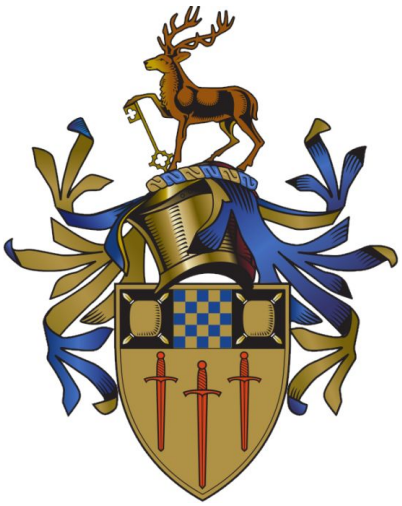
Application of Machine Learning Methods in Incompressible Fluid Flow Prediction

by

Abdel-Rahim Abdalla

Aerospace Engineering (MEng)



Department of Mechanical Engineering Sciences

Faculty of Engineering and Physical Sciences

University of Surrey

Project Report

17th August 2021

Project Supervisor: Professor Nicholas Hills

Personal Statement of Originality

I confirm that the submitted work is my own work. No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have also clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook. I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin®UK. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Abdel-Rahim Abdalla

Abstract

Fluid flow simulation is an invaluable tool for many engineering disciplines. However, the Navier-Stokes equations that such tools are built upon are regarded to be notoriously difficult to solve, so novel techniques are required to produce accurate simulations. This work moves away from the traditional numerical methods applied to this problem and instead considers the use of Machine Learning methods. Modern Machine Learning is still in its infancy compared to numerical methods, especially for engineering problems. Three Machine Learning implementations for fluid flow predictions are studied. These implementations build upon the Reynolds Averaged Navier-Stokes model and Smoothed Particle Hydrodynamics, respectively, by integrating Neural Networks into the process to reduce the computational costs of the mentioned flow simulation models. Two of the implementations leverage Convolutional Neural Networks, while the final implementation use Multilayer Perceptrons in tandem with a framework to assist in applying Machine Learning to graph-structured data, a data type Neural Networks have historically struggled in predicting. Following this, a Machine Learning simulator is implemented to predict fluid flows in meshed domains. As meshes are inherently graph-structured, the Graph Network framework introduced previously is used. The mesh-based Machine Learning simulator is trained on a dataset of Reynolds Averaged Navier-Stokes simulations of a jet impingement system generated by Computational Fluid Dynamics. The trained mesh-based simulator is tested using a mesh and inlet velocity featured in the training dataset, a mesh with an inlet velocity not featured in the dataset, and finally, a mesh never seen before by the simulator. The simulator was shown to be accurate in small time-steps but lost accuracy as a simulation was rolled out. The simulator learned the locations of inlets and boundaries, applying the correct physics of introducing fluid into the domain and developing a boundary layer. The computation time for the simulator was found to be roughly half that of the Computational Fluid Dynamics simulator for the same mesh and inlet velocity.

Acknowledgements

I want to start by thanking the University of Surrey for the education and experiences I have been able to enjoy during my four years here. From studying the degree I have dreamed of since I was a little child to giving me new passions and hobbies.

Secondly, I would like to thank my supervisor, Professor Nick Hills, for all his hard work helping me navigate this project and my tutor, Dr Matteo Carpentieri, for guiding me through University.

Thirdly, I would like to thank all my friends who have been with me through thick and thin, especially during this difficult period.

Finally, I would like to thank my family, as, without their support, I would not be where I am today.

List of Acronyms

N-S	Navier-Stokes
CFD	Computational Fluid Dynamics
ML	Machine Learning
NN	Neural Network
CNN	Convolutional Neural Network
RANS	Reynolds Averaged Navier-Stokes
GPU	Graphics Processing Unit/ Graphics Card
APE	Absolute Percentage Error
SPH	Smoothed Particle Hydrodynamics
GN	Graph Network
MPNN	Message-Passing Neural Network
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
MSE	Mean Squared Error

Contents

1 Introduction

1.1 Motivation

Simulation tools are the backbone of engineering projects, providing engineers with insight into their designs that would either take too long or be impossible to work out by hand. Many products would not exist in their current form without using simulation tools during the design process. More complex and refined designs can be produced by incorporating our ever-increasing understanding of the physical laws and relationships into simulation tools. In addition, different simulation tools can be connected to give engineers a more complete overview of the interactions in their designs.

The strive towards perfecting simulation tools comes with a significant downside, however. Some physical laws engineers are interested in simulating have no known solutions, and as such, simulation tools must calculate approximations for such problems. Due to the complex nature of these problems, the approximation process is very computationally taxing and require significant periods of time and expensive hardware. One such example is the Navier-Stokes (N-S) equations (equation 1, Cengel (2014)). The N-S equations represent Newtonian fluid flow by considering the conservation of mass and momentum.

$$\begin{aligned} \nabla \boldsymbol{u} &= 0 \\ \rho \frac{D\boldsymbol{u}}{Dt} &= -\nabla p + \mu \nabla^2 \boldsymbol{u} + \boldsymbol{F} \end{aligned} \tag{1}$$

Computational Fluid Dynamics (CFD) is a technique to solve the N-S equations numerically by applying approximations and boundary conditions. Typical CFD software uses finite volume discretization to break up a flow into control volumes represented as cells in a mesh, then calculating the conservation equations across each control volume using interpolation. The results from a CFD simulation can be improved by using finer meshes with a higher number of control volumes and using higher-order interpolation schemes, among other techniques. However, this increases the computation time making detailed CFD simulations more computationally intensive (Marxen (2019)). Running CFD at scale is also monetarily cost-prohibitive, requiring dedicated servers and staff.

Over the past decade, a strong interest in Machine Learning (ML) has appeared in research and commercial circles. Many software problems cannot be solved using traditional programming, namely pattern and language recognition (Imperial College London (2018)). The underlying concept behind ML is to allow a computer to complete a task without explicitly being programmed for such a task (Alpaydin (2014)). This directly conflicts with traditional programming, where a computer executes exactly what it is programmed to do. ML opens the possibility to solve problems without requiring a known algorithm for the given task.

Comparisons can be made between solving the N-S equations and traditional ML problems such as pattern recognition. Both the N-S equations and pattern recognition cannot be solved analytically with our current understanding of the problems, but informal guidelines can be placed to help arrive at solutions.

1.2 Objectives

This work aims to investigate the feasibility of implementing a ML approach to solving the N-S equations. This will be conducted by considering a range of ML techniques for predicting and simulating physical systems, comparing their implementations and performance. Next, a ML method is to be selected and implemented to create a fluid flow simulator. Finally, the performance and accuracy of the designed simulator will be compared against a mature CFD software suite. Due to the nature of implementing ML models, a fully trained simulator may not be possible in the time available to complete this work. The target, therefore, is to produce a proof of concept that shows that further development is possible.

2 Background Literature

2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a ML algorithm that has been heavily researched in the past decade, specifically in computer vision and pattern detection. CNNs aim to replicate the connectivity and operation of neurons in a human brain. A CNN is formed of layers of neurons. Neurons take multiple weighted inputs, x with weight w, and produce a single output with the inputs and output values being a number between 0 and 1. An activation function, $\sigma(x)$, dictates the output of a neuron. A typical activation function is the sigmoid function (equation 2). Each neuron is also assigned a bias, b, to represent its output's importance in the network (Nielsen (2015)).

$$z = \sum_j w_j x_j + b$$

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{2}$$

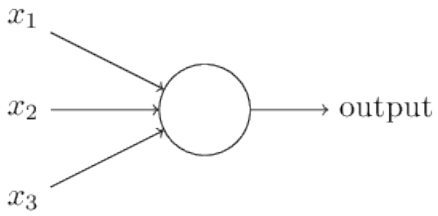


Figure 1: Graphical representation of a CNN neuron (Taken from Nielsen (2015))

Any CNN comprises at least three layers: an input layer, an output layer, and at least a single hidden layer. Hidden layers typically take one of three forms: convolution layer that acts as a filter for specific features; a pooling layer that reduces the number of neurons which in turn reduces overfitting and speeds up the training process of the CNN; or a dense layer that connects all inputs to the layer's neurons which handles the classification (Saha (2018)). The number of hidden layers and their form is chosen based on the network's function. An example of a CNN's layout can be seen in figure 2.

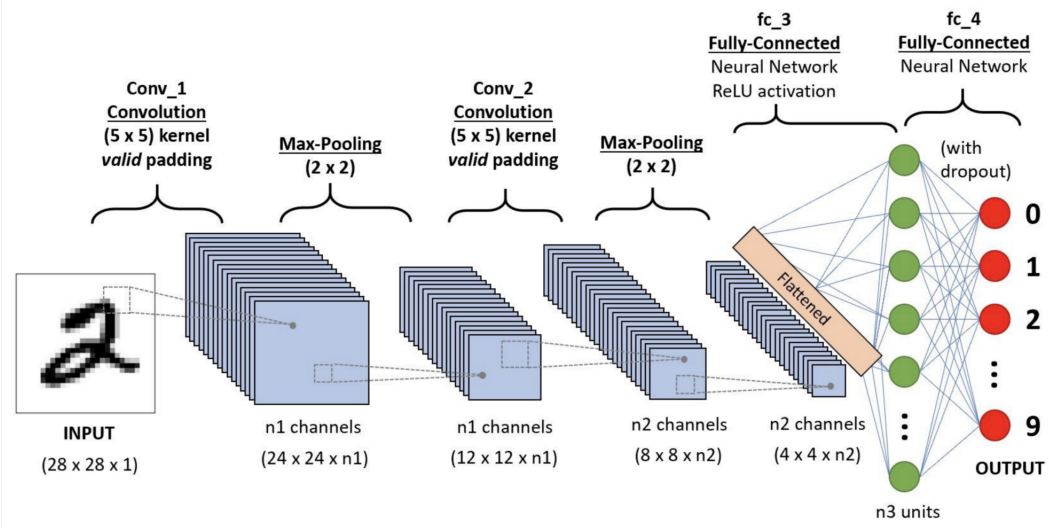


Figure 2: Example CNN for handwritten digit classification (Taken from Saha (2018))

CNNs are trained by tuning the weights and biases until the network's outputs are accurate. This is typically achieved using supervised learning, where data is fed into the network and tuned based on the difference between the real data and the output data. A loss function is used to represent the error in the output data. Reducing the value of the loss function is achieved using stochastic gradient descent to calculate the required changes in the weights by iteratively searching for the minima of the loss function (Nielsen (2015)). A stochastic gradient descent optimiser commonly used in modern NNs is the Adam optimiser which allows for individual learning rates for different parameters, improving convergence speeds (Kingma & Ba (2017)).

2.1.1 Bhatnagar et al. (2019)

Bhatnagar et al. (2019) proposed a model to predict non-uniform steady Reynolds Averaged Navier-Stokes (RANS) flow in a domain using a CNN. RANS differs from the conventional N-S equations as it takes into account for turbulence by decomposing velocities into mean and fluctuating parts (Birch (2020)). This model can predict the flow field around different geometries under variable flow conditions. The CNN's role is to identify key flow features in the domain based on an aerofoil shape and free stream conditions. This model uses a form of CNN called an Encoder-Decoder CNN, shown in figure 3. The network consists of three convolutional layers, followed by a dense layer, then finally another three convolutional layers. Each set of three convolutional layers form an encoder and decoder, respectively. The encoder takes input data and extracts key domain features, which are passed to the decoder through the dense layer to generate the flow field. Bhatnagar et al. (2019) builds on research conducted by Guo et al. (2016), which proposed using separate decoders for the velocity components and the pressure field. This was changed to a shared decoder in Bhatnagar et al. (2019) to half the number of training parameters – improving computational performance while maintaining accuracy. The model takes a mesh and flow characteristics as the input and pre-processes it to find the distance of each node to the target object.

