Academic Year
(2018/2019)

Project No.
(B325)

CONVOLUTIONAL NEURAL NETWORK BASED CFD SIMULATION

CONVOLUTIONAL NEURAL NETWORK BASED CFD SIMULATION

Kok Xuan Lun Nigel

SCHOOL OF MECHANICAL AND AEROSPACE ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Year 2018/2019

Abstract

Aerodynamics play an important role in the engineering design process. With the development of more powerful computers, engineers are now able to create design prototypes more quickly rapidly and with better performance. However, the process of prototyping still entails a trade-off between cost and accuracy. Complicated shapes in difficult-to-model environmental conditions may become too time-consuming for a first-cut design analysis.

This Final Year Project aims to use machine learning techniques to predict the velocity field around a 2D object in a steady state laminar flow condition at a freestream velocity of $0.3m/s$. The models are built from scratch and three different network architectures are explored: (1) A convolutional neural network (CNN) using pure unstrided convolution layers, (2) an encoder-decoder CNN without a fully connected layer and (3) an encoder-decoder CNN with a fully connected layer. A total of 61 training sets, each being a $500s$ fluid simulation, were generated using the OpenFOAM solver.

It was found that the use of pure unstrided convolution layers allowed the CNN to accurately an obstacle's geometry, however, it has limitations in predicting the velocity field in regions further away from obstacles. The encoder-decoder (with fully connected layer) architecture, on the other hand, produced smooth images of the velocity field surrounding the obstacle and in the wake region. The encoder-decoder (without fully connected layer) architecture yielded results that lie in between the other 2 architectures.

While the replication of the velocity fields are not close to perfect, this project has shown that convolutional layers are able to pick up some physical information from the fluid simulations. The encoder-decoder architectures have demonstrated that the loss of spatial information via the downsampling process can lead to problematic predictions where obstacle geometries are erroneously generated. In general, an inverse relationship has been found between the retention of spatial information and the size of the region where predictions are effective. Areas of improvement to the strided convolution models include the use of skipped connections or the fusing of outputs from intermediate layers.

The field of machine learning is very broad and optimal neural networks has to be determined empirically. With more research and resources directed in this field, it is without a doubt that there will be successes in the future.

Acknowledgements

The Final Year Project: Convolutional Neural Network based CFD Simulation, has not been an easy journey. A large portion of my final year in Nanyang Technological University has been devoted to picking up the tools and bridging the knowledge gap needed to complete the project. Looking back, it was a bold move picking a project from an unfamiliar domain, but it was a choice that spurred a great amount of independent learning.

Firstly, I would like to express my gratitude to my project supervisor, Associate Professor Li Hua, for giving me the opportunity to work on this project. It has allowed me to gain insights on current engineering processes in the aerospace industry and a glimpse of how these processes can change over the next decade.

Next, I would also like to thank PhD student, Yang Juntao, who has played a pivotal role in the completion of the project. He has shared with me countless resources on machine learning which allowed me to grasp the basics as well as be exposed to more complex topics. Through this project, I was able to gain a more in-depth knowledge on the CFD process and had a lot of hands-on work in programming. Machine learning, Industry 4.0 and artificial intelligence have become buzzwords in recent years. I believe this project has given me a primer on the skills needed to excel and succeed in today's dynamic environment. I am grateful to be given the opportunity to learn from Juntao's expertise in the field of machine learning and high-performance computing.

Finally, I would like to thank the other students who provided me with the ideas, resources and support needed to complete this project.

Kok Xuan Lun Nigel

April 2019

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1. Motivations

Aerodynamics play an important role in many engineering processes. However, aerodynamic processes are not easily quantified during the concept and design phases. Engineers therefore usually conduct physical experimentation on prototypes to optimise their designs. In recent years, with large advancements in computational power, Computational Fluid Dynamics (CFD) became a popular tool to generate solutions for physical phenomena associated with fluid flows.

With developments in CFD, these tools have found their way into the study of fields such as medicine, chemistry and even the impacts of environmental phenomena. There is an increasing amount of work using simulation data to create reduced order models or surrogate models that can be evaluated with significantly less resources at an acceptable level of accuracy.

A neural network approach that compresses the computation time and memory usage of fluid simulations. It is proposed to take advantage of the recent successes of neural network-based models to build a model that can replicate a CFD simulation.

1.2. Objectives and Scope

This project aims to develop a convolutional neural network (CNN) which predicts the flow around an obstacle. The scope includes the generation of the full set of training data from scratch using a CFD solver program and a freshly trained CNN model. The development of such a model provides an idea of how CFD can done more efficiently and more cost-effectively in the future.

Chapter 2 discusses some of the past work done as well as some gaps in research that this project addresses. Chapter 3 and Chapter 4 introduces some basic concepts of CFD and machine learning respectively. Chapter 5 outlines the methodology used for experimentations. Chapter 6 discusses and analyses the results obtained. This report is concluded, with some suggestions for future work in Chapter 7.

Chapter 2: Literature Review

2.1. Related Works

Machine learning has been around for a long time, yet the field of image recognition has gained traction after LeCun et al. [1] pioneered the use of convolutional neural networks. Their research has shown that convolutional neural networks have outperformed other methods of feature extraction in their research on handwritten character recognition using the Modified National Institute of Standards and Technology (MNIST) database [2].

In the 2012 ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), the AlexNet [3] emerged the top performer on a historically difficult ImageNet dataset. It used a relatively simple architecture (compared to modern architectures), with techniques such as data augmentation and dropout which are still popular today. Their paper illustrated the benefits of using CNNs and were able to support with record-breaking performance in the ILSVRC competition. While there are many other well-performing models around, the ones developed by LeCun and the AlexNet appeared to be among the most popular in machine learning literature due to their pioneer status.

There has been work conducted on machine learning techniques in the field of fluid simulations as well. Tompson et al. [4] and Yang et al. [5] used a neural network to solve the Poisson equation to accelerate Eulerian fluid simulations. The most time-consuming step in using the Eulerian method to discretise the Navier-Stokes equations is known as the projection step. As the Poisson equation needs to be solved iteratively by numerical methods, this step costs a large amount of computational power and causes inconveniences whenever tiny adjustments are required in the fluid simulation. While both aim to solve the same problem, their methodologies differ. Tompson et al. used CNNs in a novel unsupervised learning framework that accepts the divergence of the velocity field and a Boolean occupancy grid geometry field as inputs and outputs a pressure field. Yang et al. projects a grid network over the simulation and for each grid cell, an artificial neural network (ANN) is used to predict the pressure in the next frame when fed a 19-dimensional vector as input. This 19-dimensional vector includes all known variables that influence the results of the Poisson equation. The use of neural network models in their study has resulted in faster computation times and good accuracy. As it has been shown, there is no single solution to a defined problem and any solution has the potential to produce successful results.

Chu et al. [6] used CNNs to synthesise high resolution smoke flows using reusable repositories of space-time flow data. Their research proposed the use of a large collection of pre-computed space-time regions, from which they will synthesis new high-resolution volumes. In order to find the best match space-time data from this repository, a novel flow-aware feature descriptor was used. The calculation of these descriptors and the encoding of discretisation errors from effects of numerical viscosity are done via a CNN. These correspondences and repository look-ups are done for localised flow regions, known as patches. A deforming Lagrangian frame is employed to track these patches over time, with each patch being tracked independently and all patch-based computations done in parallel. Their model accurately predicts the motion of smoke flows with high resolution and shown that the CNN was able to establish correspondences between different simulations in the presence of numerical viscosity.

Guo et al. [7] used deep CNNs for CFD modelling to provide lightweight flow performance feedback to improve the interactivity of early design stages, design exploration and optimisation. The model utilised a shared-encoding and separated-decoding architecture and uses the signed distance function as an alternative geometry representation as input. There are 2 decoding parts for 2D geometry obstacles (one for the X and Y velocity components) and 3 decoding parts for 3D geometry obstacles (one for the X, Y and Z velocity components). Their model estimates the velocity field 2 orders of magnitude faster than a GPU-accelerated CFD solver and 4 orders of magnitude faster than a CPU-based CFD solver to model steady state laminar flow at a cost of low error rates.

Jin et al. [8] used a novel fusion CNN (Figure 1) to map the relationship between the pressure on a cylinder surface to the velocity field of the wake at different Reynolds numbers. In Figure 1, Conv represents a convolution layer, ReLU represents a Rectified Linear Unit activation, FC represents a fully connected layer, Pooling represents a max-pooling layer and CONCAT represents a concatenation of the different paths. In this fusion CNN architecture, there are 2 paths with a max-pooling layer each and 1 path without. It was deduced that such an architecture can capture spatial-temporal information and features that are invariant of small translations in the spatial-temporal series of pressure fluctuations on the cylinder.

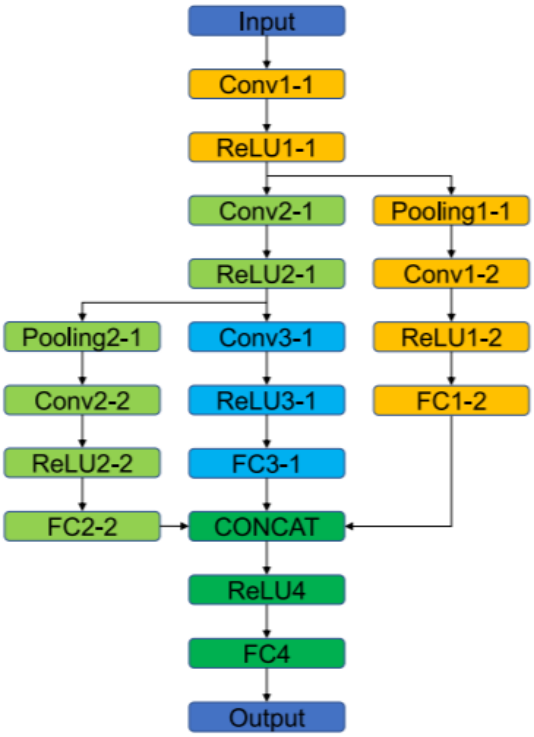

Figure 1: Fusion CNN architecture proposed by Jin et al. [8]

The model predictions of the velocity and vorticities in the wake flow agree well with the data obtained by CFD. Even though the model is trained within a range of Reynolds numbers 70 to 900, the model exhibits some extrapolation ability and accurately predicts for Reynolds numbers in the range 60 to 1100. Jin et al.'s research has also shown the importance of the quality of training data as their model is unable to predict intrinsic features that do not appear in their data. For example, since there is no vortex shedding at Reynolds 10 and this intrinsic mechanism is not embedded in the training set of Reynolds numbers 70 to 900, their model is unable to accurate predict the case where Reynolds number is 10.

Due to the relatively faster computational speeds of machine learning models, they have become a viable candidate in practical situations were speed is crucial to human safety. Na et al. [9] presented a high-accuracy gas dispersion model using a variational autoencoder with deep convolutional layers and a deep neural network with batch normalisation (VAEDC-DNN). Based on 3 inputs: wind speed, wind direction and release rate, the model outputs a contour plot of the probability of death (P_{death}) in a region. Previous studies on reduced-order models for chemical engineering problems used principal component analysis (PCA) for data reduction and feature extraction [10]. However, in more recent years, deep autoencoders have shown to be superior in terms of reducing nonlinear data compared to PCA [11]. According to their research, P_{death} images generated from CFD gas dispersion models are highly nonlinear

and is considerably difficult to extract features from. Therefore, the VAEDC-DNN model was proposed. The VAEDC extracts representation features of the P_{death} contour with complicated urban geometry in the latent space, while the deep neural network (DNN) maps the variable space into the latent space for the P_{death} image data. The model architecture is illustrated in Figure 2.

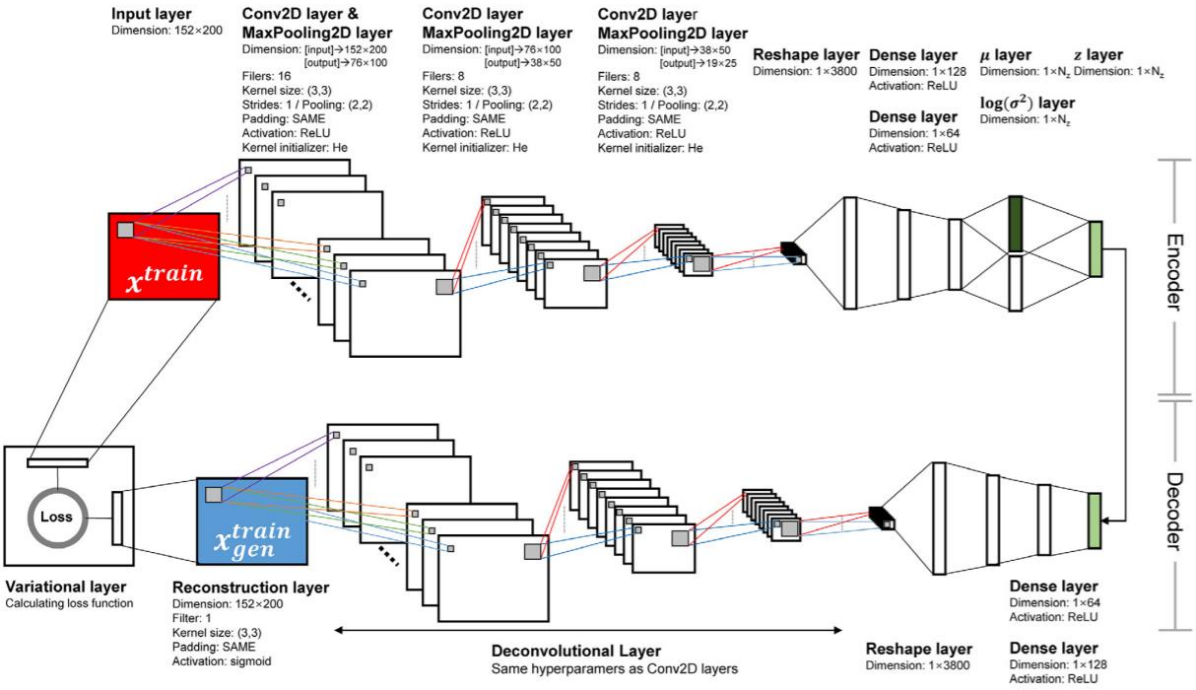

Figure 2: VAEDC-DNN architecture [9]

To test the performance of the model, the team used a toxic gas release scenario modelled by a CFD software application Flame Acceleration Simulator (FLACS). The VAEDC-DNN architecture has a mean square error (MSE) 47.7% as low as other models used in the research, exhibiting superior performance.

	CPU time	Storage	Use in real-time alarm systems
CFD model	~700 s with 16 cores	GB scale	Thousands of cases are required for predicting all variables
VAEDC-DNN	<1 s with single core	KB scale	With hundreds of cases, the surrogate model can predict all variables

Figure 3: Comparison between CFD model and VAEDC-DNN model [9]

From Figure 3, Na et al. has also shown that there is significant speed-up using a machine learning model instead of a CFD model. Speed is paramount to the problem that was targeted in the research: Constructing an early warning system for chemical accidents.

