	13	15.4
14	Model3	[None RELU None]	0.0142	18	15.42
15	Model3	[None RELU RELU]	0.0155	19	15.16
16	Model3	[None Sigmoid RELU]	0.279	36.5	15.88
17	Model3	[RELU None Sigmoid]	0.0018	5	15.73
18	Model3	[RELU RELU Sigmoid]	0.0053	9	15.77
19	Model3	[RELU None RELU]	0.0116	14	15.28
20	Model3	[RELU None None]	0.0119	15	17.64
21	Model3	[RELU Sigmoid None]	0.0126	16	15.39
22	Model3	[RELU RELU RELU]	0.0128	17	15.48
23	Model3	[RELU RELU None]	0.0178	27.5	15.49
24	Model3	[RELU Sigmoid RELU]	0.279	36.5	15.97
25	Model4	[None RELU Sigmoid]	0.0157	20	19.76
26	Model4	[None None Sigmoid]	0.0161	21.5	19.52
27	Model4	[None RELU None]	0.017	24	19.39
28	Model4	[None None None]	0.0178	27.5	21.82
29	Model4	[None RELU RELU]	0.0911	32	19.64
30	Model4	[None None RELU]	0.0951	35	19.5
31	Model4	[RELU RELU Sigmoid]	0.0161	21.5	19.95
32	Model4	[RELU None Sigmoid]	0.0162	23	19.85
33	Model4	[RELU None None]	0.0173	25.5	22.15
34	Model4	[RELU RELU None]	0.0173	25.5	19.6
35	Model4	[RELU None RELU]	0.0912	33	19.74
36	Model4	[RELU RELU RELU]	0.0916	34	19.78

Table 2: Comparison of the MSE losses on the MNIST dataset for different models and variants using 80 epochs

	Model name	**Sequence of activation functions**	**Validation loss**	**Ranking**	**training_time**
0	Model1	[None]	0.0177	5	27.19
1	Model1	[Sigmoid]	0.0208	13	26.09
2	Model1	[RELU]	0.0229	17	26.17
3	Model2	[None None]	0.0184	8	35.45
4	Model2	[RELU None]	0.0188	9	33.3
5	Model2	[None Sigmoid]	0.0211	14.5	33.08
6	Model2	[RELU RELU]	0.0211	14.5	33.36
7	Model2	[None RELU]	0.0224	16	33.24
8	Model2	[RELU Sigmoid]	0.0248	19	33.29
9	Model3	[None None RELU]	0.0164	1	44.18
10	Model3	[None None None]	0.018	6.5	46.06
11	Model3	[None RELU RELU]	0.018	6.5	44.3
12	Model3	[None None Sigmoid]	0.0192	10	44.08
13	Model3	[None RELU None]	0.0193	11	44.27
14	Model3	[None Sigmoid RELU]	0.0253	20	44.03
15	Model3	[None Sigmoid None]	0.0259	21	44.08
16	Model3	[None RELU Sigmoid]	0.0318	24	44.2
17	Model3	[RELU None RELU]	0.0172	2	43.48
18	Model3	[RELU RELU RELU]	0.0174	3.5	44.01
19	Model3	[RELU None None]	0.0174	3.5	45.66
20	Model3	[RELU RELU None]	0.0196	12	44
21	Model3	[RELU None Sigmoid]	0.0231	18	43.7
22	Model3	[RELU Sigmoid RELU]	0.0269	22	43.81
23	Model3	[RELU Sigmoid None]	0.0291	23	43.91
24	Model3	[RELU RELU Sigmoid]	0.0334	25	44.02
25	Model4	[None RELU None]	0.0549	26	64.47
26	Model4	[None None None]	0.0554	29	65.11
27	Model4	[None None Sigmoid]	0.0557	30.5	62.51
28	Model4	[None RELU Sigmoid]	0.0557	30.5	64.81
29	Model4	[None None RELU]	0.0611	34	62.75
30	Model4	[None RELU RELU]	0.0621	36	66.19
31	Model4	[RELU None None]	0.0551	27	65.81
32	Model4	[RELU RELU None]	0.0552	28	64.4
33	Model4	[RELU RELU Sigmoid]	0.0566	32	64.51
34	Model4	[RELU None Sigmoid]	0.0578	33	63.31
35	Model4	[RELU None RELU]	0.0618	35	63.63
36	Model4	[RELU RELU RELU]	0.0623	37	66.05

Table 3: Comparison of the MSE losses when one variable is tuned while the others remain fixed