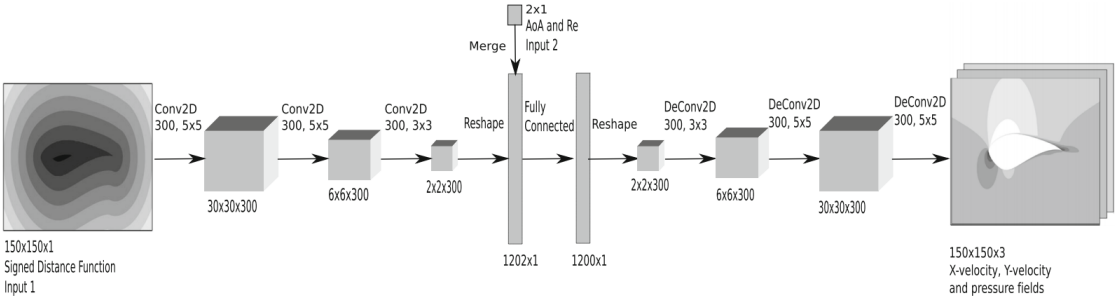


Figure 3: Graphical representation of the encoder-decoder neural network (Taken from Bhatnagar et al. (2019))

A dataset made up of 252 2-dimensional RANS simulations using three aerofoils was generated to train the model. The dataset was split such that 85% of the simulations were used as training data and the remainder as test data. The test data was selected randomly to ensure the model testing is unbiased. The training process took 33 hours to run using a high-end consumer graphics card (GPU), and 30,000 passes of the training data were conducted. Once the model is trained, predicting new flows was four orders of magnitude faster than performing a RANS simulation of the same flow.

The mean value of the absolute percentage error (APE), as seen in equation 3, was used as the metric to validate the model against the ground truth simulations. It was found that the error in the flow was between 5% to 13% (table 2) when compared against the training data simulations, while the error in the wake region ranged from 5% to 31% (table 1). Similar error ranges were seen in the validation of the Guo et al. (2016) model. However, the Guo et al. (2016) model errors were in general lower than that for Bhatnagar et al. (2019). This can be attributed to the fewer training parameters.

$$APE = \frac{|Prediction - Truth|}{|Truth|} \times 100 \tag{3}$$

Table 1: APE in wake region across the tested aerofoils (Taken from Bhatnagar et al. (2019))

Aerofoil	Angle of Attack (°)	$Re \times 10^6$	Variable	Error in wake region (%)	
				Bhatnagar et al. (2019)	Guo et al. (2016)
S809	1	1	x-velocity	12.25	11.43
S809	1	1	y-velocity	24.27	15.53
S809	1	1	pressure	5.14	5.76
S814	19	3	x-velocity	30.80	27.23
S814	19	3	y-velocity	10.43	5.57
S814	19	3	pressure	13.84	12.93

Table 2: APE in the entire flow across the tested aerofoils (Taken from Bhatnagar et al. (2019))

Aerofoil	Angle of Attack (°)	$Re \times 10^6$	Variable	Error in the entire flow (%)	
				Bhatnagar et al. (2019)	Guo et al. (2016)
S809	1	1	x-velocity	10.35	7.79
S809	1	1	y-velocity	11.52	8.74
S809	1	1	pressure	8.40	7.36
S814	19	3	x-velocity	13.13	13.20
S814	19	3	y-velocity	5.49	4.69
S814	19	3	pressure	5.70	5.71

The pressure distribution around the aerofoil calculated by the Bhatnagar et al. (2019) model also matched the expected distribution from the ground truth simulations (figure 4). The model produced equivalent errors in the flow and wake regions when predicting the flow around a previously unseen aerofoil, validating that the model has learned the key flow features.

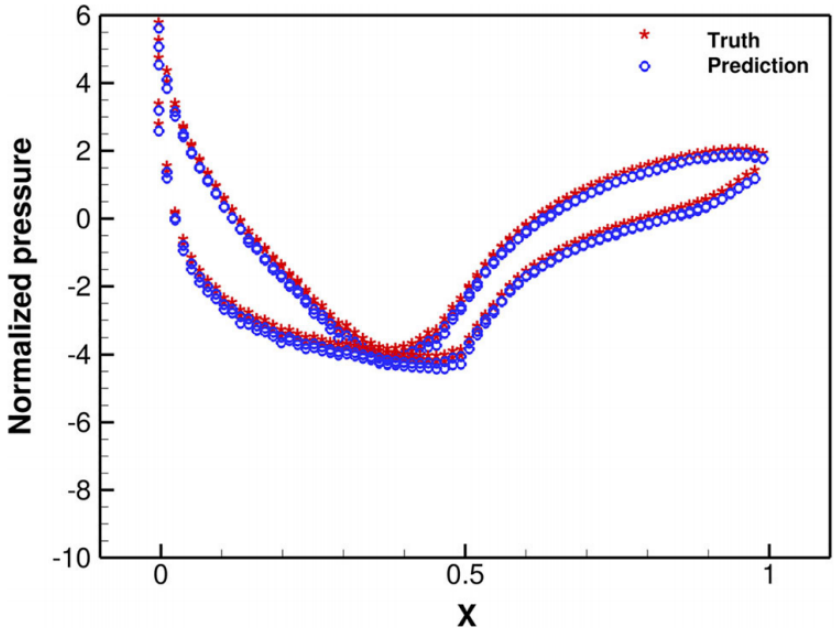


Figure 4: Pressure prediction across a S809 aerofoil at $\alpha = 1°$ and $Re \times 10^6$ (Taken from Bhatnagar et al. (2019))

2.1.2 Ummenhofer et al. (2020)

Ummenhofer et al. (2020) takes an alternative approach to predict fluid flow using CNNs. While Bhatnagar et al. (2019) represents the flow on a mesh, Ummenhofer et al. (2020) is based on

Smoothed Particle Hydrodynamics (SPH) where individual particles and their interactions together are modelled. This technique simplifies the implementation of the model while allowing for the modelling of more complex flows. It is also a versatile technique that can be implemented to model solid mechanics, for example. Fluid simulation SPH is ideal for applications such as animation where accuracy is not the main priority as the number of required particles to model the flow is significantly lower than the number required for the accuracy demanded by engineering applications (Price (2012)). Furthermore, the high computational cost makes it suited for ML prediction as the time savings will have a greater effect on improving the viability of SPH in engineering applications.

The Ummenhofer et al. (2020) model consists of nine hidden layers (figure 5). The initial three layers are at the same depth, two of which are convolution layers. One computes the interactions between each particle and the static environment particles, and the other computes the interactions between each particle and the remaining dynamic particles. The final layer in the first depth is a dense layer that processes particle features such as location and velocity. The following depth features a convolution layer and dense layer connected to the dynamic particle convolution layer and particle feature dense layer. The output of the convolution layer and dense layer are aggregated and passed to another two sets of convolution and dense layers, aggregating between each set. The final output of the network returns a displacement for each particle.

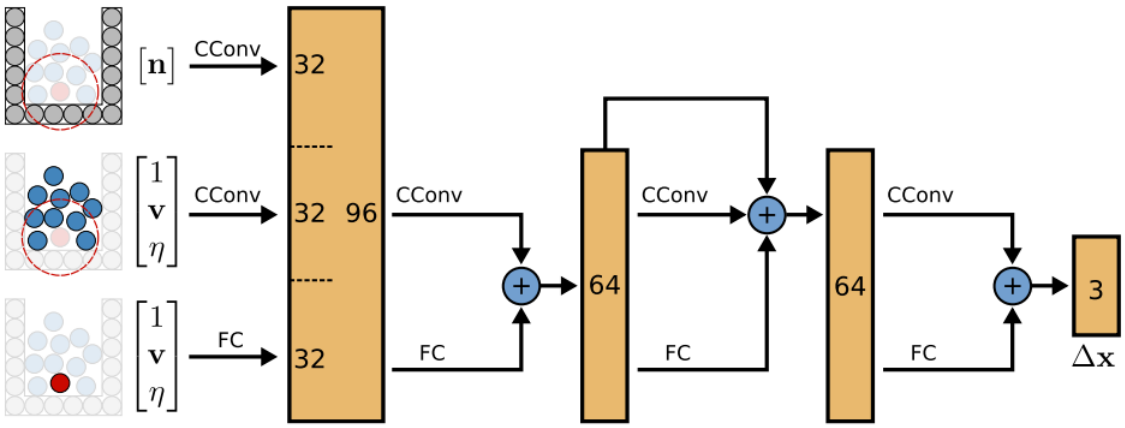


Figure 5: Graphical representation of the implemented CNN (Taken from Ummenhofer et al. (2020))

The model is trained using supervised learning by calculating the loss between the predicted particle positions and the ground-truth position. The loss function emphasises the loss for particles with fewer neighbours, typically closer to surfaces and boundaries. These particles are important as they allow the model to predict the interaction between the fluid and the surrounding environment. The loss function was optimised using the Adam optimiser with a decaying learning rate of over 50,000 iterations. The training duration was about a day using a single high-end consumer GPU. A scene consisting of a randomly positioned fluid block placed in a static box was used for the dataset. The dataset consisted of 2000 generated scenes for training and 300 for testing.

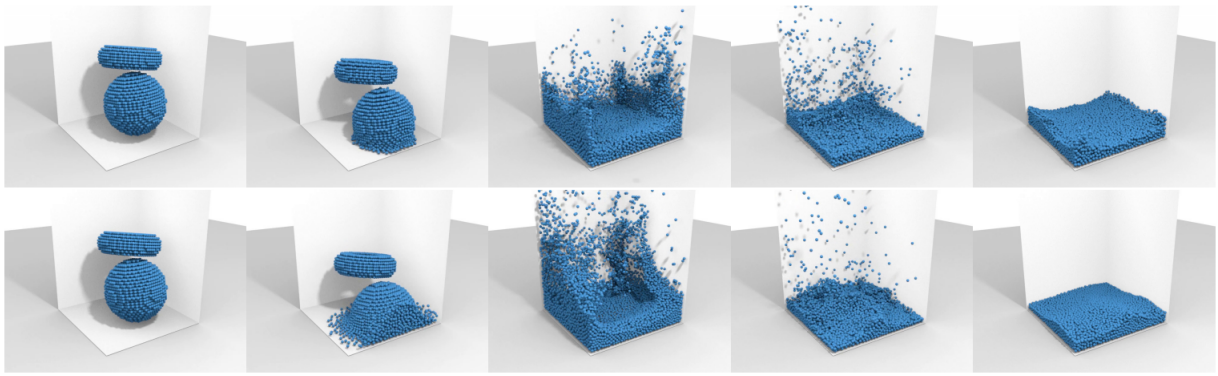


Figure 6: CNN output (top row) compared to the output from the DFSPH simulator (bottom row) for the same scenario (Taken from Ummenhofer et al. (2020))

To test the model's effectiveness, it was compared against the performance of several alternative SPH CNN models (table 3). The metric used was the average error of the predicted positions compared to the ground-truth positions. For the Ummenhofer et al. (2020) model, the average position error after one predicted frame was 0.62mm and 1.49mm after two frames. This was slightly more accurate than the other models who's average errors ranged from 0.71mm to 12.73mm for a single frame and 1.65mm to 25.38mm for two frames. The Ummenhofer et al. (2020) model achieved these results while training for a shorter time and using less computational power.

Table 3: Average error and inference times for the tested models (Taken from Ummenhofer et al. (2020))

Method	Average error (mm)		Frame inference time (ms)
	$n+1$	$n+2$	
DPI-Nets	12.73	25.38	202.56
SPNets Convs	-	-	1058.46
PCNN Convs	0.72	1.67	187.34
Ummenhofer et al. (2020)	0.62	1.49	12.01

A test was also conducted to validate the generalisation of the model. The trained model was given new geometries, drastically different to the training data. In an 800 frame trajectory, the average position error increased in the first 100 frames up to just under 100mm but decreased over the next 150 frames, where the error stabilised at around 30mm. The initial spike in position error was due to the chaotic nature of the particles as they collided with the environment. Finally, the model was tasked to calculate the viscosity of an observed fluid in a scene not present in the training data. It was found that the predicted viscosity had an average relative error of 15% across a range of 7 input viscosities.

2.2 Graph Networks

A graph is a set of node elements and edge elements where the edges represent connections between two nodes (either two different nodes or the same node connected to itself). If the edges have specific directions, the set is called a directed graph. On the other hand, the set is called an undirected graph if the edges only signify the connection between nodes. This can be seen in figure 7. Graphs are an intuitive method to represent physical systems as they allows for the abstraction of data down to the fundamental components. The most famous example of a graph is the London Underground map that allows a user to extract the most important journey information at a glance. Graphs allow for simplification for even the most complex systems, such as the human nervous system and dark matter interaction (Cranmer et al. (2020)). Over the past two decades, especially in the last five years, there has been an increase in the research into the use of NNs that operate on such graphs. This is because networks such as CNNs by nature cannot handle structured data well, so a network specifically designed for such data can allow for more accurate implementations of ML-based prediction.