Over the years, newer techniques are introduced to address the shortfalls of certain model layers. The following example adopts a technique similar to boosting used in contemporary classification networks to improve CNN performance. Long et al. [12] adapted contemporary classification networks such as AlexNet [3], VGG net [13] and GoogLeNet [14] into fully convolutional networks (FCNs) for the task of semantic segmentation. By using convolution layers in place of fully connected layers in the above networks allow the model to output a heatmap rather than a classification label (Figure 4).

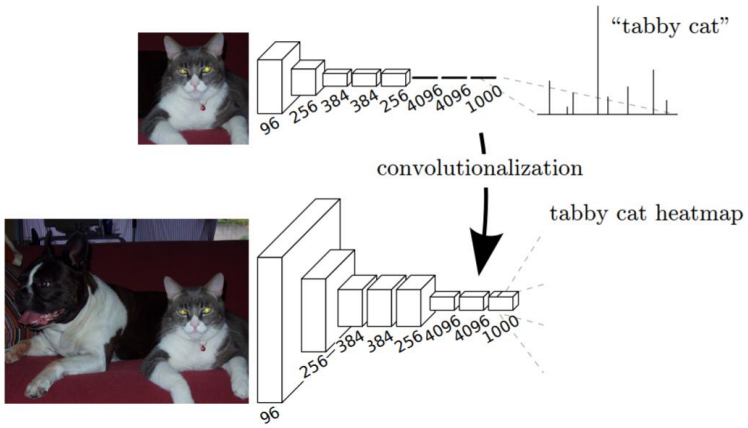

Figure 4: Transformation of fully connected layers into convolution layers

From Figure 4, it is observed that the previous convolution layers have resulted in downsampling of the input. In order to output a heatmap that has the same dimensions as the input for image segmentation, deconvolution layers are used for upsampling. 3 types of upsampling scheme are experimented, shown in Figure 5. Figure 6 illustrates the upsampling process for each scheme. FCN-32s is produced by a 32× upsampling, however, the output is rough as the location information lost through downsampling is not recovered. FCN-16s is produced by fusing the 2× upsampled output from pool5 with pool4 and then performing a 16× upsampling. FCN-8s is formed via a similar set of operations. Figure 7 compares the performance of each of the FCNs with the ground truth output. Deep features are obtained when going deeper (downsampling), but spatial information is being lost at the same time. The relatively better prediction of the FCN-8s supports the hypothesis that fusing the outputs from shallower layers can enhance the result.

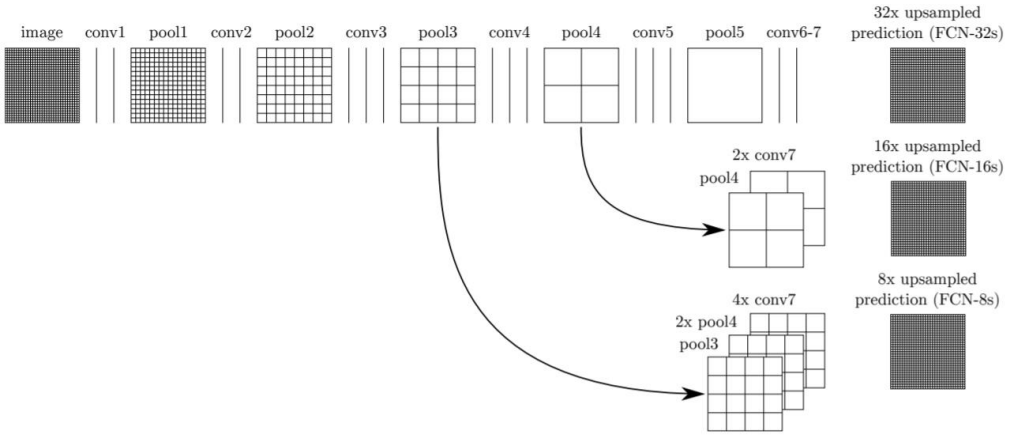

Figure 5: 3 types of FCNs explored by Long et al. [12]

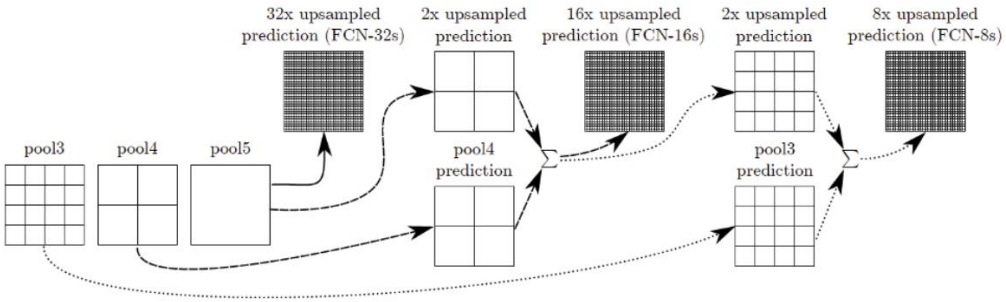

Figure 6: Enhancement of upsampled heatmaps [15]

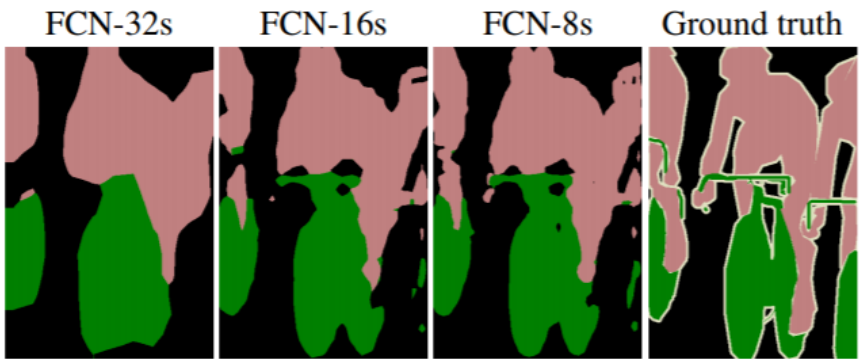

Figure 7: Performance of the FCN-32s, FCN-16s and FCN-8s [12]

2.2. Gaps in Past Research

While there has been a large amount of research in the field of machine learning in CFD simulations, there is possibly an equally large amount of uncovered ground. Meanwhile, there has been a lot more work on improving models that aim to solve computer vision problems.

Long et al. [12] and Ning et al. [16] has demonstrated the strength of FCNs in solving non-fluid simulation problems. Guo et al. [7] has achieved strong performance with their encoder-decoder CNN architecture in the prediction of laminar fluid flows around 2D and 3D objects,

therefore, the model built for this Final Year Project (FYP) will adapt some of the layer hyperparameters (number of filters, kernel size etc.) from their research. The network architecture used for this project will be different, but the layers will share some similarities to Guo et al.'s model. In addition to building a working model, an analysis will be done to evaluate the strength of a FCN in solving fluid flow problems.

Chapter 3: Computational Fluid Dynamics

Since the appearance of computers, CFD software has appeared in the playing field and been evolving ever since. It has alleviated the engineers' workload by handling the calculations much more quickly and accurately. CFD has found its place in almost any process where a fluid flow is involved and has been a pertinent tool in performance improvement. For example, in sports, CFD can be used optimise the sitting position of a road biker (Figure 8) and the design of the bicycles to minimise turbulence.

Figure 8: A road biker and corresponding aerodynamic analysis [17]

CFD is a branch of fluid mechanics using numerical methods to solve fluid flow problems. Computers perform the calculations that simulate the interaction of fluids with surfaces through codes known as CFD solvers (eg. ANSYS, FLUENT, STARCD, OpenFOAM etc.).

3.1. The Navier-Stokes Equations

The Navier-Stokes equations provide the governing mathematical framework for the changes of fluid properties during dynamic interactions. They have been used to model phenomena such as weather, pollution and flows around airfoils.

For a compressible Newtonian fluid, the equation yields:

$$\underbrace{\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)}_{1}=\underbrace{-\nabla p}_{2}+\underbrace{\nabla\cdot\left(\mu(\nabla\mathbf{u}+(\nabla\mathbf{u})^{T})-\frac{2}{3}\mu(\nabla\cdot\mathbf{u})\mathbf{I}\right)}_{3}+\underbrace{\mathbf{F}}_{4}$$

Figure 9: Components of the Navier-Stokes equation [18]

Where the various terms correspond to the inertial forces (1), pressure forces (2), viscous forces (3) and external forces applied on the fluid (4). The Navier-Stokes equations are always solved together with the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

The Navier-Stokes equations represent the conservation of momentum while the continuity equation represents the conservation of mass [18]. These equations are solved for a given set of boundary conditions for the pressure and velocity in a given geometry. For simple conditions such as the internal laminar flow of a pipe, the equations can be relatively simple to solve. However, the solution for flows around an aircraft are much more complicated and will require the use of computational methods.

The solution to the Navier-Stokes equations is a velocity field, defined at every point in a region during an interval of time. After obtaining the velocity, other quantities such as temperature or pressure can be obtained via additional relationships.

3.2. The General CFD Process

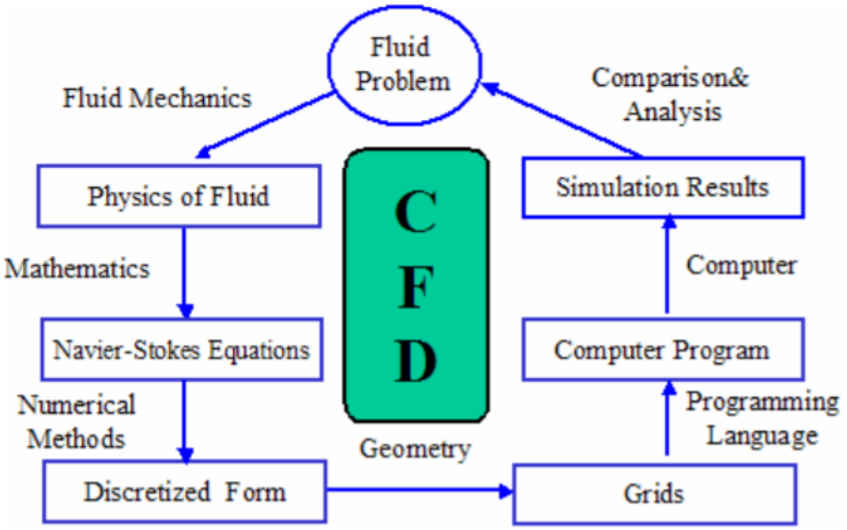

Figure 10: The CFD process [19]

The first step of CFD analysis is to formulate the fluid problem, which includes gathering information on the fluid properties, nature of the flow, geometry and the boundary conditions. These flows are governed by the Navier-Stokes equations. For computers to generate a solution to the Navier-Stokes equations, it has to be translated to the discretised form. Numerical discretisation methods, such as the finite difference method (FDM), finite element method (FEM) and finite volume method (FVM) are used to convert the non-linear partial differentiation equations of the Navier-Stokes into non-linear algebraic equations. The whole fluid problem is then divided up into grids, which the discretisation is based on. A computer programme (known as a solver) is then used to iteratively solve the discretised problem and

generate the solutions. The solutions will have to be analysed and validated with experimental results. If the simulation results do not agree with experimentation, then modifications will have to be made to the definition of the fluid problem and the whole process is repeated.

3.3. Recent Developments in CFD

Traditional CFD methods are penalised by long response times because of the complexity of the underlying physics.

In the last two decades, the Lattice Boltzmann Method (LBM) [20] has gained popularity as an alternative approach to CFD, achieving considerable successes in fluid flows and heat transfer problems. The discrete Boltzmann equation is solved to simulate the flow of a Newtonian fluid based on the Bhatnagar-Gross-Krook (BGK) collision model. The LBM was designed to run effectively on parallel architectures since the collisions are calculated locally. This makes the LBM a code that scales well on a Graphics Processing Unit (GPU). However, the LBM is still computationally expensive and requires numerous iterations.

Chapter 4: Machine Learning and Deep Learning

Artificial Intelligence (AI) has been around for years, however, this technology has only recently been able to find real world applications. With investments by tech giants and start-ups increasing 3 folds to $40 billion as of 2017, AI has finally gained traction [21].

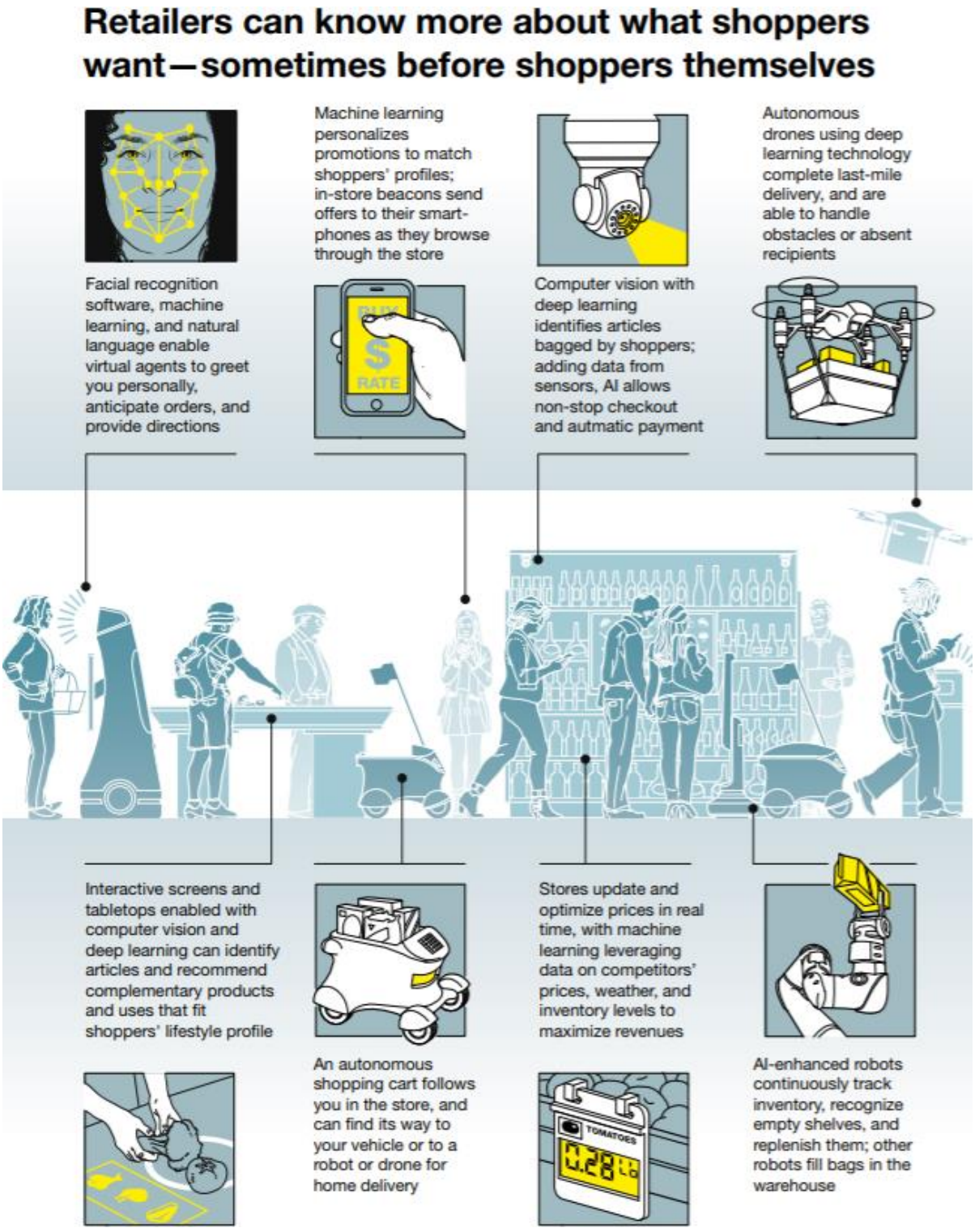

Figure 11: Infographic by McKinsey on the use of AI in retail operations [22]

The infographic (Figure 11) by McKinsey & Company illustrates the potential of incorporating AI end-to-end in the retail industry. Capabilities such as facial recognition, predictive analytics, robotics and automation are employed to provide consumers with better shopping experiences and help retailers better manage their inventories and sales. The possibilities are limitless and are limited only by creativity. The incorporation of AI into everyday lives drives the need to develop newer and more advanced capabilities, which in turn increases the areas where AI can integrate into daily life. The result is the establishment of a virtuous cycle that supports the boom of the AI development globally and Industry 4.0 in Singapore.