Tuning parameter	**Dataset**	**Model name**	**Parameter value**	**Validation loss**
LatDim	air pollution	classic_AE	latDim64	0.0004
LatDim	air pollution	classic_AE	latDim32	0.0011
LatDim	air pollution	classic_AE	latDim16	0.0048
LatDim	air pollution	classic_AE	latDim8	0.0106
LatDim	air pollution	classic_AE	latDim4	0.0134
LatDim	air pollution	classic_AE	latDim2	0.016
LatDim	MNIST	GCN_AE_GCN	latDim32	0.0192
LatDim	MNIST	GCN_AE_GCN	latDim64	0.0197
LatDim	MNIST	GCN_AE_GCN	latDim16	0.0257
LatDim	MNIST	GCN_AE_GCN	latDim8	0.0321
LatDim	MNIST	GCN_AE_GCN	latDim4	0.0478
LatDim	MNIST	GCN_AE_GCN	latDim2	0.0514
EmbSeq	air pollution	GCN_AE	8MiddleChannels	0.0008
EmbSeq	air pollution	GCN_AE	2MiddleChannels	0.0009
EmbSeq	air pollution	GCN_AE	4MiddleChannels	0.0009
EmbSeq	air pollution	classic_AE	-	0.0011
EmbSeq	air pollution	GCN_AE	1MiddleChannel	0.0011
EmbSeq	MNIST	GCN_AE_GCN	2MiddleChannels	0.0194
EmbSeq	MNIST	GCN_AE_GCN	1MiddleChannels	0.0199
EmbSeq	MNIST	GCN_AE_GCN	4MiddleChannels	0.0204
EmbSeq	MNIST	GCN_AE_GCN	8MiddleChannels	0.0204
EmbSeq	MNIST	classic_AE	-	0.0211
BatchSize	air pollution	classic_AE	batchSize2	0.0003
BatchSize	air pollution	classic_AE	batchSize4	0.0003
BatchSize	air pollution	classic_AE	batchSize8	0.0004
BatchSize	air pollution	classic_AE	batchSize16	0.0009
BatchSize	air pollution	classic_AE	batchSize32	0.0011
BatchSize	air pollution	classic_AE	batchSize64	0.0019
BatchSize	MNIST	GCN_AE_GCN	batchSize16	0.0196
BatchSize	MNIST	GCN_AE_GCN	batchSize8	0.0197
BatchSize	MNIST	GCN_AE_GCN	batchSize32	0.0201
BatchSize	MNIST	GCN_AE_GCN	batchSize2	0.0205
BatchSize	MNIST	GCN_AE_GCN	batchSize4	0.0206
BatchSize	MNIST	GCN_AE_GCN	batchSize64	0.0218

References

[AL18] Tunde Aderinto and Hua Li. Ocean wave energy converters: Status and challenges. *Energies*, 11(5):1250, 2018.

[BKG20] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *CoRR*, abs/2003.05991, 2020.

[CDH+10] Lachlan Cameron, R Doherty, Alan Henry, Kenneth Doherty, Jos van 't Hoff, D Kaye, D Naylor, Sylvain Bourdier, and Trevor Whittaker. Design of the next generation of the oyster wave energy converter. 10 2010.

[DPS09] B Drew, A R Plummer, and M N Sahinkaya. A review of wave energy converter technology. *Proceedings of the Institution of Mechanical Engineers, Part A: Journal of Power and Energy*, 223(8):887–902, 2009.

[GK20] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, abs/2009.07485, 2020.

[GSR+17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. *CoRR*, abs/1704.01212, 2017.

[Ham] William L. Hamilton. Graph representation learning. chapter 5. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[JPG18] Siya Jin, Ron Patton, and Bingyong Guo. Viscosity effect on a point absorber wave energy converter hydrodynamics validated by simulation and experiment. *Renewable Energy*, 129, 06 2018.

[KKAH17] N. Khan, A. Kalair, N. Abas, and A. Haider. Review of ocean tidal, wave and thermal energy technologies. *Renewable and Sustainable Energy Reviews*, 72:590–604, 2017.

[KW16a] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[KW16b] Thomas N. Kipf and Max Welling. Variational graph auto-encoders, 2016.

[Les21a] Jure Leskovec. Cs224w: Machine learning with graphs // graph neural networks 2: Design space. University Lecture, 2021.

[Les21b] Jure Leskovec. Cs224w: Machine learning with graphs. // introduction; machine learning for graphs. University Lecture, 2021.

[Mic22] Umberto Michelucci. An introduction to autoencoders. *CoRR*, abs/2201.03898, 2022.

[MJK20] Diego P. P. Mesquita, Amauri H. Souza Jr., and Samuel Kaski. Rethinking pooling in graph neural networks. *CoRR*, abs/2010.11418, 2020.

[Nav19] T.M. Navamani. Chapter 7 - efficient deep learning approaches for health informatics. In Arun Kumar Sangaiah, editor, *Deep Learning and Parallel Computing Environment for Bioengineering Systems*, pages 123–137. Academic Press, 2019.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[PRSS+22] Dominika Przewlocka-Rus, Syed Shakib Sarwar, H. Ekin Sumbul, Yuecheng Li, and Barbara De Salvo. Power-of-two quantization for low bitwidth and hardware compliant neural networks, 2022.

[QLW21] Shenghao Qiu, You Liang, and Zheng Wang. Optimizing sparse matrix multiplications for graph neural networks. *CoRR*, abs/2111.00352, 2021.

[RNS20] Davide Rigoni, Nicolò Navarin, and Alessandro Sperduti. Conditional constrained graph variational autoencoders for molecule design. *CoRR*, abs/2009.00725, 2020.

[Ste] John Stevenson. Powering a sustainable future through wave energy.

[TK93] C.Z. Tang and H.K. Kwan. Multilayer feedforward neural networks with single powers-of-two weights. *IEEE Transactions on Signal Processing*, 41(8):2724–2727, 1993.

[TYL22] Mingyue Tang, Carl Yang, and Pan Li. Graph auto-encoder via neighborhood wasserstein reconstruction, 2022.

[VCC$^+$17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[WHWW14] Wei Wang, Yan Huang, Yizhou Wang, and Liang Wang. Generalized autoencoder: A neural network framework for dimensionality reduction. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 496–503, 2014.

[WRHD15] Yanji Wei, Ashkan Rafiee, Alan Henry, and Frederic Dias. Wave interaction with an oscillating wave surge converter, part i: Viscous effects. *Ocean Engineering*, 104:185–203, 2015.

[ZMW$^+$19] Haitao Zhang, Lingguo Meng, Xian Wei, Xiaoliang Tang, Xuan Tang, Xingping Wang, Bo Jin, and Wei Yao. 1d-convolutional capsule network for hyperspectral image classification. 03 2019.