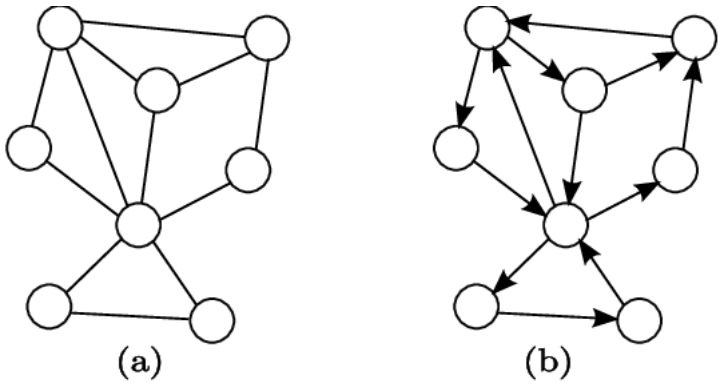


Figure 7: Examples of (a) undirected and (b) directed graphs (Taken from Fionda & Palopoli (2011))

As research into the application of ML using graphs is fairly new, a unified framework does not exist. However, Battaglia et al. (2018) sets out the Graph Networks (GN) framework that defines a set of functions that can be connected to form 'blocks' that can, in turn, be connected to create complex network architectures. In this framework, as shown in figure 8, directed graphs represent the data structure and attributes can be assigned to the nodes, $\boldsymbol{v}_i$, and edges, $\boldsymbol{e}_k$, or even the entire graph as global attributes, $\boldsymbol{u}$.

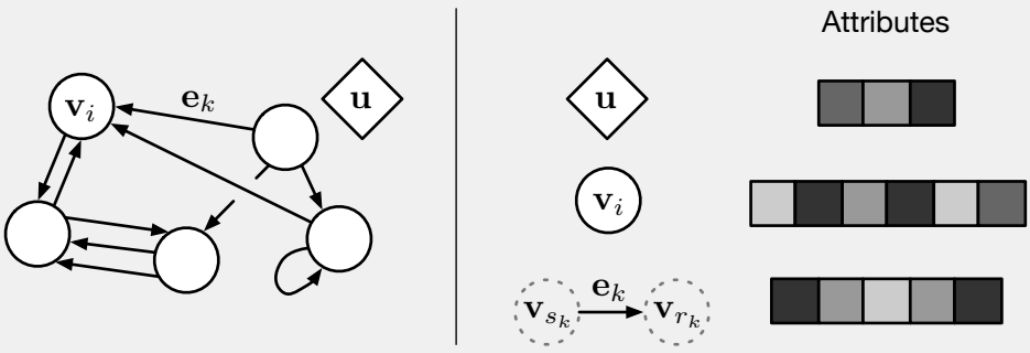


Figure 8: Graphical definition of the GN variables. Note, $\boldsymbol{v}_{sk}$ and $\boldsymbol{v}_{rk}$ represent the sender and receiver nodes respectively. (Taken from Battaglia et al. (2018))

The primary block used in the framework is the GN block (figure 9), which comprises three updater functions, $\phi()$, and three aggregation functions, $\rho()$ - a pair for the edge, node, and global attributes (equation 4). The GN block takes a graph as an input, $[\,\boldsymbol{u}\ V\ E\,]$, and returns an updated graph, $[\,\boldsymbol{u'}\ V'\ E'\,]$. The GN block begins by updating the edge attributes and aggregating the edge attributes per node. Next, the node attributes are updated then both the edge and node attributes are aggregated globally. Finally, the global attributes are updated. The order in which the update functions are executed can be rearranged based on the purpose of the GN block.

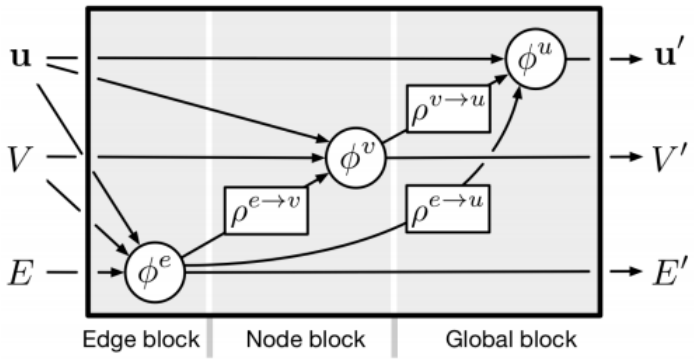


Figure 9: Graphical representation of a GN block (Taken from Battaglia et al. (2018))

$$
\begin{aligned}
\phi^e(\boldsymbol{e}_k, \boldsymbol{v}_{rk}, \boldsymbol{v}_{sk}, \boldsymbol{u}) &= f^e(\boldsymbol{e}_k, \boldsymbol{v}_{rk}, \boldsymbol{v}_{sk}, \boldsymbol{u}) \\
\phi^v(\bar{\boldsymbol{e}}'_k, \boldsymbol{v}_i, \boldsymbol{u}) &= f^v(\bar{\boldsymbol{e}}'_k, \boldsymbol{v}_i, \boldsymbol{u}) \\
\phi^u(\bar{\boldsymbol{e}}'_k, \bar{\boldsymbol{v}}'_i, \boldsymbol{u}) &= f^v(\bar{\boldsymbol{e}}'_k, \bar{\boldsymbol{v}}'_i, \boldsymbol{u})
\end{aligned}
$$

$$
\begin{aligned}
\rho^{e\to v}(E'_i) &= \sum_{k:r_k=i} \boldsymbol{e}'_k \\
\rho^{v\to u}(V') &= \sum_i \boldsymbol{v}'_i \\
\rho^{e\to u}(E') &= \sum_k \boldsymbol{e}'_k
\end{aligned} \tag{4}
$$

For some physical systems, the data encoded into the edges of the graph representation are not impacted by the node interactions. For example, the edges in a CFD mesh show the connections between nodes and do not change as a simulation steps in time. As such, the GN block can be adjusted to represent this by removing the edge output function. Another key block that takes this into account is the Message-Passing Neural Network (MPNN) block. Unlike the GN block, the MPNN block takes the edge and node attributes as inputs and returns node and global attributes as outputs. As the name suggests, the MPNN block is primarily used for message-passing (Gilmer et al. (2017)). This technique is used to propagate information from a node to all connected nodes, as shown in figure 10.

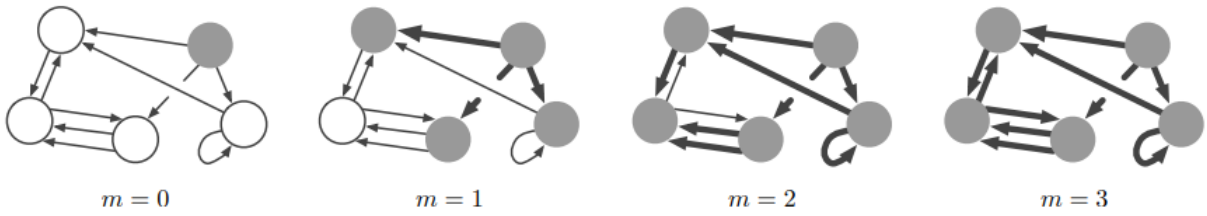


Figure 10: Data propogatting through a graph using three message passing steps, m. (Taken from Battaglia et al. (2018))

As mentioned previously, the framework's power comes from the ability to connect blocks to model a physical system. Connecting multiple MPNN blocks in series allows the model to predict the node interactions over a period of time by propagating the node information a step further through the graph at each block. These connected blocks can be represented as a 'processing core' that can be further connected to an input and output block for an 'encode-process-decode' architecture (figure 11). The 'encoder' input block takes a system, constructs a graph representation, and extracts initial interactions, which are then processed by the core to calculate the network changes. Finally, the 'decoder' output block extracts the resulting data.

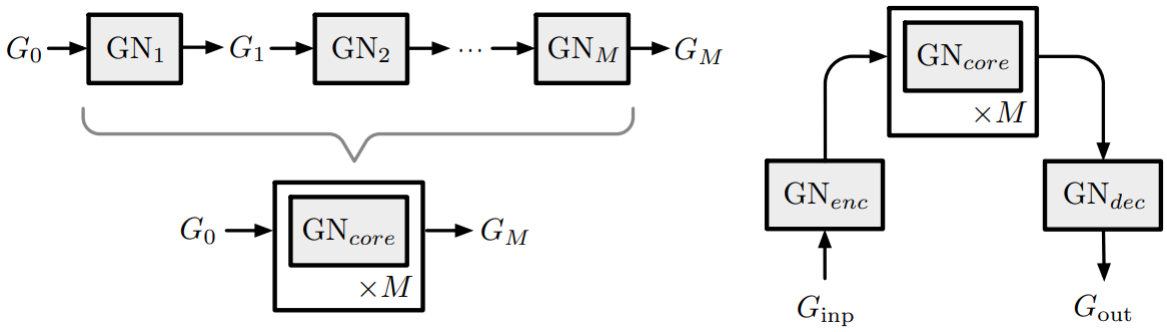


Figure 11: Connected blocks to form an encode-process-decode architecture (Taken from Battaglia et al. (2018))

Returning to the underlying GN block equations, it can be seen in equation 4 that the functions that drive the aggregators were loosely defined. For simple problems, algebraic expressions can be used. However, this may be impossible in more complex, and so NNs are used in their place. The choice of NN depends on the problem, but typically, Multilayer Perceptron (MLP) NNs are selected. As with CNNs discussed in section 2.1, MLPs are formed of layers of neurons and feature at least three layers. However, unlike CNNs, the hidden layers in an MLP are always fully connected, i.e., all neuron outputs are used as inputs for all neurons in the next layer. MLPs are trained using the same techniques discussed previously for CNNs.

2.2.1 Sanchez-Gonzalez et al. (2020)

Sanchez-Gonzalez et al. (2020) proposes a GN simulation model based on the prediction of particle interactions similar to the process of SPH. The model uses a trained 'encode-process-decode' architecture to simulate the particle interactions. The simulation domain is represented as a graph with the particles and domain boundaries making up the nodes.

In the encoding stage, the particle location is encoded into the nodes. Edges connecting are added by identifying nodes within a sphere of a given radius, r, to each node, and the edge embedded data is made up of the displacement between the connected nodes. The value for gravitational acceleration is represented as a global embedding. The node interactions, acceleration, and velocity are then calculated in the processing stage, where the constructed graph goes through a given number of message-passing steps, M. The radius and number of message-passing steps are hyperparameters that are adjusted and optimised in order to improve the training speed and accuracy of the model. The values found to produces the best results were 10 message-passing steps and a connectivity radius of 0.015. Finally, the dynamics information is extracted from the processed graph in the decode stage, and the future particle displacements, $\boldsymbol{p}^{t_{k+1}}$, are calculated using an semi-implicit Euler integrator (equation 5).

$$\begin{aligned} \dot{\boldsymbol{p}}^{t_{k+1}} &= \dot{\boldsymbol{p}}^{t_k} + \Delta t \cdot \ddot{\boldsymbol{p}}^{t_k} \\ \boldsymbol{p}^{t_{k+1}} &= \boldsymbol{p}^{t_k} + \Delta t \cdot \dot{\boldsymbol{p}}^{t_k} \end{aligned} \tag{5}$$

This process can be conducted once to simulate one timestep or repeated to simulate a complete trajectory of the fluid. With each step, the encoder recalculates the node connections to take into account the movement of the particles.

The processor comprises of an M number GN blocks, with the aggregator functions of the blocks being made up of a two-hidden-layer MLP. The neurons in the MLP use Rectified Linear Unit (ReLU) activation functions (equation 6) instead of the traditional sigmoid function (equation 2) as recent studies have found ReLU computationally cheaper to use and optimises faster due to its linear nature (Glorot et al. (2011)).

$$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases} \tag{6}$$

The MLPs are trained using supervised learning, with the loss function being the mean squared error (MSE) of the target and predicted particle acceleration:

$$MSE = \left\| \ddot{\boldsymbol{p}}^{t_k}_{predicted} - \ddot{\boldsymbol{p}}^{t_k}_{real} \right\|^2 \tag{7}$$

To avoid error accumulation, randomly generated noise is added to the training data. This allows the NN to avoid over-fitting and produce more generalised predictions. The Adam optimiser is used with a learning rate of 10^-4, decaying to 10^-6. Using a decaying learning rate allows the NN to learn generally at the start of the training process then optimise further as time goes by. This increases the training speed as the optimiser is not making small iterations right from the start (You et al. (2019)). Training was conducted on a high-end HPC (high-performance computing) GPU, where training times were found to range from a few hours for simpler datasets and up to a week for complex datasets. The simpler datasets typically two-dimensional domains made up of around 2000 particles, while the complex datasets were three-dimensional and featured up to 20000 particles. These datasets featured different materials, flows, and boundaries. Combinations of the named variables were also tested. The training was stopped when the MSE change was negligible.