4.1. Conceptual Understanding of Machine Learning

Machine learning can be explained as using past data to make predictions on the future. It has gained popularity over the past decade due to improvements in computational power and the ever-increasing amounts of data available. With reference to Figure 12, based on the data, a model tunes its parameters (known as weights and bias, denoted by W and b respectively) to "fit" a relationship between the data so that it can make predictions. The tuning process is iterative. With each iteration, an error value is computed and is used by the model to re-adjust its parameters to improve its accuracy as it makes it next prediction. A more mathematical explanation will be discussed in Section 4.4.

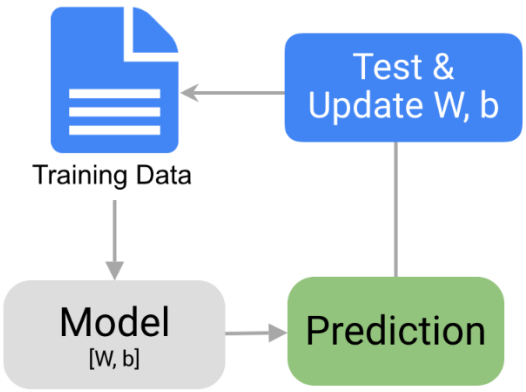

Figure 12: Simplified machine learning process [23]

Deep learning is a branch of machine learning that emphasises on the use of multiple levels of abstraction of data. It features multiple layers of nonlinear processing units for feature extraction and transformation of data. Deep learning methods have been popularly used in dimensionality reduction, feature learning, collaborative filtering and solving classification problems. More recently, deep learning has garnered traction in science and engineering as a tool for data-driven decision-making systems (eg. automated driving systems [24]).

4.2. Types of Machine Learning Problems

Machine learning is not a “one size fits all” solution. For each problem defined, there is a type of machine learning solution that can better address its needs. This section introduces two broad categories of machine learning solutions and the types of problem they address. The concepts mentioned in this section are non-exhaustive but are more commonly used.

4.2.1. Supervised Learning

The training data for supervised learning consists of both inputs and output labels. The machine learning model forms a hypothesis (relationship) between the inputs and output labels and uses this hypothesis to make predictions for new inputs it has not encountered before.

4.2.1.1. Classification Problems

In classification problems, the model predicts the categorical response value based on the inputs. There are binary classification problems such as predicting whether an email is a spam or not a spam and there are multi-class classification problems such as predicting whether an image is a cat, dog, bird or insect.

4.2.1.2. Regression Problems

In regression problems, models predict continuous response values ranging from $-\infty$ to ∞. An example of a regression problem is predicting the price of a house based on parameters such as location, area etc.

4.2.2. Unsupervised Learning

Unlike supervised learning, the training data for unsupervised learning does not contain the output labels. The goal of unsupervised learning is to discover some hidden structure in unlabelled data.

4.2.2.1 Clustering

Clustering is used to group similar things together. It works similarly to classification problems except that the output labels are not given. An example of a clustering problem would be to cluster the news in a newspaper into different types of news. In this case, the type of news is not known, but the news in each cluster share certain similarities.

4.3. Convolutional Neural Networks

CNNs are the neural networks of choice when the data involves spatial information that needs to be preserved. It has applications in areas such as classification, speech recognition, facial recognition and image restoration.

Traditional CNNs consist of an input layer, convolutional layers, pooling layers and fully connected layers. Figure 13 shows the components in a typical classifier neural network, where the model takes in an image as input and outputs the probabilities as to what the image is showing. Essentially, the CNN able to "see" and recognise the object shown in the image.

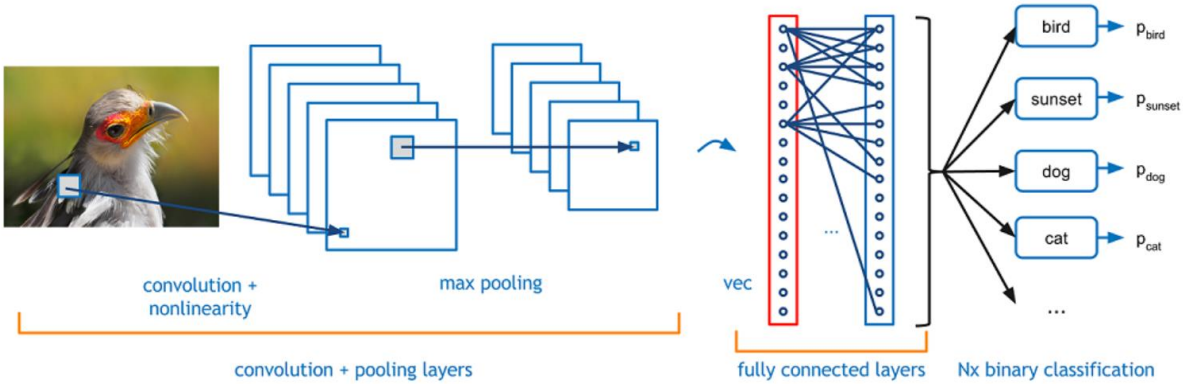

Figure 13: Schematic of a classifier built using CNNs [25]

Subsequent sub-sections will describe some of the layers and components of a typical classifier CNN. It should be noted that the CNN used in this FYP aims to solve a regression problem while a classifier CNN aims to solve a classification problem. While both CNNs differ in terms of the fundamental problem and architecture, it is still important to understand the functions of the individual components and appreciate how they fit into a CNN.

Section 4.3. serves as a primer to CNNs in general. Explaining the individual components and choices available for each hyperparameter are numerous and will require a large amount of time. Therefore, a greater focus will be placed on commonly used basic components.

4.3.1. Convolutional Layers

A convolutional layer extracts features from the input while preserving spatial relationships. A smaller matrix, known as a kernel (or filter), slides over the input and computes the dot product (in other words, sums the results of an elementwise multiplication) to generate a feature map (or convolved feature or activation map).

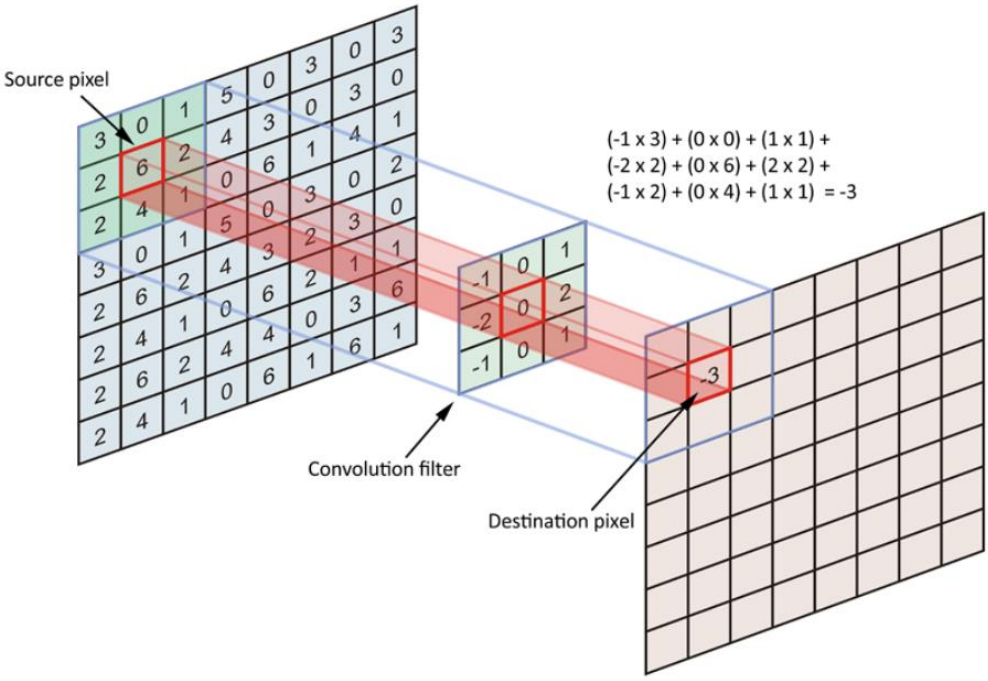

Figure 14: Convolution operation for one element of the feature map (pink, right)

From Figure 14, the convolution operation between the input matrix (blue, left) and kernel matrix (green, middle) produces the feature map (pink, rght). The kernel matrix "slides" over the input matrix and an element-wise multiplication is applied to the overlapping elements, then the resulting elements are summed to form each entry of the feature map. The values on the kernel matrix are known as the weights of the kernel matrix. These weights are determined via backpropagation (to be covered in Section 4.4.) iteratively to obtain a model that makes accurate predictions. How much the kernel matrix "slides" depends on a parameter called stride. A stride of 1 means that the kernel matrix slides over 1 pixel each time. Increasing the stride to 2 or more will result in a feature map of smaller dimensions. A stride of (2, 2) is interpreted as a stride of 2 along the first and second axis of the input matrix.

4.3.2. Activation Function

After every convolution operation, an activation function is applied to the output. In order to introduce non-linearity into the CNN, an activation function known as the Rectified Linear Unit (ReLU) is commonly used. The ReLU function is applied to each element in the output of the convolution layer. It replaces all negative values with zero.

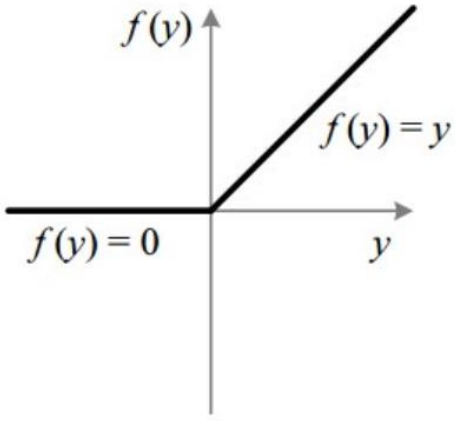

Figure 15: Graph of ReLU activation function [26]

Mathematically, the ReLU function can be expressed as:

$$ReLU(x) = \begin{cases} 0, & if\ x < 0 \\ x, & otherwise \end{cases}$$

There are also other activation functions available, such as the sigmoid function, hyperbolic tangent function, and many more. Other functions are used when certain types of output are expected or when for neural networks that accomplish specific purposes.

4.3.3. Pooling Layers

The pooling layers down-sample the data by reducing the dimensionality of the input feature maps. The most commonly used type of pooling layer in a CNN is the max pooling layer, which extracts the maximum value in a given pooling window.

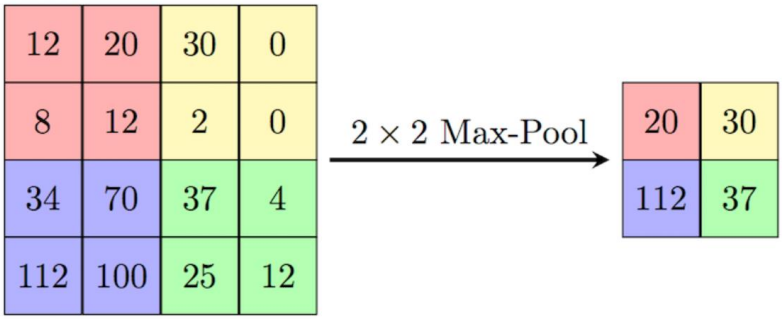

Figure 16: Max pooling operation [27]

First, a spatial neighbourhood is defined by a a 2×2 window (Figure 16, right). Starting from the top left corner of the input feature map (Figure 16, left), the maximum value in the window is retained, then the window slides to adjacent, non-overlapping elements along the row (for the case where the stride is 2×2). This operation is then repeated on the next non-overlapping row.

Reducing the feature dimension significantly reduces the number of weights that have to be trained by the CNN during the training step, resulting in shorter computational times. Reducing the feature dimensions also makes the CNN less likely to overfit.

4.3.4. Fully Connected Layer

The neurons of a fully connected layer are connected to all the neurons from the previous layer. The previous convolution and pooling layers extract several high-level features from the initial input. The fully connected layer brings together all the information to generate a hypothesis for the CNN. For a CNN used for classification, the final layer is normally a fully connected layer using a Softmax activation function, which assigns probabilities to each class in a multi-class classification problem.

4.4. Training

The training step is the process that provides the neural network model the capability to make predictions. Through training, the neural network learns the necessary features that allows it to make predictions.

Section 4.4. uses a fully connected ANN to illustrate and explain the concept of training a model. While the specific mathematical formulations on how the inputs are transformed as they are passed through the model differs from the types of neural network architectures, the mechanism by which the weights are updated are fairly similar. An ANN is used as an example because of its simplicity.

4.4.1. Forward Propagation

The input is passed through the network layers, with transformations applied to the input as it passes through each layer. This is known as a forward propagation.

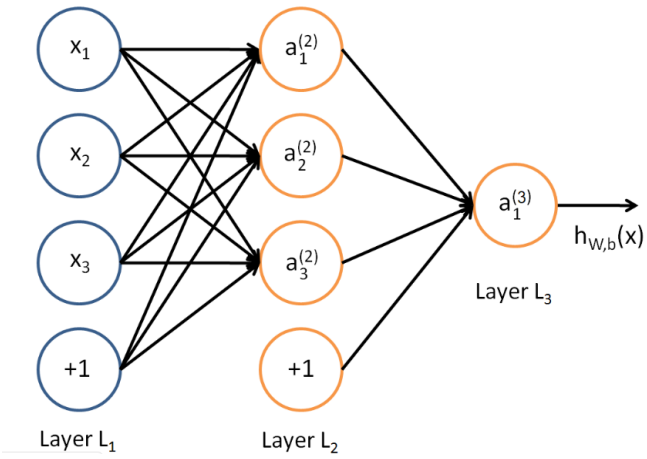

Figure 17: Forward propagation [28]

This section will illustrate a simple forward propagation process. Figure 17 shows a 3-layer neural network with inputs X_1, X_2 and X_3 (represented by vector X). The $+1$ nodes are known as bias nodes (subsequently denoted by X_0) and the arrows represent weights (subsequently denoted by θ). The weights are initialised randomly before the first forward propagation.