The performance of the trained simulator was measured by using the MSE of the acceleration as used for the training. Table 4 shows the MSE values for a selection of datasets tested. An average MSE of $3.5x10^-9$ across the different datasets was found for a single step simulation. The average MSE increased to $8.2x10^-3$ when calculated across a complete rollout of the simulation. One outlier was the BoxBath dataset which features a solid cube of particles in a 'bath' of water particles. The one-step MSE was much greater than the average at $54.5x10^-9$, but the rollout MSE was in line with the others at $4.2x10^-3$. No reason was given for this; however, the node count, edge count, and simulation length were significantly smaller than the other three-dimensional datasets. The opposite was true for the time step duration and node connectivity radius.

Table 4: MSE values for tested model outputs (Taken from Sanchez-Gonzalez et al. (2020))

Domain		MSE	
		Single Step ($\times 10^{-9}$)	Full Flow ($\times 10^{-3}$)
3D	Water	8.66	10.1
	High viscosity fluid	1.32	0.618
	BoxBath	54.5	4.2
2D	Water	2.82	17.4
	High viscosity fluid	2.91	1.89
	Multiple fluid types	1.81	16.9
	Water with inclined boundary	4.91	11.6

Visually, the Sanchez-Gonzalez et al. (2020) simulator results are similar to the original dataset. The simulated outputs show realistic-looking fluid flows and interactions with boundaries are correct with no fluid passing through areas where it would be impossible to do so, as seen in figure 12.

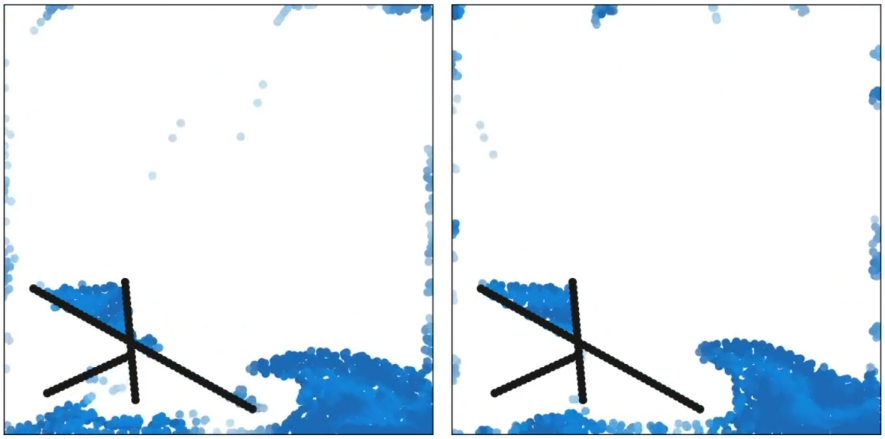


Figure 12: Water with inclined boundary domain result. The right domain shows the simulator output, the left shows the SPH output (Taken from Sanchez-Gonzalez et al. (2020))

It was found that the Sanchez-Gonzalez et al. (2020) simulator was sufficiently generalised after training that it was possible to generate simulations on domains very different to those it was trained on. This was tested by training the model on a two-dimensional dataset featuring a freefalling 'block' of water onto an inclined boundary. A simulation was then generated with a domain 32 times larger than the training dataset, 34 times more particles and a complex set of boundaries. This resulted in a simulation that was visually similar to a conventional simulation of the same enlarged domain (figure 13). MSE results were not given in the paper for this generalisation test.

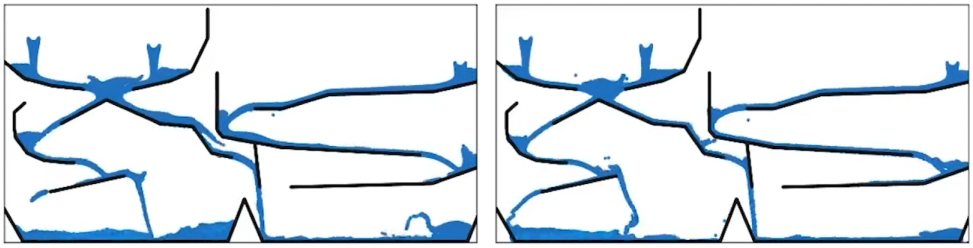


Figure 13: Generalisation test result. The right domain shows the simulator output, the left shows the SPH output (Taken from Sanchez-Gonzalez et al. (2020))

The Sanchez-Gonzalez et al. (2020) simulator was compared to the Ummenhofer et al. (2020) simulator (figure 14). It was found that the Sanchez-Gonzalez et al. (2020) simulator outperformed the other simulator on all datasets, especially when using a high viscosity fluid or with sand. However, using the BoxBath simulation, the Ummenhofer et al. (2020) simulator was unable to maintain the shape of the solid cube.

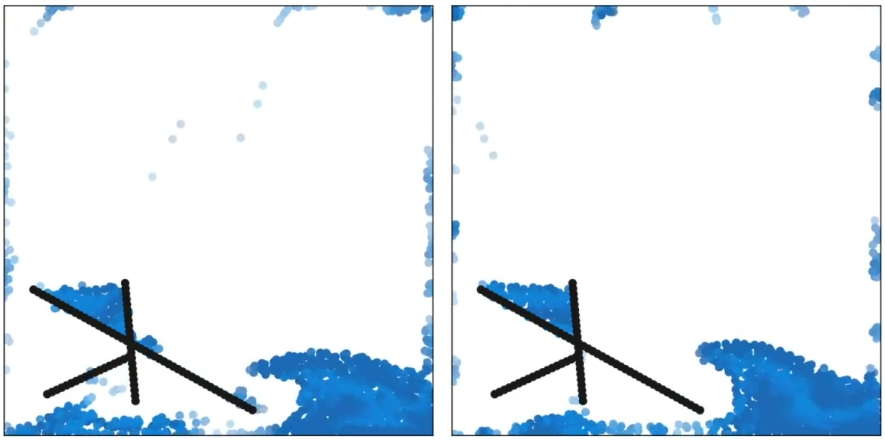


Figure 14: BoxBath domain result. The center domain shows the simulator output, the left shows the SPH output, and the right shows the Ummenhofer et al. (2020) output (Taken from Sanchez-Gonzalez et al. (2020))

3 Methodology

3.1 Jet Impingement Dataset Design

To train and validate the ML model, a dataset must be constructed. This dataset must be large enough to allow the model to train on a wide variety of different simulations to arrive at a general solution. Furthermore, the dataset's simulations must also allow the model to learn a range of fluid flows and interactions. On the other hand, the dataset may not feature too much detail to minimise the time required for training.

Jet impingement is a cooling system used in industrial applications, from turbine blades and electronics cooling to textile drying. This cooling solution is simple to set up and maintain while achieving high rates of heat transfer. Jet Impingement works by discharging a fluid jet onto the target surface, facilitating forced convective heat transfer (Glynn et al. (2005)). The objective of this work is focussed on fluid flow; therefore, the details of the heat transfer mechanisms will not be discussed in further detail as they are outside the scope.

The configuration of jet impingement systems can be classified as submerged jets if the fluid discharged has the same density as the fluid surrounding the target. An example of this is using a water jet on a plate in a bath of water. The system is classified as a free jet if the density of the jet fluid and surrounding jet is different; for example, a water jet on a plate exposed to air. Another classification is a confined or non-confined jet. The jet is bound between two surfaces for a confined jet, while non-confined jets can expand freely away from the target plate (Marzec & Anna (2014)).

For a free jet impingement system (figure 15), the fluid flow can be described as follows. First, the fluid leaves the nozzle and travels in the y axis to form a jet. A potential core where constant velocity is present until the turbulent shear layer generated from the nozzle exit expands fully across the jet in the x-axis. Next, the developing stage begins. With enough distance between the nozzle and the impingement plate, the flow can become fully developed. Near the plate, the fluid rapidly decelerates in the y-direction and accelerates in the x-direction, forming a stagnation region. At the stagnation region, the pressure is greatest. The jet now flows across the impingement plate, expanding as it travels along.

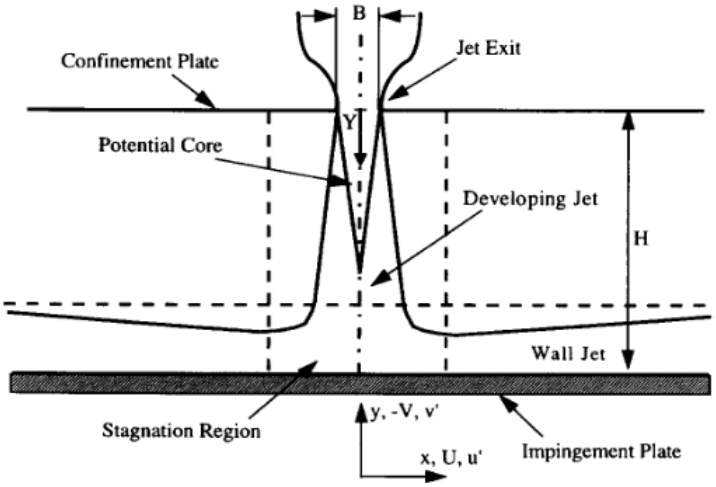


Figure 15: Jet impingement layout and typical flow (Taken from Marzec & Anna (2014))

The use of jet impingement for the dataset is a sensible choice as it provides several common flow features that the ML model can learn. A simplified geometry of a confined, free jet impingement system was created. Due to the symmetrical nature of the flow, half of the system was generated, and a symmetry boundary condition is used for the CFD data generation. The mesh features an inlet boundary condition in the top right corner directed towards a plate with an adiabatic, no-slip wall boundary extending across the mesh's bottom. The top of the mesh, except for the inlet, is also an adiabatic, no-slip wall boundary. The right-hand side is set up with an outlet boundary condition.

The mesh generated from the geometry was purposely made coarse, with 3718 cells (figure 16). Although this will reduce the accuracy of the CFD data generation, the decision was made with respect to time constraints, especially around the ML training phase. This decision will be further explained in section 4.1.

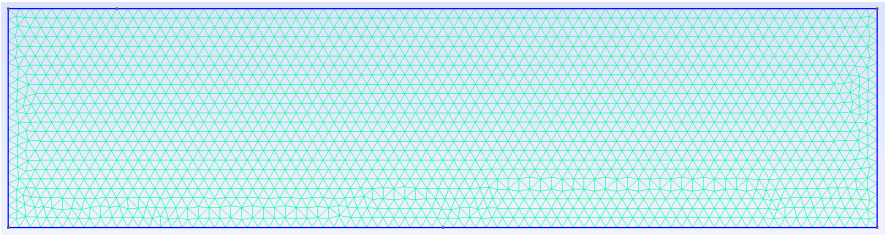


Figure 16: Generated mesh

3.2 Data Generation

The SU2 CFD tool was used to generate the dataset due to its flexibility and open-source nature. Other tools such as ANSYS were considered; however, it was found that automating the process of running multiple simulations and extracting the relevant data was difficult due to the restrictive nature of such closed-source programs. Automation was an essential aspect due to the large number of simulations, and therefore files that needed processing.

The CFD solver was set up to use Incompressible RANS with a constant fluid density model. Simulations were run on a range of velocities to create a suitably large dataset. The fluid selected for the simulations was air at standard sea-level conditions (table 5). As such, all velocities simulated are below Mach 0.3.

Table 5: CFD Parameters

Parameter	Value
Simulation time	1s
Time step length (Δt)	5ms
Air density	1.2886 kg/m^3
Air temperature	288.15K

3.3 Machine Learning Implementation

3.3.1 Design

The ML approach selected for this work is the GN framework. Although the CNN approaches showed good results, their use is limited as it is difficult to generalise these systems to different fluid flows. Contemporary CFD solvers and their counterparts in, for example, heat transfer and solid mechanics solvers use meshes as the backbone for their computations. Therefore, designing a ML model that computes on meshes is a good approach as it can integrate with other simulation tools. Furthermore, meshes are inherently graphs, so using the GN framework is justifiable.

The ML model uses Python as its programming language. This was selected due to the significant work in the past five years to develop ML libraries for the language, making it stand out as the programming language of choice for such projects. The TensorFlow platform was developed by Google for their internal ML needs but is now freely available as an open-source tool. TensorFlow is an ideal platform due to its high-level programming approach. It allows the rapid development of ML models by providing the necessary building blocks to design a system without requiring in-depth knowledge of the underlying code. Having the backing of Google means TensorFlow is designed for speed and scalability – perfect for large scale engineering projects where an ML model can be developed at a small scale then deployed for use by a team of engineers working on complex problems. TensorFlow works by constructing a graph of all operations and calculations required by a ML model with the relevant data, stored as tensors, passed around this graph. This graph is a mathematical representation of the computations instead of being coded. It is platform-independent, thus allowing a ML model to scale from a desktop computer to a supercomputer with no additional work by the user (Yegulalp (2019)).

The use of velocity values can calculate the fluid flow by rearranging the N-S conservation of

momentum equation. As only incompressible flow is being considered, the density can be considered a constant. The N-S equation can be rewritten by combining all the terms apart from the velocity derivative into a function, $f(\boldsymbol{u})$ (equation 8). This function is replaced with the ML model output, called acceleration in the rest of the text for simplicity.

$$\frac{D\boldsymbol{u}}{Dt} = f(\boldsymbol{u}) = NN(\boldsymbol{u}) \tag{8}$$

As discussed previously, a GN ML model is implemented to predict fluid flow on a mesh. Using the GN framework, an encode-process-decode architecture is implemented. The GN block for the processor takes the node, and edge attributes as inputs and returns updated node attributes (equation 9). This is a modification of the MPNN block where the global attributes are not taken into account. The decision for this was made as no global attributes are to be considered in this work to limit the dataset size. However, this means that the model will be unable to adapt to different fluids, for example.

$$\begin{gathered}\phi^e(\boldsymbol{e}_k, \boldsymbol{v}_{rk}, \boldsymbol{v}_{sk}) = f^e(\boldsymbol{e}_k, \boldsymbol{v}_{rk}, \boldsymbol{v}_{sk}) \\ \phi^v(\bar{\boldsymbol{e}}'_k, \boldsymbol{v}_i) = f^v(\bar{\boldsymbol{e}}'_k, \boldsymbol{v}_i)\end{gathered}$$

$$\rho^{e\to v}(E'_i) = \sum_{k:r_k=i} \boldsymbol{e}'_k \tag{9}$$

In the encoding step, the initial mesh is inputted. The mesh comprises triangular cells whose corners are represented as nodes on the graph built by the encoder. The sides of the cells are then extracted from the mesh and implemented into the graph as pairs of directed edges. Finally, the initial velocity is embedded as the node attributes, and the relative physical distances between nodes are embedded as edge attributes (i.e., the length of the triangles' sides).

In the processing step, the node velocity attributes are modified to include noise to aid the simulator's accuracy. The graph then undergoes M message-passing steps, where M is a hyperparameter. For each message-passing step, the edge attributes will be updated with the edge and node attributes as inputs then aggregated, followed by the node attributes being updated with the node attributes and updated edge attributes as inputs then aggregated also. Thus, the message-passing process produces an updated graph with the acceleration at each node.

Finally, the decoder step extracts the acceleration and velocity attributes and, using Euler integration (equation 10), calculates the updated velocity. The updated velocity values are then used to construct an updated mesh. The new mesh can then be outputted by the simulator for a single time-step simulation or re-inputted into the simulator to predict the fluid flow for additional time steps.

$$\boldsymbol{u}^{t+1} = \boldsymbol{u}^t + \dot{\boldsymbol{u}}^t \cdot \Delta t \tag{10}$$

The coded implementation of the ML simulator can be found in the Appendix.

3.3.2 Training

The updater functions used in the GN blocks are MLPs that need training to predict the fluid flow correctly. To achieve this, supervised training is implemented. The simulator is given a mesh to predict. The output is then compared to the true data by calculating the MSE. Using the Adam optimiser with the MSE as the input, the MLPs are tuned to reduce the MSE. This process is repeated on the entire training dataset until the MSE is sufficiently low enough that further training will not significantly affect the accuracy. The hyperparameters for the message-passing steps and MLP size will be taken from Sanchez-Gonzalez et al. (2020) and can be seen in table 6.

Table 6: Model hyperparameters (Taken from Sanchez-Gonzalez et al. (2020))

Hyperparameter	Value
Message-passing steps	10
Learning rate	10^{-4} to 10^{-6}
MLP Hidden Layers	2
Layer Size	128 Neurons

Due to the Coronavirus pandemic affecting the supply of computer hardware, particularly for GPUs (Molloy (2021)), training was conducted on a machine with a 12 core CPU and 16GB of RAM. This dramatically reduced the number of training steps that can be conducted as GPUs can provide as much as three times faster training times (Steinkraus et al. (2005)).

4 Results

4.1 CFD Results

A fine mesh of the jet impingement system featuring 58944 triangles was generated, and a simulation of 200 timesteps with a time delta of 5ms was run for a flow inlet speed of 20m/s was run to validate the CFD model. It was found that the flow behaved as expected. A stagnation region developed directly below the inlet, which caused the y-component of the velocity to approach 0 as the flow neared the plate and the x-component increased. In addition, an eddy can be seen forming just outside the stagnation region, which moves away from the inlet. This matches what is expected from figure 15.

This simulation took 23 minutes to compute, which would lead to a dataset of 100 simulations taking just over a day to generate. Another issue was the storage requirements for such a dataset which would require 720MB per simulation, leading to a 72GB dataset. Finally, the most critical hurdle was the RAM requirements for the training process. It was found that, on average, the memory allocation for the training stage was half the size of the dataset. Therefore, a 72GB dataset would require about 36GB of RAM. Although consumer computers with enough RAM exist, they are prohibitively expensive. To make running the training possible on most computers, the RAM requirement was capped to 3GB and, therefore, a maximum dataset size of 6GB.

Running the same simulation on a mesh with 3718 triangles produced a flow field with a, predictably, lower resolution. Although some detail is lost with this coarser mesh, it is possible to identify key flow features (figure 17). For example, the stagnation region and eddy both appear in the correct locations. It is clear, however, that the eddy is not as well defined in the coarse mesh. Another deviation from the fine mesh is the boundary layer at the plate. Wall treatment is typically used to ensure boundary layers are correctly modelled. However, it was not conducted in both the fine and the coarse mesh generation process. This decision was made to minimise the cell count and, therefore, the dataset size. The effect of this is that the boundary layer is more defined in the finer mesh. The RAM requirements justify this reduction in detail from the coarse mesh. An inaccuracy was observed in all simulations where periodically, the x component of the velocity will increase slightly then decrease once again. Although it was initially dismissed, it lead to an interesting effect discussed in section 4.3. The simulation on the coarse mesh was computed in 2 minutes, reducing the dataset generation time to just over 3 hours. The total size of the dataset was 5GB, in line with the RAM rule laid out previously. The coarse mesh also displays the same flow features expected in a jet impingement system, making it usable for the dataset.

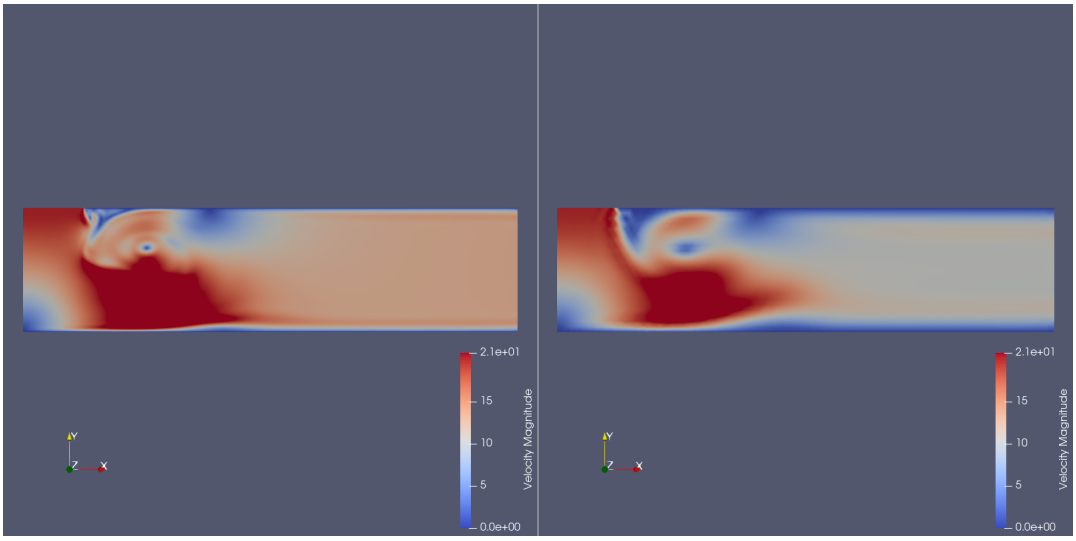


Figure 17: Fine mesh (left) versus coarse mesh (right) at time-step 150

4.2 Training Results

The ML model was trained with 100 CFD simulations of random inlet velocities between 1m/s and 40m/s. Just over 1 million training iterations were conducted with an average step time of 2.15 seconds for a total time of 25.5 days (figure 18). It can be seen from figure 19 that the smoothed loss gradient was steepest in the first 30000 training, and after this point, the gradient decreased. At the one-millionth iteration, a smoothed loss of 7.5 was achieved. It is clear from the graph that the loss is still decreasing, and as such, the model would need further training to provide accurate simulations. It is worth noting that the true loss at the one-millionth mark was 0.09. However, simply using this value as the loss without smoothing across the timespan is misleading as it only considers the performance for the single piece of training data used in the iteration. Low losses may be followed by relatively higher losses at other points in the training process, which shows that the model requires more training. Nonetheless, the model was sufficiently trained. Although it may not be accurate enough to measure the flow properties at a specific point on the mesh, it is possible to consider the overall flow.

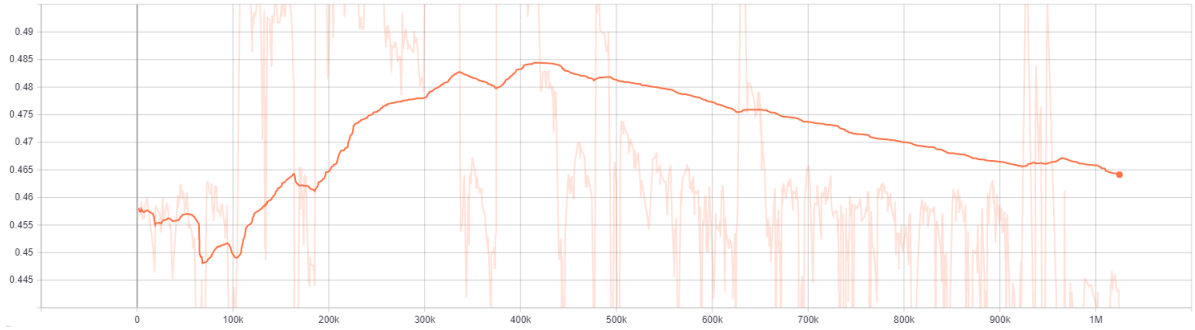


Figure 18: TensorFlow training speed output. The dark line represents the smoothed values, the light line represents the true speed. The x-axis shows the number of training steps, and the y-axis shows the number of steps per second

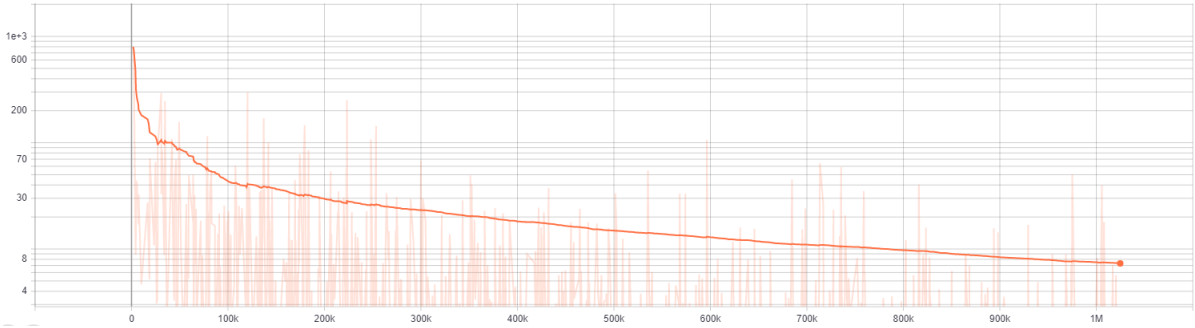


Figure 19: TensorFlow model loss output. The dark line represents the smoothed values, the light line represents the true loss. The x-axis shows the number of training steps, and the y-axis shows the loss

4.3 Machine Learning Model Correlation

Due to the large number of flows simulated, the following discussion will be focussed on three scenarios with an example flow for each. The scenarios are a simulation the ML model was trained on, a simulation with a velocity outside the training range, and a simulation with a velocity within the range but with a new mesh. For all figures in this section, the flow contour plot on the left of a figure represents the CFD output, and the right represents the ML model output. The data files for these simulations can be found in the Appendix.