Mathematically, the input vector and the vector of Layer L_2 are as follows:

$$X = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}, \qquad A^{(2)} = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}$$

The weights for Layer L_1 will be a 3×4 matrix (3 outputs from 4 inputs), given as follows:

$$\theta^{(1)} = \begin{bmatrix} \theta_{10}^{(1)} & \theta_{11}^{(1)} & \theta_{12}^{(1)} & \theta_{13}^{(1)} \\ \theta_{20}^{(1)} & \theta_{21}^{(1)} & \theta_{22}^{(1)} & \theta_{23}^{(1)} \\ \theta_{30}^{(1)} & \theta_{31}^{(1)} & \theta_{32}^{(1)} & \theta_{33}^{(1)} \end{bmatrix}$$

Where $\theta_{nm}^{(z)}$ refers to the arrow pointing to node $a_n^{(2)}$ from node X_m and z represents the layer that the weights vector is operating on. $\theta_{10}^{(1)}$ (highlighted red in Figure 18) represents the weight that operates on X_0 that contributes to a portion of the value of $a_1^{(2)}$.

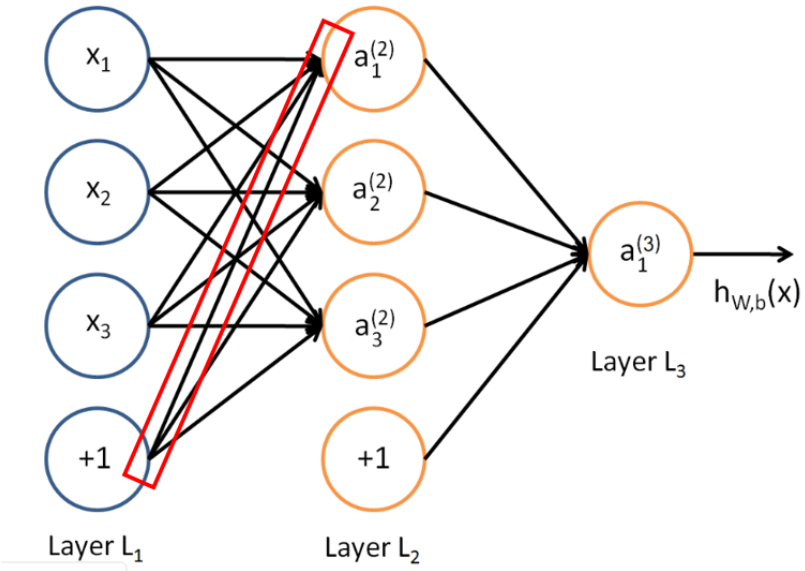

Figure 18: Highlighted weight representing $\theta_{10}^{(1)}$

Given an activation function g, the relationship between the vectors X and $A^{(2)}$ is given as

$$A^{(2)} = g(\theta^{(1)}X)$$

Such that

$$a_1^{(2)} = g\left(\theta_{10}^{(1)}X_0 + \theta_{11}^{(1)}X_1 + \theta_{12}^{(1)}X_2 + \theta_{13}^{(1)}X_3\right)$$
$$a_2^{(2)} = g\left(\theta_{20}^{(1)}X_0 + \theta_{21}^{(1)}X_1 + \theta_{22}^{(1)}X_2 + \theta_{23}^{(1)}X_3\right)$$
$$a_3^{(2)} = g(\theta_{30}^{(1)}X_0 + \theta_{31}^{(1)}X_1 + \theta_{32}^{(1)}X_2 + \theta_{33}^{(1)}X_3)$$

Given another activation function f, the relationship between vectors $A^{(2)}$ and $A^{(3)}$ is given by

$$A^{(3)} = f(\theta^{(2)}A^{(2)})$$

Such that

$$h_{w,b}(x) = a_1^{(3)} = f(\theta_{10}^{(2)}a_0^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}a_2^{(2)} + \theta_{13}^{(2)}a_3^{(2)})$$

Where $h_{w,b}(x)$ is the output of the forward propagation of the model.

4.4.2. Backpropagation

Since the weights are initialised randomly before the first forward propagation, there will be an error between the predicted output of the model from the first forward propagation and the corresponding ground truth values. This error is distributed backwards throughout the layers of the network to compute a gradient value that is needed to re-adjust the values of the weights in a process called backpropagation.

4.4.1.1. Cost Function

As each input has a corresponding ground truth output that was fed into the neural network model, the transformed input at the end of the layers is compared with the ground truth output to determine the amount of deviation. Minimising this deviation is the goal of machine learning problems since it will mean that predictions are closer to the ground truth and hence, more accurate. The function that is used to quantify this deviation is known as a cost function (or loss function) and the value computed by the cost function is known as the loss. Common cost functions used in regression problems include MSE and mean absolute error (MAE) while classification problems usually make use of cross-entropy loss. The loss values will determine the magnitude at which the weights will be re-adjusted using an optimiser.

4.4.1.2. Optimisers

The main goal of optimisers is to minimise the cost function, making the model's predictions closer to the ground truth output. This section aims to explain the concept of optimisers using the gradient descent optimiser, which many new optimisers are based on.

Suppose there is a simple hypothesis function where $y = h(x)$ and y is related to x by a single weight W, then a possible cost function $J(W)$ is shown in Figure 19.

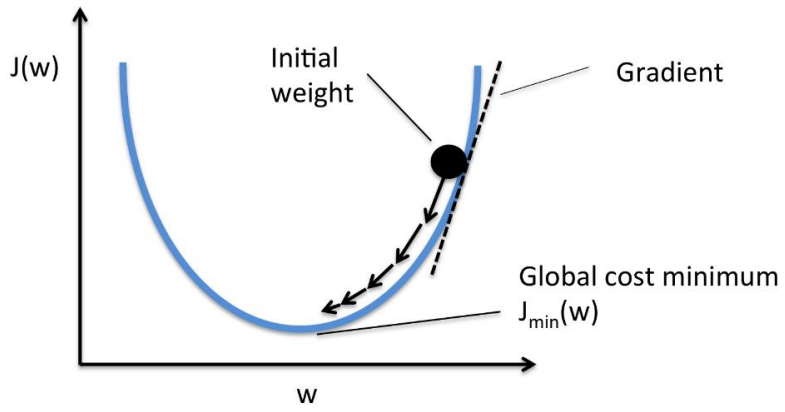

Figure 19: Gradient descent

The mathematical formulation for this example is given as:

$$W \leftarrow W - \alpha \frac{d}{dW} J(W)$$

Where W is the weight, $J(W)$ is the cost function and α is the learning rate, a hyperparameter required by the model. The $\leftarrow$ symbol means that the previous value of W will be replaced by the calculated by the right-hand side of the expression each iteration. This is how the weights are updated.

The negative of the derivative of $J(W)$ will "point" in the direction of the minima. In the next iteration, the weight W will move towards the left along the curve in this example. It is noted that the magnitude which the weight W shifts is proportional to the magnitude of $\frac{dJ(W)}{dw}$ at that point and the learning rate used. As the weight W approaches the value that minimises $J(W)$, the value of $\frac{dJ(W)}{dw}$ becomes smaller and thus, the adjustments become smaller and smaller. The computation of $\frac{dJ(W)}{dw}$ is the gradient that was previous referred to in Section 4.4.2. and is the essence of backpropagation.

For a hypothesis with more weights, the mathematical formulation is as follows:

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

This expression is very similar to the one above, the difference being θ_j are matrices containing more than 1 weight values, such as the one exemplified in Section 4.4.1.

The loss is expected to decrease with each iteration as the weights are updated and predictions made by the model are expected to become more accurate. Over the years, there are newer optimisation algorithms such as Adam, Adadelta, AdaGrad, RMSprop and more [29], each with their strengths and weaknesses.

4.4.3. Batch Size, Epochs and Iterations

Table 1 provides an explanation on some of the hyperparameters that control the way the training data is fed into the network.

Name	Description
Batch size	Number of training sets used in a single forward and backward pass.
Epochs	Number of times the algorithm sees the entire data set.
Iterations	Number of forward and backward passes of the specified batch size.

Table 1: Definitions of batch size, epochs and iterations

For example, 1000 training samples to be trained with a specified batch size of 100 will take 10 iterations to complete 1 epoch.

For a fixed training sample size, there is a trade-off between the batch size and the number of iterations. A smaller batch size means that the gradients are computed a larger number of times. For example, a batch size of 1 for 1000 samples mean that the gradients are calculated 1000 times. This can lead to faster convergence of the optimisation algorithm. A larger batch size improves computational speed at the expense of using more memory. For example, a batch size of 1000 for 1000 samples mean that the gradients are calculated 1 time, which results in faster computation times. The optimal value to use for the batch size is normally done experimentally, however, past research has shown that the use of a smaller batch size has not only led to faster optimiser algorithm convergence, but also better accuracy [30].

4.4.4. Prediction

With the final set of parameters (weights and biases), the neural network model is now trained and is ready to make predictions. The input matrix will undergo a series of transformations

determined by the values of the parameters and becomes the model's prediction. If the neural network is well-trained, then this prediction will be close to the ground truth.

While the training of the neural network model may require a lot of time depending on the size of the inputs and number of layers in the model, it is only done before the model is deployed for use. The prediction step is very fast and can be done in real-time as it only involves simple matrix operations.

4.5. Overfitting and Underfitting

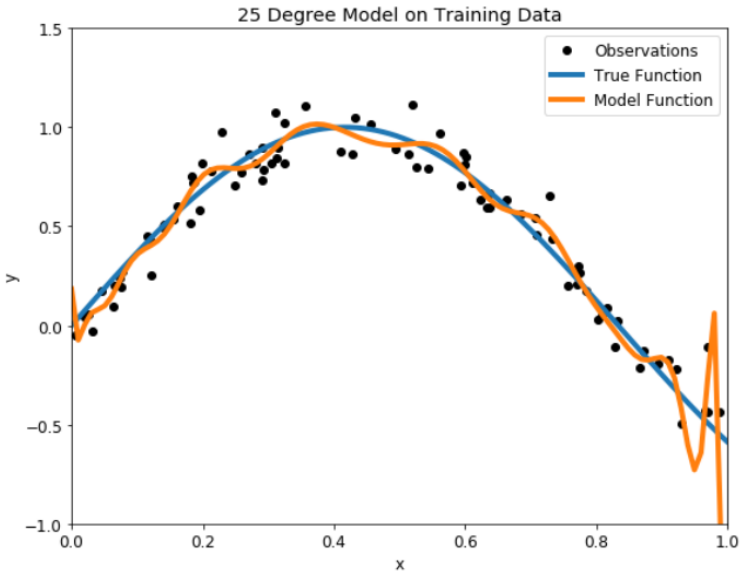

Figure 20: Overfitting of the training data [31]

A model may give a 99% prediction accuracy during training, however, it could produce a prediction accuracy significantly lower when making predictions on data that the model that has not encountered before. This situation is known as overfitting.

Overfitting occurs when the model picks up the noise and fluctuations in the data to the extent where its ability to generalise is negatively impacted. The result of overfitting is obtaining a hypothesis function that approximates the training data set too well. Figure 20 shows an example of overfitting using a polynomial function whose order is too high for the approximation. The high degree polynomial hypothesis function has a high degree of flexibility and will try to account for every single training point.

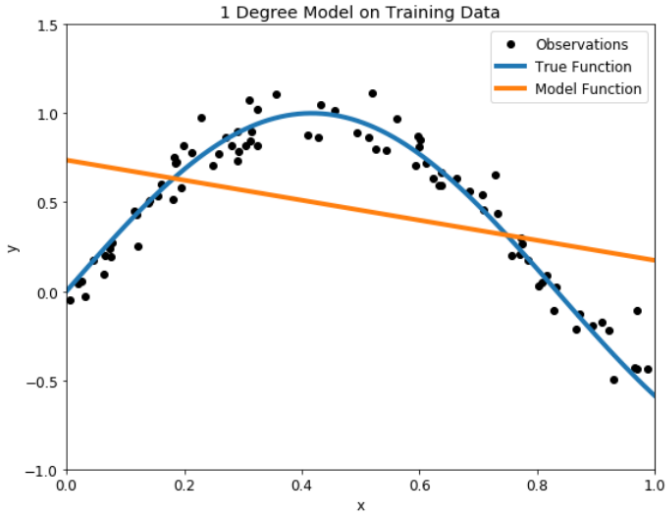

Figure 21: Underfitting of the training data [31]

On the other hand, the model underfits when it does not perform well during training. This occurs when the model is unable to capture the relationship between the inputs and their respective outputs. Figure 21 shows an example of underfitting using a polynomial function of order 1, which is too low for the approximation. The low degree polynomial hypothesis function has a low degree of flexibility and is unable to account for a sufficient number of training points.

Preventing overfitting and underfitting requires some experimentation. Overfitting is the more commonly encountered problem and is usually addressed through regularisation, adding dropout layers or increasing the data in the training set. Underfitting is usually addressed by using a different neural network architecture, adding more layers or increasing the number of nodes in current layers. Usually, in the rectification of overfitting or underfitting, a combination of techniques are used.

4.6. Dataset Split

The goal of a machine learning model is to generalise well on data that has not been encountered before and make accurate predictions based on these data. One commonly used technique applied to the data set is to split it into 3 categories: A training set, validation set and test set. Table 2 gives a brief description of each category, as well as the proportion of the overall data set is split for each category. The way the overall data set is split in Table 2 serves as a rough guideline and is not meant to be an optimum mix in general.

Name	Percentage of Data Set	Description
Training Set	70	The training set is the set of data that is actually used to train the model.
Validation Set	20	The validation set is used to evaluate how well the model generalises. The validation set is not used in the training of the model.
Test Set	10	The remaining data sets that were not used for training or validation. The optimally trained model has not encountered any of the data that comes from the test set and its performance should be a good indicator of its performance after being deployed for use.

Table 2: Definition of training set, validation set and test set

The loss (against epoch) based on the training set and validation set is used to diagnose possible issues with the model. Figure 22 shows both validation and training losses decreasing initially, then validation loss increasing while the training loss continues to decrease to a very low level. This divergence between the validation set loss and training set loss is an indicator of overfitting. As the model works on the set of data it trains on, it performs very well on the data it trained on but generalises worse on data that it has not encountered before.