In the first scenario, the ML model was given the same mesh used in training with an inlet velocity of 30m/s. As discussed in section 4.1, the CFD output took 2 minutes to compute while the ML model required just under 1 minute. The ML output matched the CFD output well for the first ten time-steps, looking at the velocity magnitude (figure 20). The MSE for the first six time-steps were 0 and remained below 0.1 up to the 10th time-step (figure 24). By the 50th time-step, the distinctive eddy can be seen, as well as the stagnation region (figure 21). Between the 10th and 50th time-steps, the MSE fluctuated between 1 and 29. After that, the MSE continued to increase,

which is apparent at the 100th time-step (figure 22), with the MSE peaking at 60 a few time-steps prior. Here, it can be seen that the ML model output has deviated significantly from the CFD output. Finally, by the 150th time-step, the flow breaks down completely as the MSE reaches 100 (figure 23). In this simulation, the ML model performed accurately at the start, but as the time-steps increased, the errors in the model accumulated, leading to it failing. During all time-steps, however, the velocity remained accurate directly ahead of the inlet. As the flow in this region is almost constant throughout the simulation, the ML model had a large amount of stable data to learn from and managed to keep the MSE low here. Plotting the MSE across time shows the expected trend upwards. However, peaks appear in the line approximately every ten time-steps, with every other peak being significantly pronounced. This is due to the oscillation observed in section 4.1. Breaking down the MSE into the x and y components shows that the y component MSE increases smoothly, and the x MSE oscillates (figure 25). Although the training data was flawed in the x component, increasing the overall MSE, looking solely at the y component shows the ML model maintained an MSE under one until the 46th time-step. The oscillation error in the x-component may have also helped learn the flow characteristics in the x-direction. In all the CFD outputs, the boundary layer was clearly defined regardless of the oscillation error, and this has carried over to the ML model.

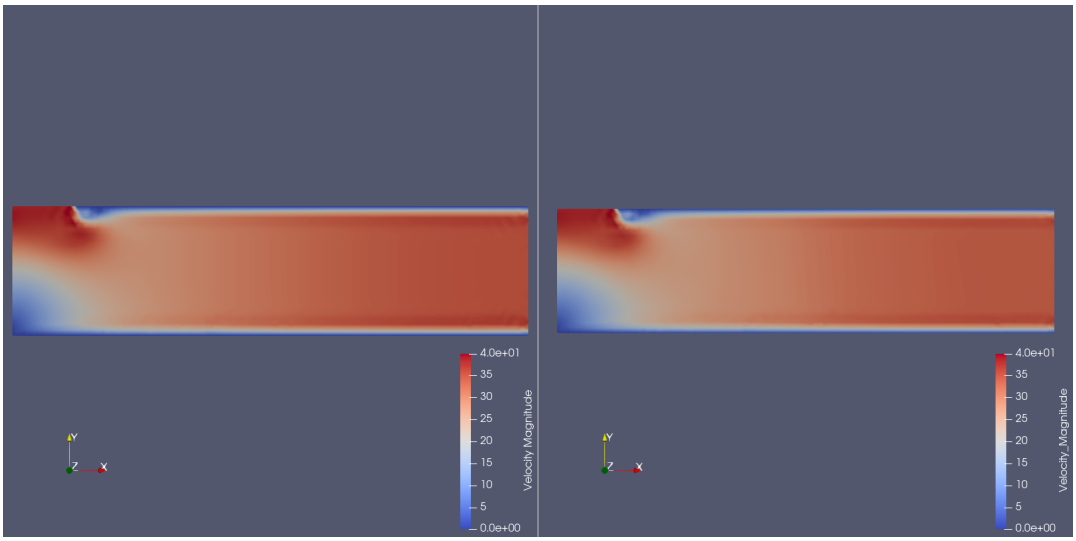


Figure 20: 30m/s simulations at time-step 10

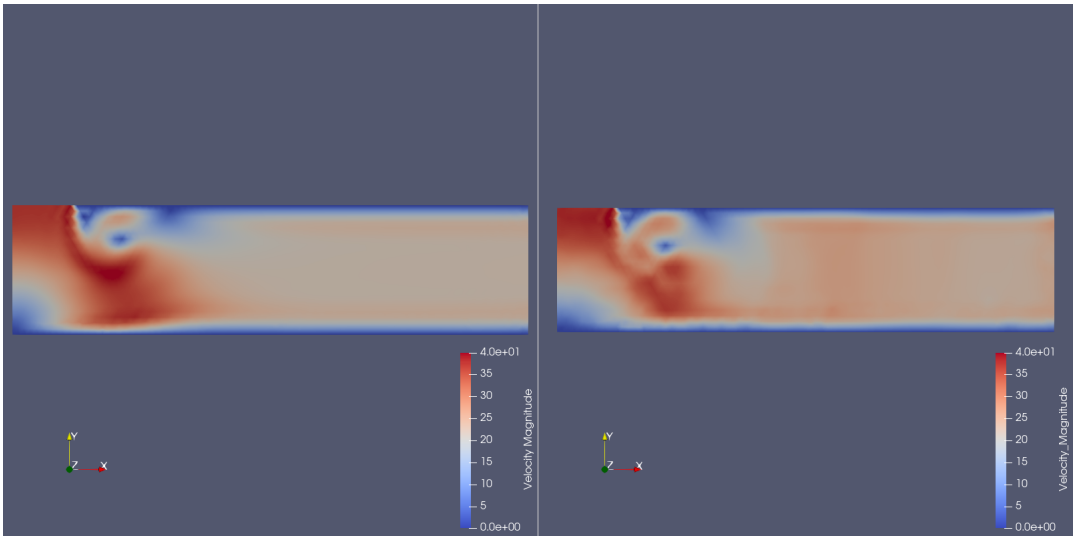


Figure 21: 30m/s simulations at time-step 50

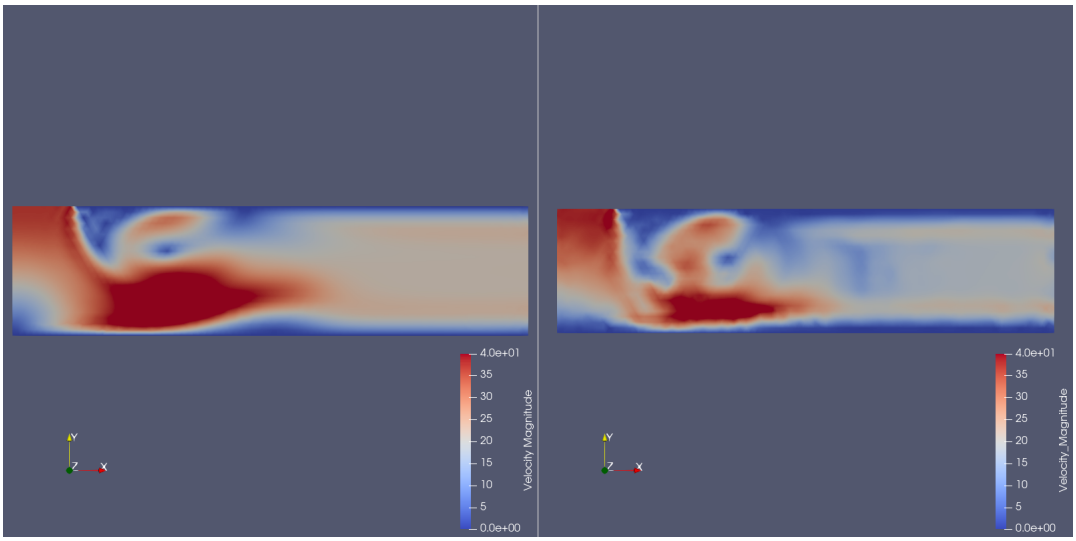


Figure 22: 30m/s simulations at time-step 100

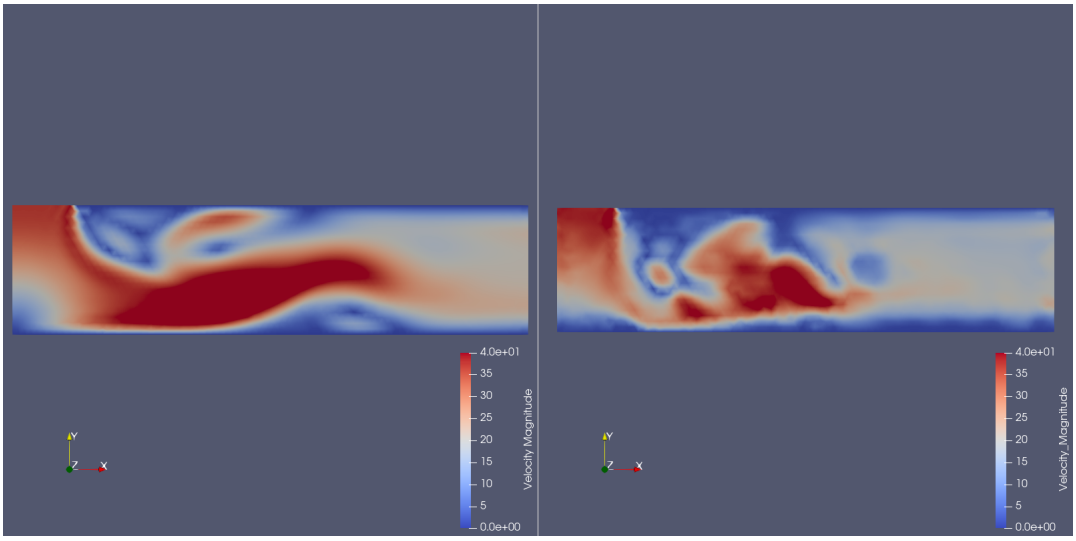


Figure 23: 30m/s simulations at time-step 150

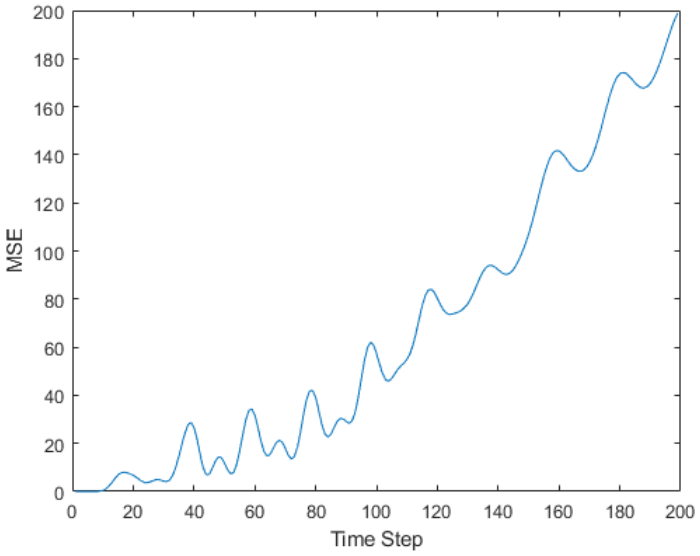


Figure 24: Average MSE with respect to time-steps

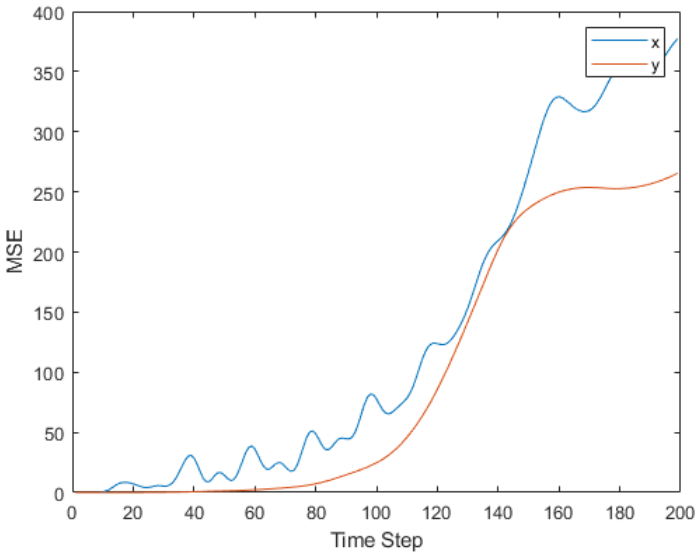


Figure 25: Average x and y component MSE with respect to time-steps for 30m/s simulation

In the second scenario, the same mesh was again used but with a velocity of 46m/s. Comparing the MSE over time for the 46m/s simulation with that of the 30m/s simulation (26), both start the first six time-steps with an MSE of 0, and an accurate velocity profile by the 10th time-step (27). After the 20th time-step, the MSE begins increasing rapidly. It can be seen that in the 100th (29) to 150th time-steps (30) region, the flow downstream was not simulated correctly. Nonetheless, the ML model was able to represent the stagnation region, inlet, and boundary layers. The eddy appeared slightly in the first few time-steps, but by the 50th (28), it had almost disappeared. This scenario shows that the ML model can predict flows outside the training data; however, it is very sensitive to errors. Although, more training and a larger dataset can rectify this problem.

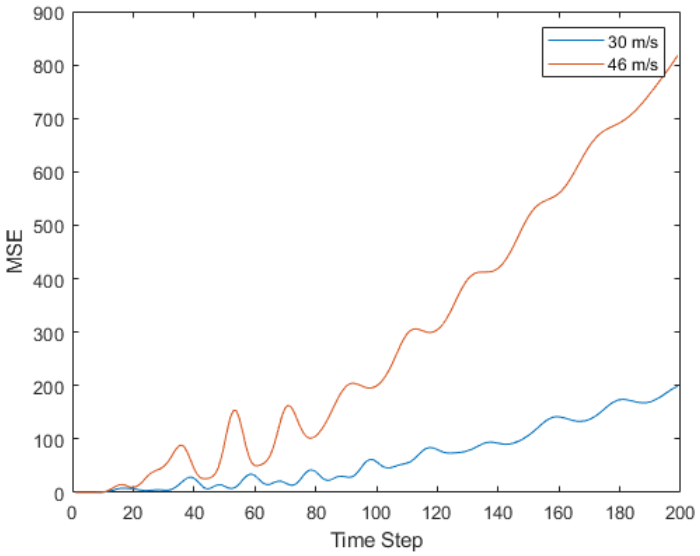


Figure 26: Average MSE with respect to time-steps for 30m/s and 45m/s simulations

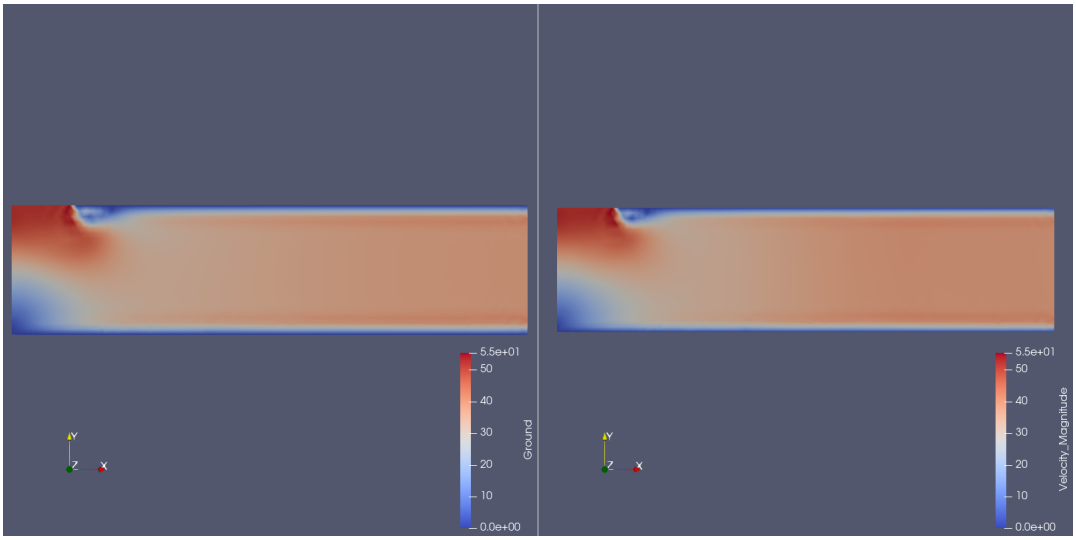


Figure 27: 46m/s simulations at time-step 10

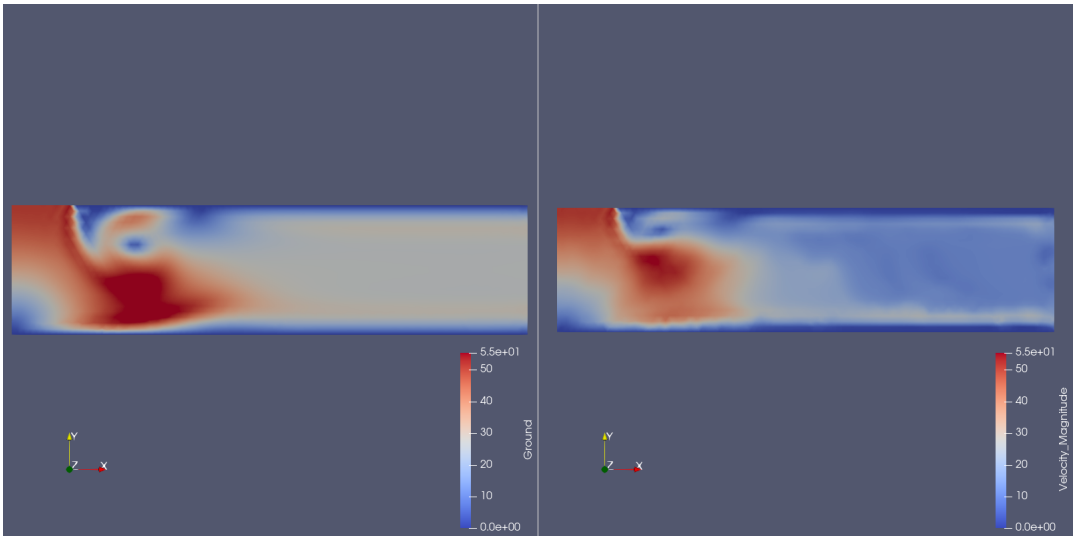


Figure 28: 46m/s simulations at time-step 50

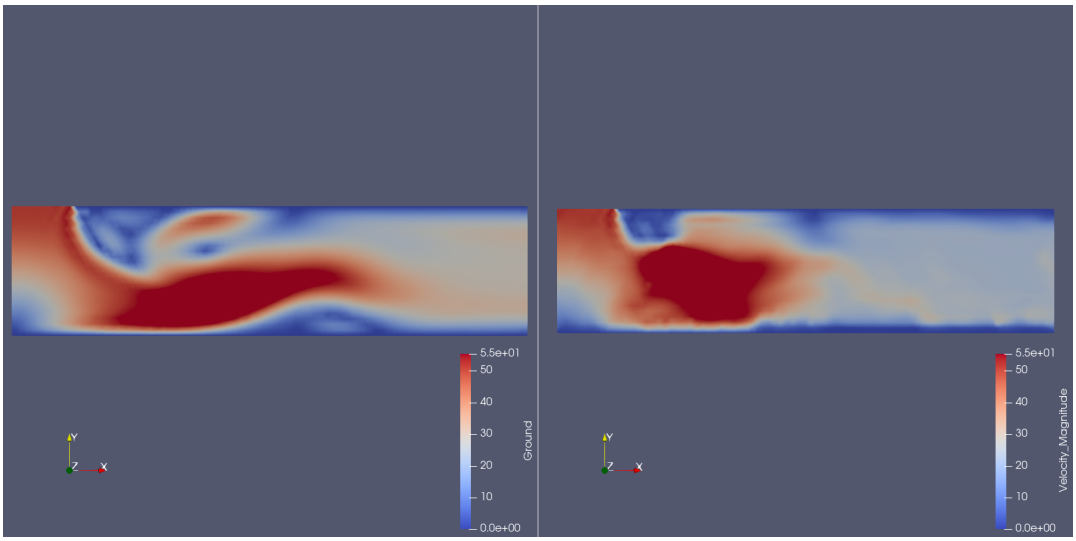


Figure 29: 46m/s simulations at time-step 100

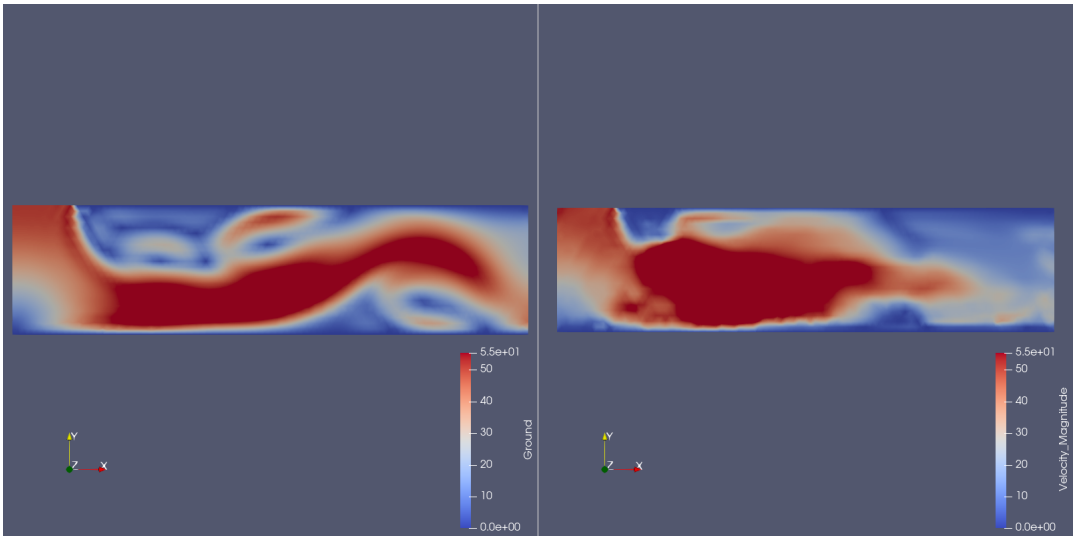


Figure 30: 46m/s simulations at time-step 150

Finally, a new mesh was introduced to the ML model to investigate the flexibility of the model. Due to the high MSE errors (figure 31) from the model not being fully trained, only a slight change to the mesh was made. The number of cells in the mesh was reduced to 2722. The ML model output is relatively accurate in the first 20 time-steps (figure 32). However, the output deviates from the CFD output within the next 20 time-steps (figure 33). By the 100th time step, the ML model output had broken apart (figure 34). From this, though, the areas where the ML model is most accurate are clear: the inlet, stagnation region, and boundary layer. These areas have been consistently accurate across all tests, and this is because these are flow characteristics that have been present in every training simulation, giving the model the most amount of data to learn from.