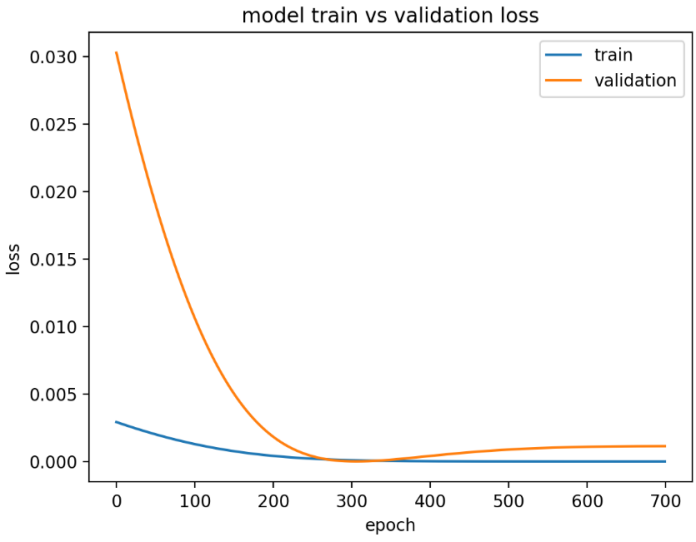

Figure 22: Loss curves indicating an overfit [32]

Figure 23 shows both validation set loss and training set loss decreasing initially, then plateauing off. Unlike the graph above, underfitting results in the model not performing well

even on the training set (the training set loss is about 0.1 in this case while the training set loss is close to 0 for the above scenario). This means that the model is too simple and is unable to capture any higher order details. As expected, the model also performs poorly on the validation set.

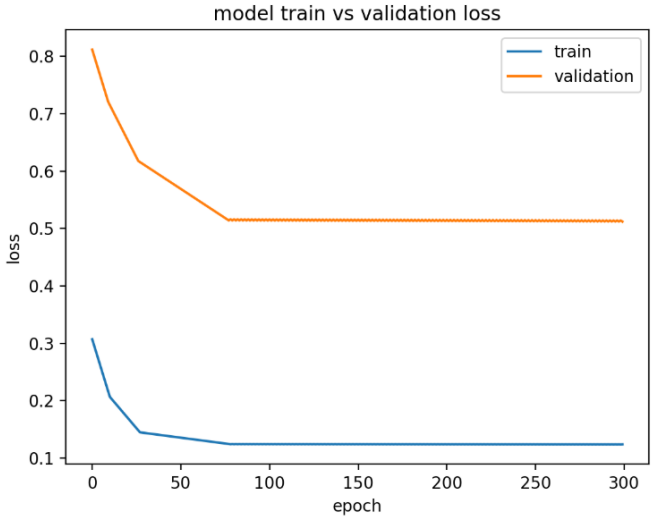

Figure 23: Loss curves indicating an underfit [32]

4.7. Machine Learning Frameworks

Writing your own code to implement the components and execute the steps mentioned in this section are extremely complicated and difficult, and possibly beyond the academic abilities of individuals who are just beginning to explore the field of AI.

The use of machine learning frameworks simplify the development of machine learning models. These frameworks consist of libraries, which are routines and functions that help developers perform complex tasks without the need to rewrite code. For example, in Keras, minor modifications such as changing the loss function from MSE to MAE is as simple as changing a function's argument to "MSE". These frameworks normally have commonly used components and functions built-in and users do not need to write complicated code to execute them. Popular frameworks in machine learning include: Scikit-learn, Tensorflow, Keras, Theano, CNTK and Caffe.

4.8. GPUs for Deep Learning

In building the models for Google Translate back in 2016, Google bought 2000 server-grade GPUs from Nvidia to accomplish hundreds of one-week Tensorflow runs, which otherwise would have taken months to converge [33].

CUDA (Compute Unified Device Architecture) is a parallel programming platform developed by Nvidia for use with its own GPUs. It speeds up compute-intensive applications by tapping on the power of GPUs for parallelisation sections of the computations.

Deep learning involves large amount of matrix computations which can be sped up by several orders through the use of GPU support. Currently, several frameworks (eg. Tensorflow, Keras, Theano, CNTK etc.) rely on CUDA for GPU support, using the CuDNN library for deep learning computations.

Chapter 5: Methodology

5.1. Defining Problem Statement

The project aims to build a CNN model to predict fluid flows around an obstacle. The model will be able to accept inputs containing information on the obstacle's geometry and output the corresponding fluid flow field parameters for a given initial condition. It is essentially solving the fluid flow problem without the numerical methods used by CFD solvers (Figure 24).

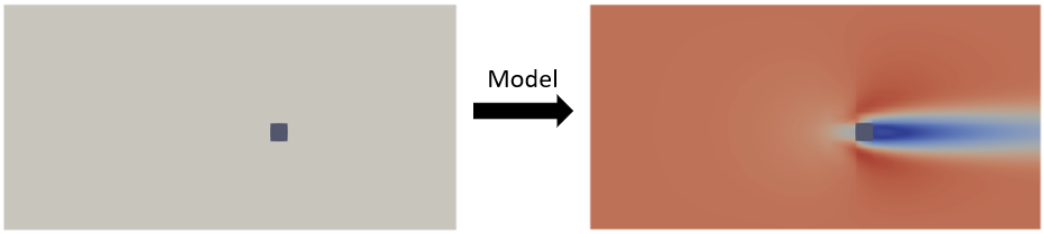

Figure 24: Using a model to predict fluid flow around an input geometry

This is done by training a CNN model on inputs and their corresponded labelled outputs in a supervised learning regression problem. The project will focus on the non-uniform, incompressible, steady laminar flow for a 2D geometry in a freestream flow at $0.3m/s$. A small freestream velocity is chosen to ensure that a steady state can be achieved for the simulations within a reasonable time.

5.2. Programs and Utilities

5.2.1. OpenFOAM

OpenFOAM (abbreviation for "Open source Field Operation and Manipulation") is a free, open source CFD software released and developed primarily by OpenCFD Ltd. It has an extensive range of features that are able to solve complex fluid flows involving chemical reactions, turbulence and heat transfer, to solid dynamics and electromagnetism problems.

For the project, OpenFOAM v1806 will be used to generate all the fluid simulation cases used for training and testing of the neural network. The commands are executed on an Ubuntu 18.04.1 LTS terminal on a Windows 10 Operating System. A third-party package, ParaView, will be used for graphical post-processing of the simulation data.

5.2.2. Keras

Keras is a high-level application programming interface (API) written in Python which uses a TensorFlow, Theano or CNTK backend. Its development is backed primarily by Google. Microsoft maintains the CNTK backend, Amazon AWS maintains the Keras fork with MXNet

support, with contributions from other technology companies such as Nvidia, Uber and Apple [34].

Keras is chosen mainly because of its capability to allow for quick and easy prototyping of neural network models. Keras has the second strongest adoption in the industry and research community, coming in after Tensorflow (and the Keras API is the official frontend of Tensorflow) [34]. Keras models are also trainable on various hardware platforms, including Nvidia GPUs.

5.3. Outline

In this machine learning problem, it is first required to determine the type of model to be used and the type of inputs to be used with the model. Figure 25 illustrates the decision process.

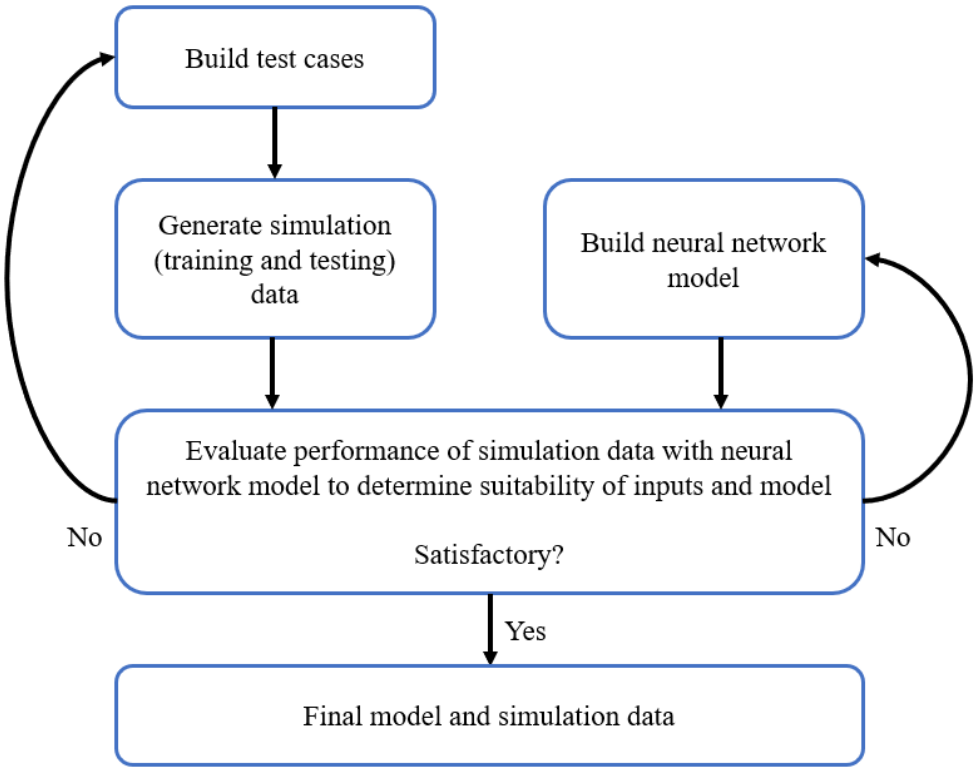

Figure 25: Flow chart to determine suitability of inputs and model

Several iterations are expected before finalising the structure of the input data and the final set of hyperparameters for the model. With a functioning basic model, further changes can be made to improve the performance of the model or provide new capabilities.

5.3.1. Building Case Files

The data generation step characterises the function of the neural network model and affects the type of information that can be extracted by the layers in the model. The CFD simulation cases

are built to emulate a wind tunnel test, where an obstacle is placed in a freestream fluid to study the flow of the fluid around the obstacle. Simple obstacle geometries are used for the CFD simulation in the generation of data. The “wind tunnel” will be a rectangular area of $25.5\times12.7m$ and the obstacle is an empty space located within the “wind tunnel”. The dimensions of the “wind tunnel” are not chosen arbitrarily, but to make the extraction of data and construction of the CNN models easier, especially for the encoder-decoder architectures discussed in Section 5.3.3.

This section illustrates how a case file is generated. It covers the case where the geometry is a regular shape and the case where the geometry is of a more complex shape. The process is repeated for all 61 data sets. All shapes generated in the data set are simple geometries such as squares, rectangles, L-shape figures and more, translated spatially to various locations.

5.3.1.1. Geometry Creation (Regular Geometries)

Figures 26 and 27 show schematics of a sample regular shaped geometry with a number label given to certain points. The order of labels is arbitrary and unimportant. The larger cuboid has a height of $12.7m$, length of $25.5m$ and width of $1m$. The regular geometry in the middle is represented by a void in the larger cuboid. A coordinate system is formed by mapping a cartesian grid with opposite ends (0, 0, 0) and (255, 127, 0) over the "wind tunnel”. The code snippet in Figure 28 is used to assign coordinates to the labelled points. The resulting shape is shown in Figures 29 and 30 when viewed using ParaView.

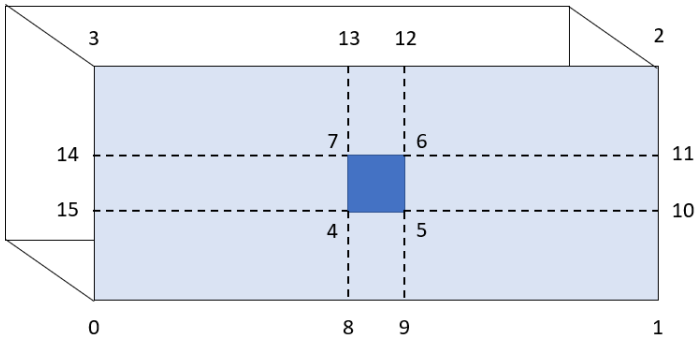

Figure 26: Coordinates numbering system (front)

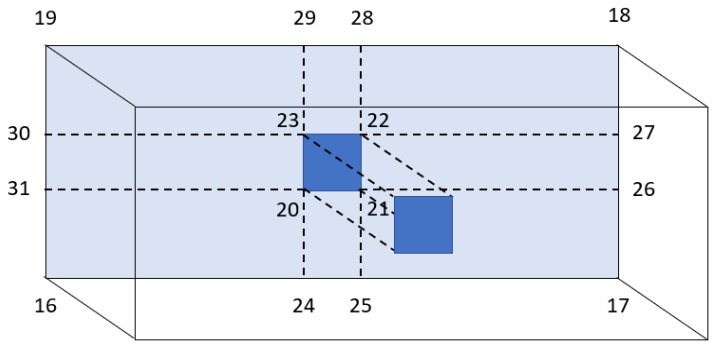

Figure 27: Coordinates numbering system (back)

```
code
#{
    pointField points(16);
    points[0]  = point(0, 0, 0.5);
    points[1]  = point(25.5, 0, 0.5);
    points[2]  = point(25.5, 12.7, 0.5);
    points[3]  = point(0, 12.7, 0.5);
    points[4]  = point(15, 5, 0.5);
    points[5]  = point(16, 5, 0.5);
    points[6]  = point(16, 6, 0.5);
    points[7]  = point(15, 6, 0.5);
    points[8]  = point(15, 0, 0.5);
    points[9]  = point(16, 0, 0.5);
    points[10] = point(25.5, 5, 0.5);
    points[11] = point(25.5, 6, 0.5);
    points[12] = point(16, 12.7, 0.5);
    points[13] = point(15, 12.7, 0.5);
    points[14] = point(0, 6, 0.5);
    points[15] = point(0, 5, 0.5);

    // Duplicate z points
    label sz = points.size();
    points.setSize(2*sz);
    for (label i = 0; i < sz; i++)
    {
        const point& pt = points[i];
        points[i+sz] = point(pt.x(), pt.y(), -pt.z());
    }

    os  << points;
```

Figure 28: Code snippet from directory: system\blockMeshDict

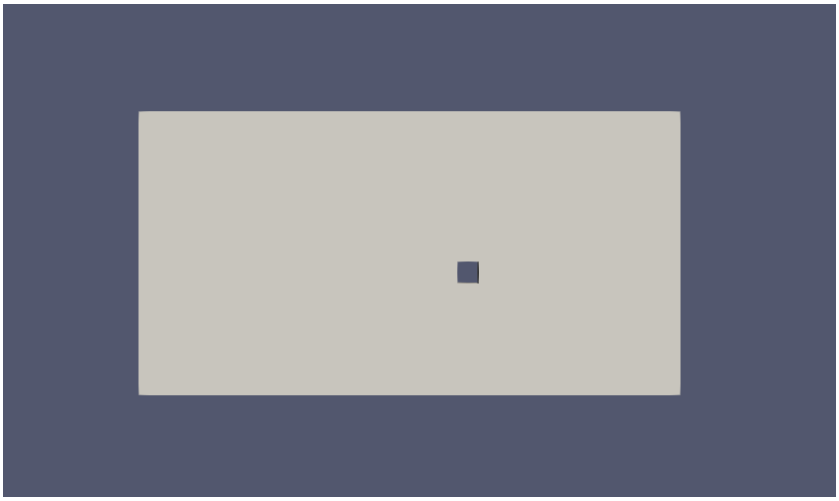

Figure 29: Front view of regular geometry

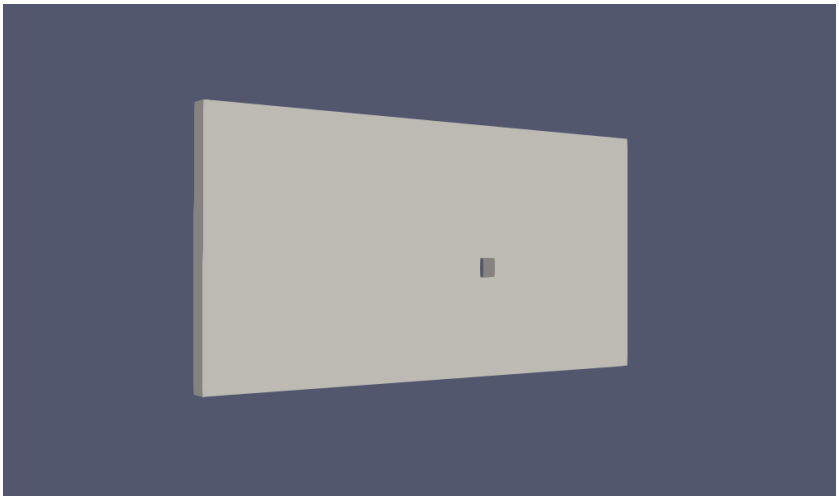

Figure 30: Secondary view of regular geometry

5.3.1.2. Geometry Creation (Irregular Geometries)

The process of using OpenFOAM's inbuilt dictionaries to create complex and irregular obstacle geometries can be extremely time-consuming and tedious. This may not be ideal when a large volume of data is required. Therefore, the complicated obstacle geometries will be created using ANSYS and the *.msh* files will then processed by the OpenFOAM meshing utilities and solver.

5.3.1.3. Mesh Generation (Regular Geometries)

The meshing of regular geometries is done using *blockMesh*, the mesh generation utility supplied by OpenFOAM. *blockMesh* reads the *blockMeshDict* (directory: *system\blockMeshDict*) dictionary and writes out the mesh data to other files. *blockMesh* decomposes the domain geometry into a set of 1 or more 3D hexahedral blocks, where each block is defined by 8 vertices at each corner of the hexahedron [35].

The code snippet in Figure 31 in the *blockMeshDict* dictionary controls the meshing size. The numbers in the brackets after hex define a particular block segment for the meshing; these numbers refer to the same points in Figure 26 and 27. The numbers in the next bracket define the resolution of the meshing. (20 10 1) means that there are 20 squares along the X-axis, 10 squares along the Y-axis and 1 square along the Z-axis in the block defined by (21 26 27 22 5 10 11 6). The *simpleGrading* (1 1 1) term creates meshes that are of the same size along all the axes. The *checkMesh* command is executed after the *blockMesh* command to verify that the geometry is properly meshed. Figures 32 and 33 show the meshed geometry viewed using ParaView.

```
60  blocks
61  (
62      hex (16 24 20 31 0 8 4 15) (20 20 1) simpleGrading (1 1 1)
63      hex (24 25 21 20 8 9 5 4) (10 20 1) simpleGrading (1 1 1)
64      hex (25 17 26 21 9 1 10 5) (20 20 1) simpleGrading (1 1 1)
65      hex (21 26 27 22 5 10 11 6) (20 10 1) simpleGrading (1 1 1)
66      hex (22 27 18 28 6 11 2 12) (20 20 1) simpleGrading (1 1 1)
67      hex (23 22 28 29 7 6 12 13) (10 20 1) simpleGrading (1 1 1)
68      hex (30 23 29 19 14 7 13 3) (20 20 1) simpleGrading (1 1 1)
69      hex (31 20 23 30 15 4 7 14) (20 10 1) simpleGrading (1 1 1)
70  );
```

Figure 31: Code snippet from directory: system\blockMeshDict

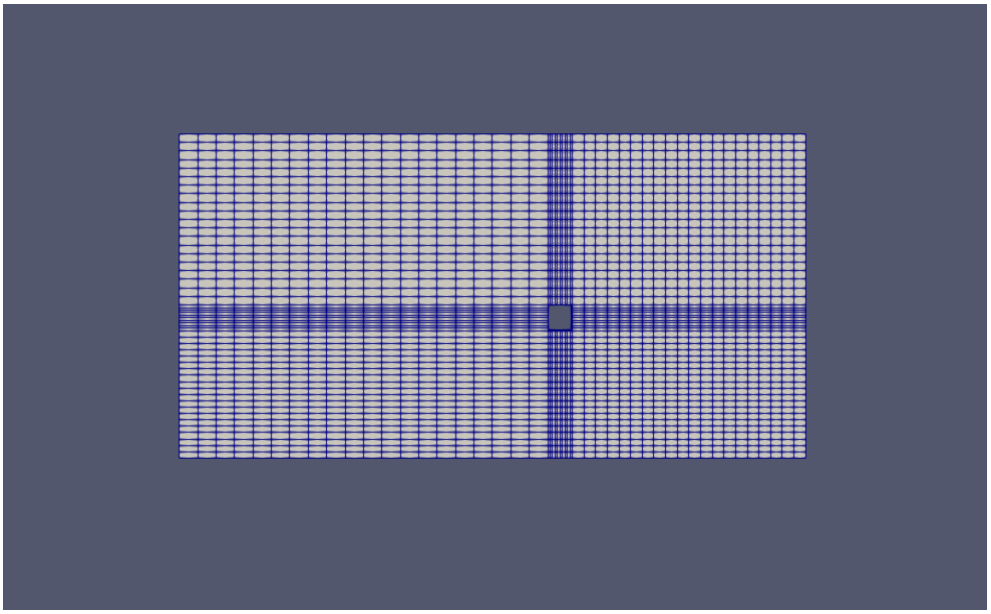

Figure 32: Front view of regular geometry after meshing

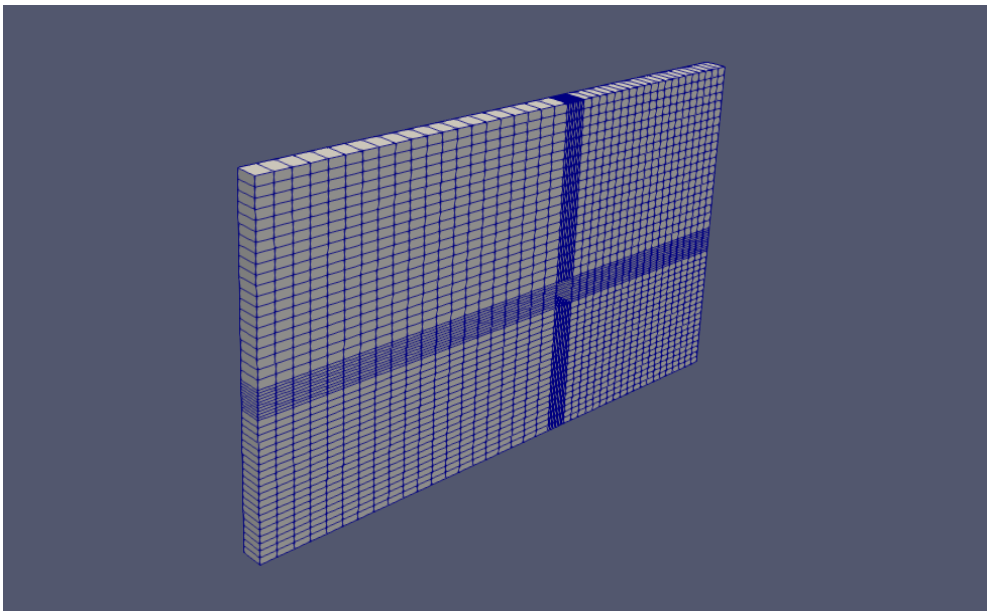

Figure 33: Secondary view of regular geometry after meshing

5.3.1.4. Mesh Generation (Irregular Geometries)

Irregular geometries are saved in *.msh* files and meshed using the *fluentMeshToFoam* mesh converter. The converter attempts to capture the *Fluent* boundary condition definition as much as possible, however, as there is no direct correspondence between the OpenFOAM and *Fluent* boundary conditions [36], minor amendments have to be made to the *boundary* (directory: *constant\polyMesh\boundary*) dictionary.

The *checkMesh* command is used to verify that the geometry is properly meshed. Figure 34 shows the meshed irregular geometry using ParaView.

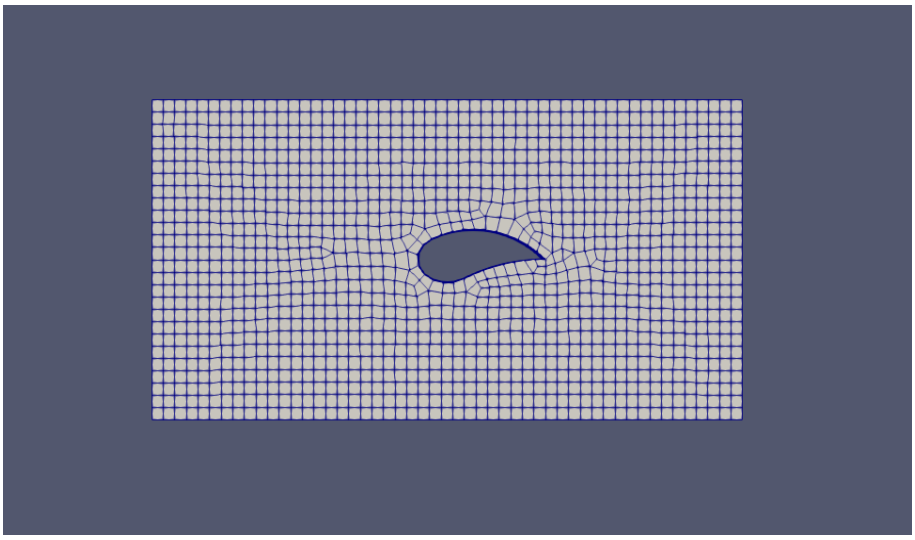

Figure 34: Front view of irregular geometry after meshing

5.3.1.5. Boundary and Initial Conditions

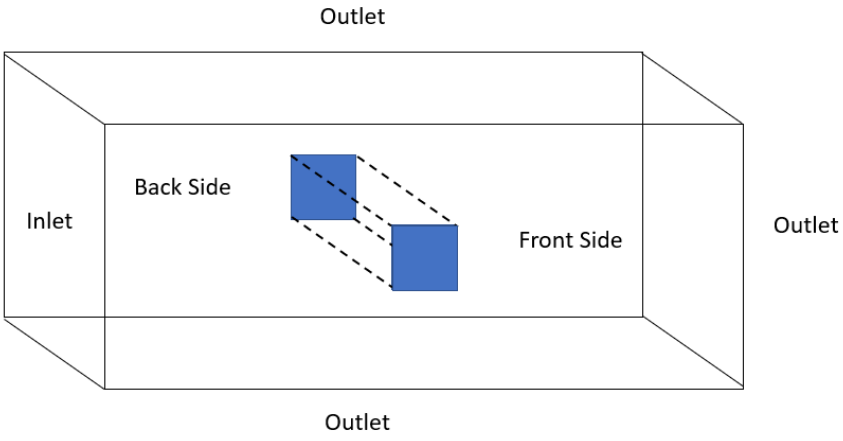

Figure 35: Faces of a typical geometry

The labels for the faces in Figure 35 will be used when identifying their associated boundary conditions. In addition, the 4 faces of the smaller cuboid that are inside the larger cuboid are labelled as *Obstacle*. Table 3 describes the boundary conditions for each of the faces.

Face	Variable	Constraint	Description
Inlet	Velocity	*fixedValue*	A fixed value constraint is applied
	Pressure	*zeroGradient*	A zero-gradient condition is applied
Outlet	Velocity	*zeroGradient*	A zero-gradient condition is applied
	Pressure	*fixedValue*	A fixed value constraint is applied
Front Side/ Back Side	Velocity	*empty*	Used for reduced dimensions cases (3D to 2D), applied to patches whose normal is aligned to geometric directions that do not constitute solution directions
	Pressure		
Obstacle	Velocity	*noSlip*	Fixes the velocity to be zero at the walls
	Pressure	*zeroGradient*	A zero-gradient condition is applied

Table 3: Boundary conditions applied to geometry

The initial conditions of the simulation are controlled by 2 dictionaries, one for the initial velocities in X, Y and Z directions and the other dictionary for the initial pressure conditions. The above conditions are applied in the *p* (directory: *0\p*) and *u* (directory: *0\u*) dictionaries. Snippets from both dictionaries are shown in Figures 36 and 37. The fluid used in the simulation has a kinematic viscosity of 0.01.

```
21  boundaryField
22  {
23      inlet
24      {
25          type            fixedValue;
26          value           uniform (0.3 0 0);
27      }
28
29      outlet
30      {
31          type            zeroGradient;
32      }
33
34      upAndBottom
35      {
36      type        empty;
37      }
38
39      obstacle
40      {
41          type            noSlip;
42      }
43  }
44
```

Figure 36: Code snippet from directory: 0\U

```
boundaryField
{
    inlet
    {
        type            zeroGradient;
    }

    outlet
    {
        type            fixedValue;
    value       uniform 0;
    }

    upAndBottom
    {
    type        empty;
    }

    obstacle
    {
        type            zeroGradient;
    }
}
```

Figure 37: Code snippet from directory: 0\p

5.3.1.7. Running of Simulations

The code snippet in Figure 38 from the *controlDict* (directory: *system\controlDict*) file provides the parameters for running the simulation. The simulation runs for 500*s* at a time step of 0.02*s* and outputs the simulation data in the folder every 10*s*.

```
application     icoFoam;

startFrom       startTime;

startTime       0;

stopAt          endTime;

endTime         500;

deltaT          0.02;

writeControl    timeStep;

writeInterval   500;

purgeWrite      0;

writeFormat     ascii;

writePrecision  6;

writeCompression off;

timeFormat      general;

timePrecision   6;

runTimeModifiable true;
```

Figure 38: Code snippet from directory: system\controlDict

The Courant-Friedrichs-Lewy (CFL) condition is necessary for convergence while solving certain partial differential equations. For a one-dimensional case, the CFL has the following form:

$$C = \frac{u\Delta t}{\Delta x}$$

Where C is the Courant number, u is the magnitude of the velocity, Δt is the time step and Δx is the length interval (which is controlled by the mesh). For the OpenFOAM solver to converge to a solution, the Courant number has to be less than 1. The parameters in the *controlDict* file are chosen such that the Courant number is always less than 1 for all the simulation cases in this project.