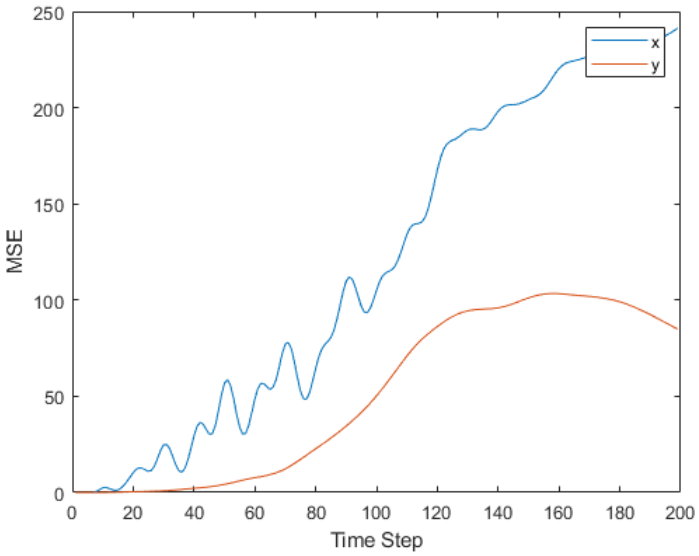


Figure 31: Average x and y component MSE with respect to time-steps for 30m/s alternative mesh simulation

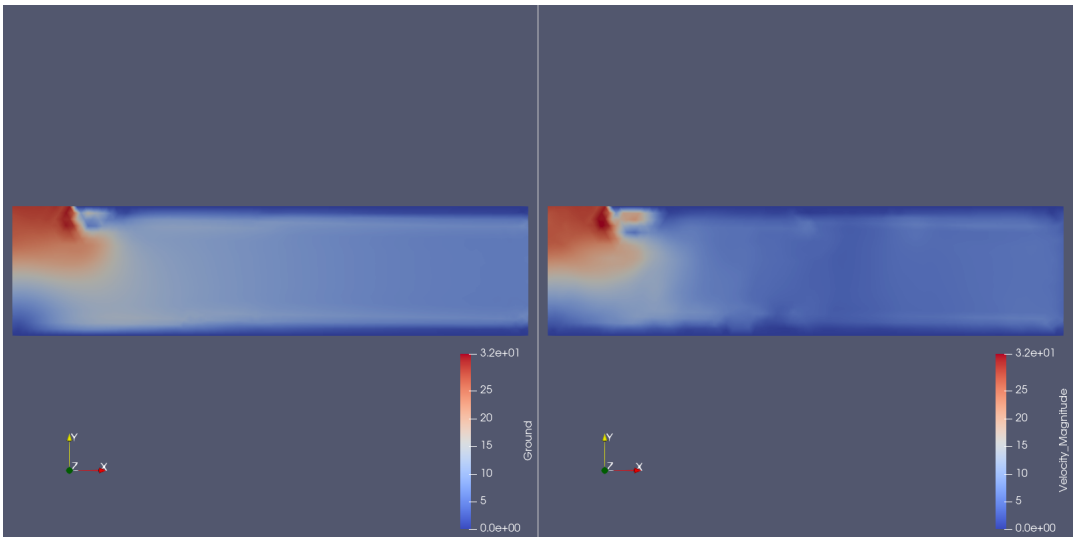


Figure 32: 30m/s alternative mesh simulations at time-step 20

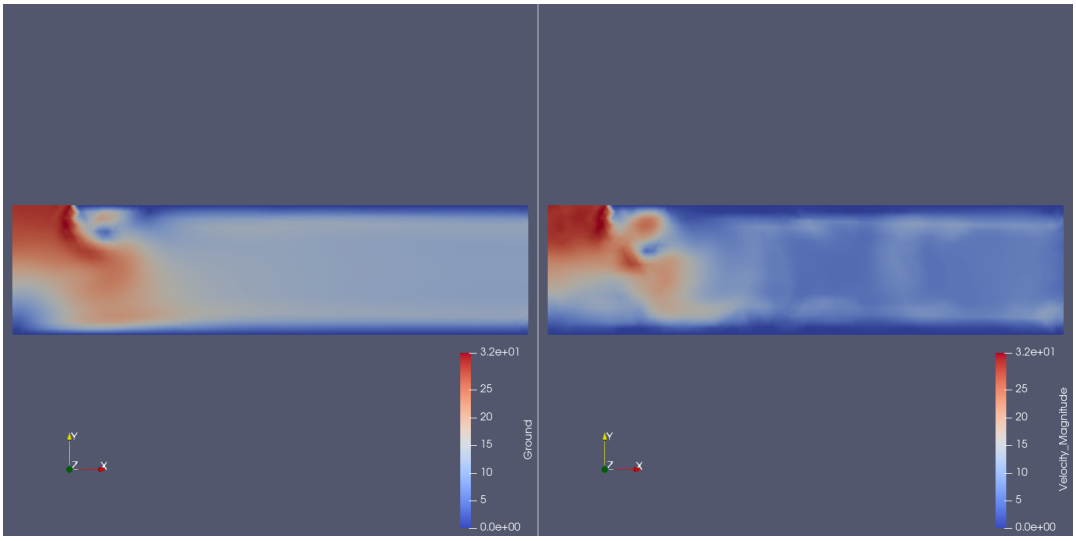


Figure 33: 30m/s alternative mesh simulations at time-step 40

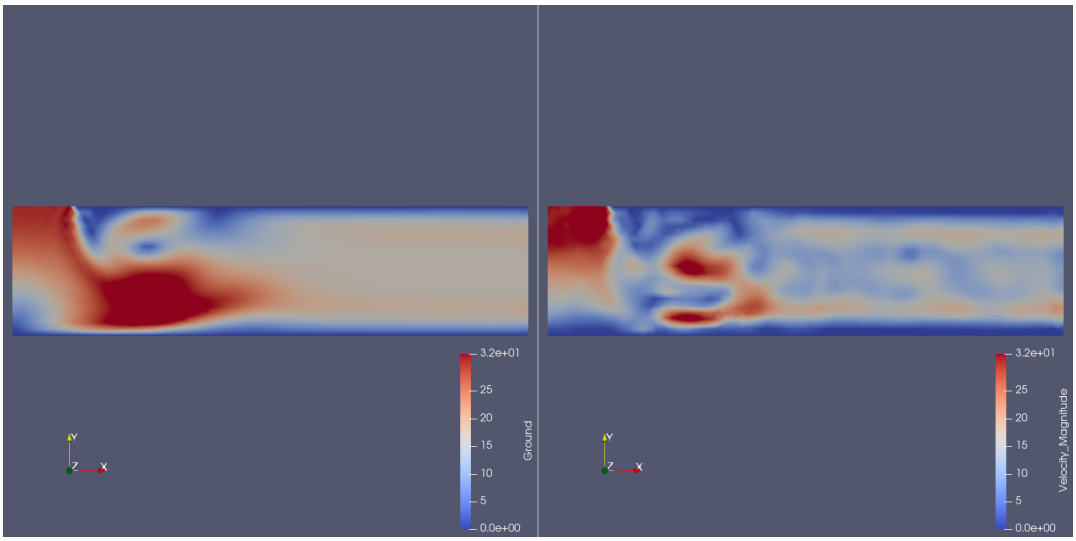


Figure 34: 30m/s alternative mesh simulations at time-step 100

5 Concluding Remarks

The implementation of ML techniques into fluid flow prediction has been shown to be in its early days but showing healthy progress. CNNs were shown to work effectively at simulating the pressure distribution around aerofoils and predicting fluid-particle interactions. Both implementations performed well in their given domains but lacked the flexibility to simulate domains particularly different to their training dataset. This is due to the CNN models predicting using trained 'knowledge'. The GN framework was introduced as a deviation from the 'training only' approach. The GN framework allows for the building of ML simulators that can also leverage known physical laws, reducing the complexity of the NNs, reducing training times and increasing accuracy. An example of the framework implemented to predict fluid-particle interactions was studied. The GN implementation was shown to outperform its CNN counterpart. From here, a ML simulator based on the GN model was implemented to predict fluid flow over meshes. Although the GN framework was not leveraged to its fullest, having only implemented one physical rule, the model showed promise with little training. The ML model learned key fluid flow properties such as boundary layers by training on a model of a jet impingement system.

Important lessons and recommendations arose from this work to apply to future projects or improvements to the implemented model. As computers continue to increase in performance, larger, more complex datasets can be used to train such models improving their accuracy. This was shown in the mesh-based ML model, where it could not generalise due to the lack of training steps and dataset size. Second, as mentioned, integrating physical laws such as RANS and boundary layer approximations can allow for training datasets specialised in areas that cannot be algebraically solved or are computationally intensive to simulate. Finally, this work can be expanded by using the techniques discussed to integrate other mesh-based simulators, such as Finite Element Analysis, into the model, providing rapid prototyping and insight into engineering projects, improving design workflows.

Appendix

The code for the mesh-based simulator can be found here: `https://github.com/abdelabdalla/Zephyrus/tree/main/primary`

The meshes shown in the results section can be found here. The .VTK files can be opened in programs such as Paraview and included in the files are the CFD output, ML simulator output, and the node MSE values. The averaged MSE values for each time-step can be found in the .CSV files: `https://github.com/abdelabdalla/Zephyrus/tree/main/Results`

References

Alpaydin, E. (2014), *Introduction to machine learning*, Adaptive computation and machine learning series, third edition. edn, MIT Press, Cambridge, Massachusetts.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y. & Pascanu, R. (2018), 'Relational inductive biases, deep learning, and graph networks'.

Bhatnagar, S., Afshar, Y., Pan, S., Duraisamy, K. & Kaushik, S. (2019), 'Prediction of aerodynamic flow fields using convolutional neural networks', *Computational Mechanics* **64**(2), 525–545.
URL: *https://doi.org/10.1007/s00466-019-01740-0*

Birch, D. M. (2020), 'Engm249 turbulence: Engm249 turbulence'.

Cengel, Y. A. (2014), *Fluid mechanics : fundamentals and applications*, third edition. edn, McGraw-Hill Education, New York, New York.

Cranmer, M., Sanchez-Gonzalez, A., Battaglia, P., Xu, R., Cranmer, K., Spergel, D. & Ho, S. (2020), 'Discovering symbolic models from deep learning with inductive biases'.

Fionda, V. & Palopoli, L. (2011), 'Biological network querying techniques: Analysis and comparison', *Journal of computational biology : a journal of computational molecular cell biology* **18**, 595–625.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O. & Dahl, G. E. (2017), 'Neural message passing for quantum chemistry'.

Glorot, X., Bordes, A. & Bengio, Y. (2011), Deep sparse rectifier neural networks, *in* G. Gordon, D. Dunson & M. Dudík, eds, 'Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics', Vol. 15 of *Proceedings of Machine Learning Research*, PMLR, Fort Lauderdale, FL, USA, pp. 315–323.
URL: *http://proceedings.mlr.press/v15/glorot11a.html*

Glynn, C., O'Donovan, T. & Murray, D. (2005), 'Jet impingement cooling', *Proceedings of the 9th UK National Heat Transfer Conference* .

Guo, X., Li, W. & Iorio, F. (2016), Convolutional neural networks for steady flow approximation, *in* 'Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining', KDD '16, Association for Computing Machinery, New York, NY, USA, p. 481–490.
URL: *https://doi.org/10.1145/2939672.2939738*

Imperial College London (2018), 'History of machine learning'.
URL: *https://www.doc.ic.ac.uk/ jce317/history-machine-learning.html*

Kingma, D. P. & Ba, J. (2017), 'Adam: A method for stochastic optimization'.

Marxen, O. (2019), 'Numerical methods & cfd (eng 3165): Solving for incompressible flow'.
URL: *https://surreylearn.surrey.ac.uk/d2l/le/lessons/189449/topics/1679165*

Marzec, K. & Anna, K.-P. (2014), 'Heat transfer characteristic of an impingement cooling system with different nozzle geometry', *Journal of Physics: Conference Series* **530**.

Molloy, D. (2021), 'The great graphics card shortage of 2020 (and 2021)', *BBC* .
URL: *https://www.bbc.co.uk/news/technology-55755820*

Nielsen, M. A. (2015), *Neural Networks and Deep Learning*, Determination Press.
URL: *http://neuralnetworksanddeeplearning.com/index.html*

Price, D. J. (2012), 'Smoothed particle hydrodynamics and magnetohydrodynamics', *Journal of Computational Physics* **231**(3), 759–794.
URL: *http://dx.doi.org/10.1016/j.jcp.2010.12.011*

Saha, S. (2018), 'A comprehensive guide to convolutional neural networks — the eli5 way', *Towards Data Science* .
URL: *https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53*

Sanchez-Gonzalez, A., Godwin, J., Pfaff, T., Ying, R., Leskovec, J. & Battaglia, P. W. (2020), 'Learning to simulate complex physics with graph networks'.

Steinkraus, D., Buck, I. & Simard, P. (2005), Using gpus for machine learning algorithms, *in* 'Eighth International Conference on Document Analysis and Recognition (ICDAR'05)', pp. 1115–1120 Vol. 2.

Ummenhofer, B., Prantl, L., Thuerey, N. & Koltun, V. (2020), Lagrangian fluid simulation with continuous convolutions, *in* 'International Conference on Learning Representations'.
URL: *https://openreview.net/forum?id=B1lDoJSYDH*

Yegulalp, S. (2019), 'What is tensorflow? the machine learning library explained', *InfoWorld* .
URL: *https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html*

You, K., Long, M., Wang, J. & Jordan, M. I. (2019), 'How does learning rate decay help modern neural networks?'.