The simulation is ran using OpenFOAM's *icoFoam* solver. *icoFoam* solves the incompressible Navier-Stokes equations using the Pressure Implicit with Splitting of Operators (PISO) algorithm. The algorithm works in the following steps [37]:

1. Define the boundary conditions
2. Compute an intermediate velocity field by solving the discretised momentum equation
3. Compute the mass fluxes at the cell faces
4. Solve the pressure equation
5. Correct the mass fluxes at cell faces
6. Correct the velocities based on the new pressure field
7. Update the boundary conditions
8. Repeat from Step 3 for a prescribed number of times
9. Increase the time step and repeat from Step 1

```
Time = 500

Courant Number mean: 0.0117668 max: 0.0753621
smoothSolver:  Solving for Ux, Initial residual = 2.2028e-08, Final residual = 2.2028e-08, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 1.08079e-07, Final residual = 1.08079e-07, No Iterations 0
DICPCG:  Solving for p, Initial residual = 8.39408e-07, Final residual = 8.39408e-07, No Iterations 0
time step continuity errors : sum local = 2.59679e-12, global = 1.27011e-13, cumulative = -3.798e-08
DICPCG:  Solving for p, Initial residual = 9.96213e-07, Final residual = 9.96213e-07, No Iterations 0
time step continuity errors : sum local = 3.08188e-12, global = 1.38947e-13, cumulative = -3.79799e-08
ExecutionTime = 92.65 s  ClockTime = 158 s
```

Figure 39: Last time step of OpenFOAM cased solved using icoFoam

Figure 39 shows the computational steps involved in the final time step of the simulation. The maximum Courant number is 0.075362, which is safely below the maximum allowable value of 1. This means that the settings in Figure 38 can be used without worrying that subsequent

simulations may result in a maximum Courant number above 1. Figure 40 displays the velocity field around the geometry at 500s using ParaView.

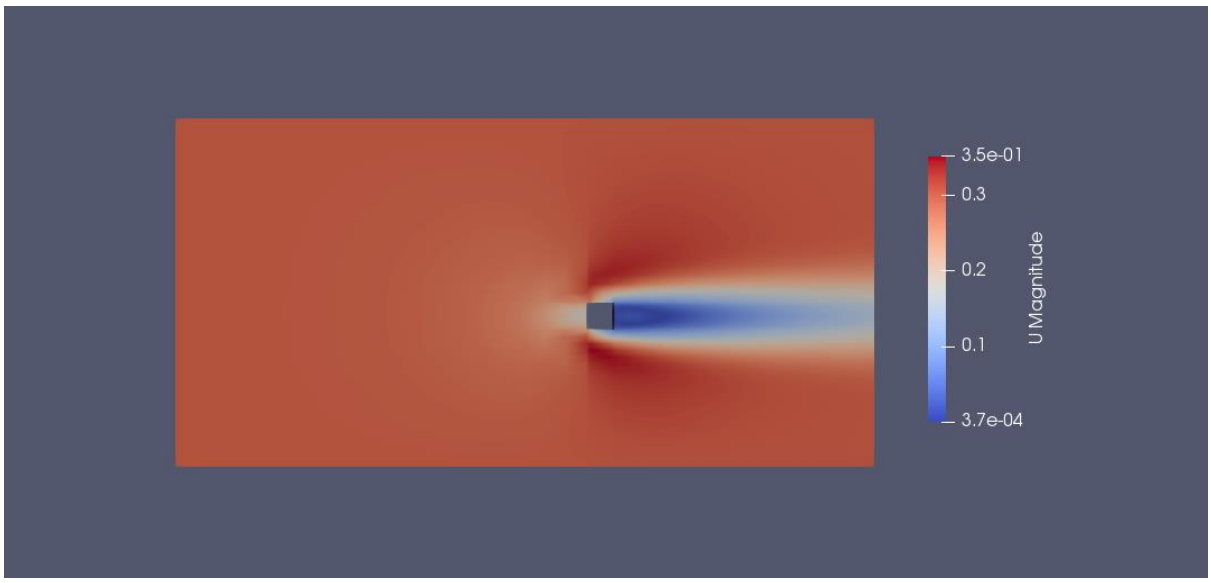

Figure 40: Velocity field at 500s, plot using ParaView

Once the simulation is complete, the velocities and pressure values of the midplane containing edges (0, 0, 0), (0, 12.7, 0), (25.5, 0, 0) and (25.5, 12.7, 0) are written to a *.xlsx* file using the *postProcess -func sampleDict* command. The final output are the velocities (U_x, U_y, U_z) and pressures of every coordinate for every 10s time step for 500s after the start of the simulation. Each time step consists of 256×128 sets of velocities (U_x, U_y, U_z) and pressure data.

5.3.2. Model Inputs

This step involves extracting the necessary values from the simulation data to form the inputs to the neural network model.

The focus of the machine learning problem is to derive a relationship between an output simulation field to an input geometry and initial condition. Therefore, the *.xlsx* files that will be used are those at time 0s and 500s. For each *.xlsx* file, the velocities at each coordinate are extracted and arranged in a *numpy* array with arguments (height, width, velocities) and shape (128, 256, 3). The *numpy* arrays extracted from the files from 0s will form the training inputs into the model and the *numpy* arrays extracted from the files from 500s will form the corresponding training outputs of the model.

Data pre-processing methods such as normalisation are not used as the magnitudes of the elements in the inputs and outputs are of similar order. Normalisation is not expected to enhance the performance of the model.

Even though the data set size is relatively small, data augmentation methods normally associated with CNNs are not used in this project. The reason for this is that changing the position or orientation of the obstacle results in an entirely new flow scenario and the augmented data may not be valid.

5.3.2.1. Encoding Geometry Information

Extracting the velocity field at time $0s$ results in a *numpy* array of mostly zeroes (due to the way the *numpy* arrays were initialised). This means that all the training inputs do not encode any information on the obstacle geometry to be used in the simulation. To overcome this problem, the areas in the input *numpy* array grid that corresponds to the obstacle will be given a value of -1. The value of -1 is chosen because it is expected that the other values in the velocity field will not come close to -1.

Figure 41 shows how this will appear on Spyder's Variable explorer function. The obstacle is shown in red, with a value of -1 assigned to the associated cells. The figure is only a partial snapshot of the velocity fields surrounding the geometry and the geometry itself. Since there is a mapping from the cartesian coordinates of size $12.7 \times 25.5m$ to an imaginary array of 128×256, there is a possibility where the shape is not captured perfectly due to the resolution used for the simulations and data extraction.

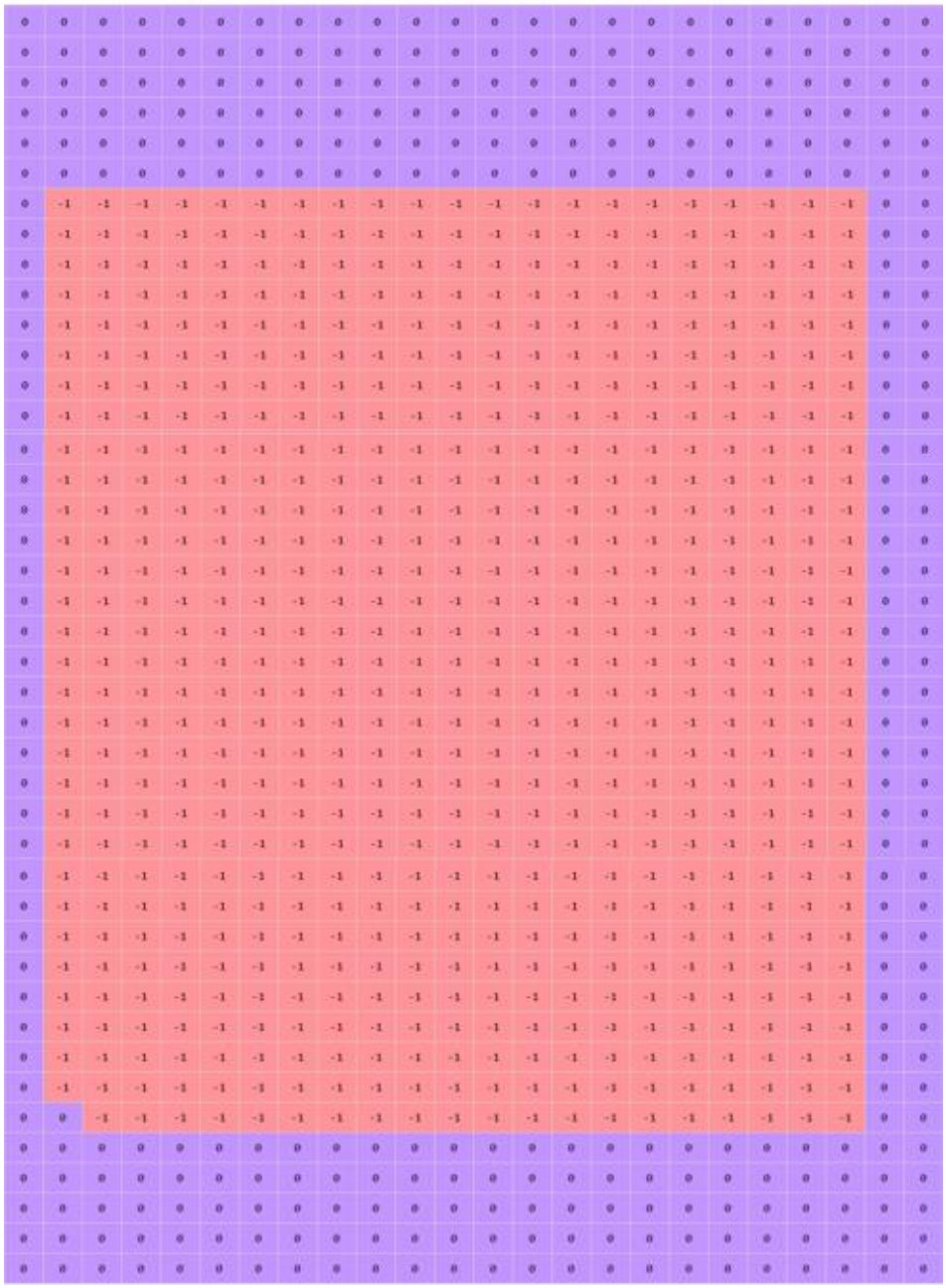

Figure 41: Sample (partial) of how geometry information (red) is encoded

5.3.3 Model

5.3.3.1. Architecture

There are many possible ways to approach the machine learning regression problem specified in Section 5.1. This section discusses the 3 architectures that will be explored in this project.

5.3.3.1.1. CNNModel 1 – fullCon

The first model is a FCN with unstrided convolutions, where the dimensions of the inputs and outputs are the same for every layer. The overall model (referred to as CNNModel 1) contains 2 CNNs. One CNN (*fullConXVelocity*) makes a prediction on the X velocity and the other CNN

(*fullConYVelocity*) makes a prediction on the Y velocity. Figure 42 provides an illustration of how the network is organised. Both CNNs in CNNModel 1 will have the same architecture. There will be a total of 2 convolution layers and 4 deconvolutional layers. The arguments of each convolution layer are given in Table 4.

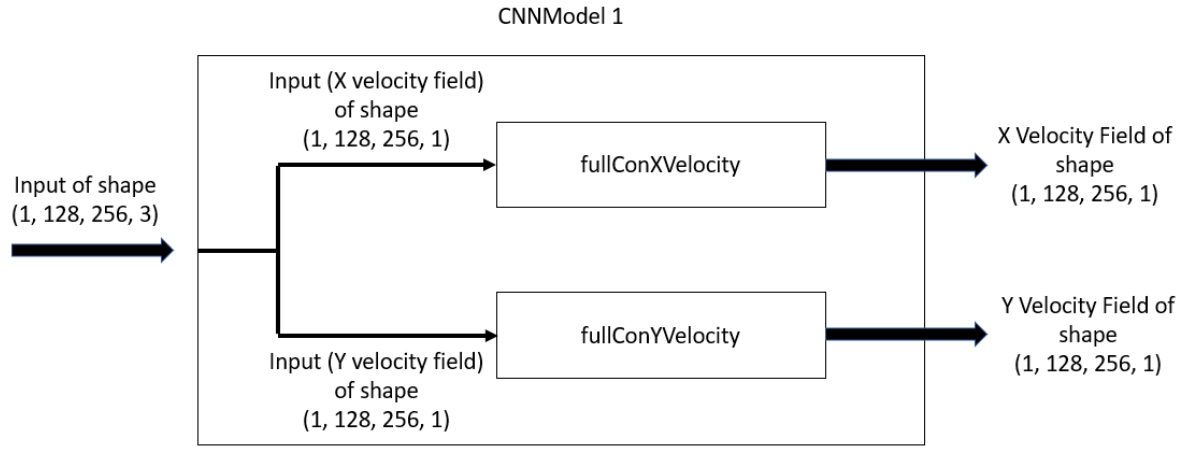

Figure 42: Schematic of CNNModel 1

Layer	Type	Filters	Kernel Size	Stride	Padding	Activation Function
1	Convolution	128	(8, 16)	0	Yes	ReLU
2	Convolution	512	(4, 4)	0	Yes	ReLU
3	De-convolution	512	(8, 8)	0	Yes	ReLU
4	De-convolution	256	(4, 8)	0	Yes	ReLU
5	De-convolution	32	(2, 2)	0	Yes	ReLU
6	De-convolution	1	(2, 2)	0	Yes	Linear

Table 4: Summary of layers for CNNModel 1

Some elements in the ground truth outputs are negative values. Using a ReLU activation function will cause the negative values to be 0. A Linear activation function outputs values from $-\infty$ to ∞ and will be a more suitable activation function for the last layer as the ground truth output arrays contain negative values (which represent flow towards the left) as well.

Using *same* as the argument of *padding* ensures that the output of each convolution layer has shape of (128, 256, n) where n is the number of filters used for that layer. This can be seen more clearly with a model summary object (Figure 43). The summary shows the shape of the output of each layer and the number of trainable and non-trainable parameters. The summary for both CNNs are the same. The code snippet in Figure 44 shows how the layers are defined

for *fullConXVelocity* using Keras' Sequential object. The layers for *fullConYVelocity* are defined in the same way.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 128, 256, 128)     16512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 128, 256, 512)     1049088
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 128, 256, 512)     16777728
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 128, 256, 256)     4194560
_________________________________________________________________
conv2d_transpose_11 (Conv2DT (None, 128, 256, 32)      32800
_________________________________________________________________
conv2d_transpose_12 (Conv2DT (None, 128, 256, 1)       129
=================================================================
Total params: 22,070,817
Trainable params: 22,070,817
Non-trainable params: 0
_________________________________________________________________
```

Figure 43: Model summary for the fullConXVelocity

```
cnn = Sequential()
cnn.add(Conv2D(128, (8,16), padding='same', input_shape=(128, 256, 1), activation='relu'))
cnn.add(Conv2D(512, (4,4), padding='same', activation='relu'))

cnn.add(Conv2DTranspose(512, (8,8), padding='same', activation='relu'))
cnn.add(Conv2DTranspose(256, (4,8), padding='same', activation='relu'))
cnn.add(Conv2DTranspose(32, (2,2), padding='same', activation='relu'))
cnn.add(Conv2DTranspose(1, (2,2), padding='same', activation='linear'))
```

Figure 44: Code snippet from model.py (CNNModel 1)

5.3.3.1.2. CNNModel 2 – encoderDecoderNoDense

The second model is also a FCN, but it uses strided convolutions, resulting in an encoder-decoder architecture. The overall model (referred to as CNNModel 2) contains 2 CNNs. One CNN (*encoderDecoderNoDenseXVel*) makes a prediction on the X velocity and the other CNN (*encoderDecoderNoDenseYVel*) makes a prediction on the Y velocity. Figure 45 provides an illustration of how the network is organised. Both CNNs in CNNModel 2 will have the same architecture. There will be a total of 2 convolution layers and 4 deconvolutional layers. Here, the *stride* argument is no longer 0. This means the output of each convolution layer has smaller dimensions than the inputs and the output of each deconvolution layer has larger dimensions than their inputs. The arguments of each convolution layer are given in Table 5.

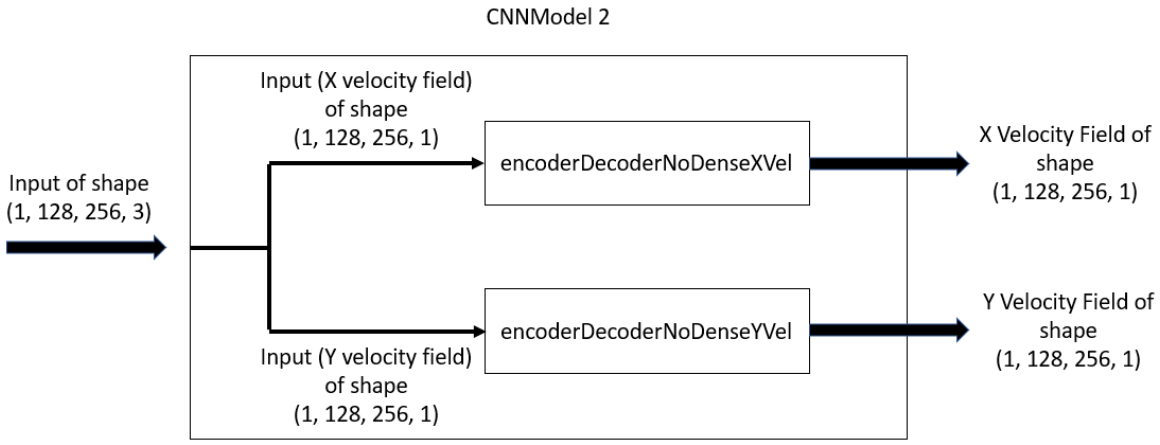

Figure 45: Schematic of CNNModel 2

Layer	Type	Filters	Kernel Size	Stride	Padding	Activation Function
1	Convolution	128	(8, 16)	(8, 16)	None	ReLU
2	Convolution	512	(4, 4)	(4, 4)	None	ReLU
4	De-convolution	512	(2, 2)	(2, 2)	None	ReLU
5	De-convolution	256	(4, 8)	(4, 8)	None	ReLU
6	De-convolution	32	(2, 2)	(2, 2)	None	ReLU
7	De-convolution	1	(2, 2)	(2, 2)	None	Linear

Table 5: Summary of layers for CNNModel 2

The summary in Figure 46 shows the shape of the output of each layer and the number of trainable and non-trainable parameters. The summary for both CNNs are the same. The code in Figure 34 shows how the layers are defined for *encoderDecoderNoDenseXVel* using Keras' Sequential object. The layers for *encoderDecoderNoDenseYVel* are defined in the same way.

```
1
2  Layer (type)                 Output Shape              Param #
3  =================================================================
4  conv2d_109 (Conv2D)          (None, 16, 16, 128)       16512
5
6  conv2d_110 (Conv2D)          (None, 4, 4, 512)         1049088
7
8  conv2d_transpose_217 (Conv2D (None, 8, 8, 512)         1049088
9
10 conv2d_transpose_218 (Conv2D (None, 32, 64, 256)       4194560
11
12 conv2d_transpose_219 (Conv2D (None, 64, 128, 32)       32800
13
14 conv2d_transpose_220 (Conv2D (None, 128, 256, 1)       129
15 =================================================================
16 Total params: 6,342,177
17 Trainable params: 6,342,177
18 Non-trainable params: 0
19
```

Figure 46: Model summary for the encoderDecoderNoDenseXVel

```
295     cnn = Sequential()
296     cnn.add(Conv2D(128, (8,16), strides= (8,16), input_shape=(128, 256, 1), activation='relu'))
297     cnn.add(Conv2D(512, (4,4), strides= (4,4), activation='relu'))
298
299     cnn.add(Conv2DTranspose(512, (2,2), strides= (2,2), activation='relu'))
300     cnn.add(Conv2DTranspose(256, (4,8), strides= (4,8), activation='relu'))
301     cnn.add(Conv2DTranspose(32, (2,2), strides= (2,2), activation='relu'))
302     cnn.add(Conv2DTranspose(1, (2,2), strides= (2,2), activation='linear'))
```

Figure 47: Code snippet from model.py (CNNModel 2)

5.3.3.1.3. CNNModel 3 – encoderDecoderWithDense

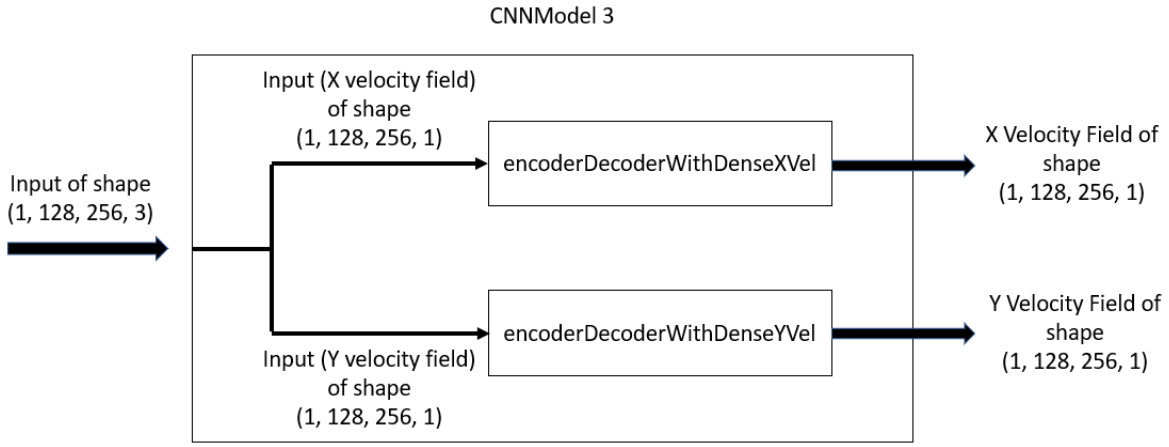

Figure 48: Schematic of CNNModel 3

Layer	Type	Filters/ Units	Kernel Size	Stride	Padding	Activation Function
1	Convolution	128	(8, 16)	(8, 16)	None	ReLU
2	Convolution	512	(4, 4)	(4, 4)	None	ReLU
3	Flatten	-	-	-	-	-
3	Dense	1024	-	-	-	Hyperbolic Tangent
3	Reshape	-	-	-	-	-
4	De-convolution	512	(8, 8)	(8, 8)	None	ReLU
5	De-convolution	256	(4, 8)	(4, 8)	None	ReLU
6	De-convolution	32	(2, 2)	(2, 2)	None	ReLU
7	De-convolution	1	(2, 2)	(2, 2)	None	Linear

Table 6: Summary of layers for CNNModel 3

The third model uses strided convolutions in an encoder-decoder architecture with a fully connected layer between the convolution and deconvolution layers. The overall model (referred to as CNNModel 3) contains 3 CNNs. One CNN (*encoderDecoderWithDenseXVel*) makes a

prediction on the X velocity and the other CNN (*encoderDecoderWithDenseYVel*) makes a prediction on the Y velocity. Figure 48 provides an illustration of how the network is organised.

The arguments of each layer are given in Table 6. Both CNNs will have the same architecture. There will be a total of 2 convolution layers and 4 deconvolutional layers. However, in CNNModel 3, a fully connected (*Dense*) layer will be inserted between the convolution and deconvolution layer. The *Flatten* layer transforms the input into a column (Figure 49) so that its output can be inputs into the *Dense* layer that follows. The *Reshape* layer does the reverse of *Flatten* so its output can become inputs into the deconvolution layers that follow. Similar to CNNModel 2, the *stride* argument is not 0.

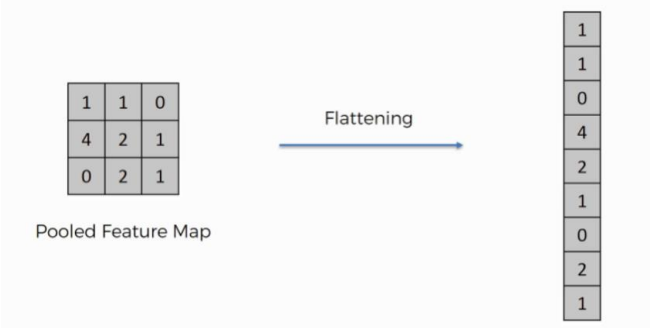

Figure 49: Flatten operation [38]

The *Dense* layer uses the hyperbolic tangent as its activation function. Figure 50 is a graphical comparison between the sigmoid and hyperbolic tangent (tanh) activation functions. The sigmoid function is a popular activation function normally used with fully connected layers, however, the hyperbolic tangent was chosen to prevent the negative values (if any) in the feature map from the encoding process from being removed.

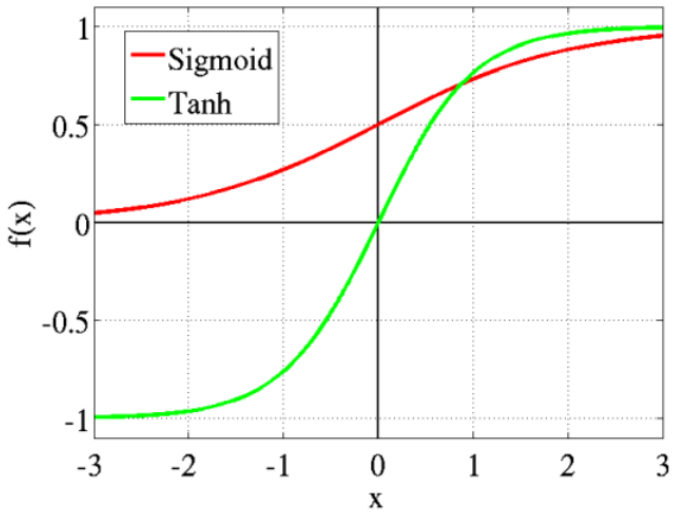

Figure 50: Sigmoid vs hyperbolic tangent activation function [39]

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_105 (Conv2D)          (None, 16, 16, 128)       16512
_________________________________________________________________
conv2d_106 (Conv2D)          (None, 4, 4, 512)         1049088
_________________________________________________________________
flatten_26 (Flatten)         (None, 8192)              0
_________________________________________________________________
dense_26 (Dense)             (None, 1024)              8389632
_________________________________________________________________
reshape_26 (Reshape)         (None, 1, 1, 1024)        0
_________________________________________________________________
conv2d_transpose_209 (Conv2D (None, 8, 8, 512)         33554944
_________________________________________________________________
conv2d_transpose_210 (Conv2D (None, 32, 64, 256)       4194560
_________________________________________________________________
conv2d_transpose_211 (Conv2D (None, 64, 128, 32)       32800
_________________________________________________________________
conv2d_transpose_212 (Conv2D (None, 128, 256, 1)       129
=================================================================
Total params: 47,237,665
Trainable params: 47,237,665
Non-trainable params: 0
_________________________________________________________________
```

Figure 51: Model summary for the encoderDecoderWithDenseXVel

The summary in Figure 51 shows the shape of the output of each layer and the number of trainable and non-trainable parameters. The summary for both CNNs are the same. The code snippet in Figure 52 shows how the layers are defined for *encoderDecoderWithDenseXVel* using Keras' Sequential object. The layers for *encoderDecoderWithDenseYVel* are defined in the same way.

```
140    cnn = Sequential()
141    cnn.add(Conv2D(128, (8,16), strides= (8,16), input_shape=(128, 256, 1), activation='relu'))
142    cnn.add(Conv2D(512, (4,4), strides= (4,4), activation='relu'))
143
144    cnn.add(Flatten())
145    cnn.add(Dense(1024, activation='tanh'))
146    cnn.add(Reshape((1, 1, 1024), input_shape=(1024,)))
147
148    cnn.add(Conv2DTranspose(512, (8,8), strides= (8,8), activation='relu'))
149    cnn.add(Conv2DTranspose(256, (4,8), strides= (4,8), activation='relu'))
150    cnn.add(Conv2DTranspose(32, (2,2), strides= (2,2), activation='relu'))
151    cnn.add(Conv2DTranspose(1, (2,2), strides= (2,2), activation='linear'))
```

Figure 52: Code snippet from model.py (CNNModel 3)

5.3.3.1.4. Compilation and Fitting the Models

After defining the layers in the model, the *compile* method is used (Figure 53). The arguments in the *compile* method will be discussed in the subsequent sub-sections.

```
47    cnnModel = cnn.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])
```

Figure 53: Code snippet from model.py (model compile)

The *fit* method (Figure 54) is used to specify the training of the model. For both CNN models, a value of 0.2 is assigned to *validation_split*. This means 20% of the total input data sets will

be used as the validation set. For Keras, the validation set is selected from the last samples of *input1* and *output1*, then a shuffle is done on the whole data set before the next epoch.

```
51        len1 = len(X_array3U_negGeometry)
52        input1 = np.reshape(X_array3U_negGeometry[:,:,:,0], (len1, 128, 256, 1))
53        output1 = np.reshape(Y_array3U[:,:,:,0], (len1, 128, 256, 1))
54        cnnModel = cnn.fit(input1, output1, validation_split=0.2, batch_size = 1, epochs = 100)
```

Figure 54: Code snippet from model.py (model fitting)

5.3.3.2. Loss Function

The loss function provides a feedback during the training phase on how accurately the model is making predictions. As the problem in this project is essentially a regression problem, the loss function used in all 3 models is the MSE, otherwise known as quadratic loss or L2 loss.

The formula for MSE for this model is given as follows:

$$MSE = \frac{1}{128 \times 256} \sum_{n=0}^{127} \sum_{m=0}^{255} \left(y_{n,m} - y_{n,m}^{predicted}\right)^2$$

The MSE gives the average value of the squared distance between each element of the predicted output array and the element in the corresponding position in the actual output array (ground truth) used for the training.

5.3.3.3. Optimiser

The Adam optimiser is used in the training of the models. The Adam optimiser is known to combine the advantages of 2 other extensions of the stochastic gradient descent: The Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp).

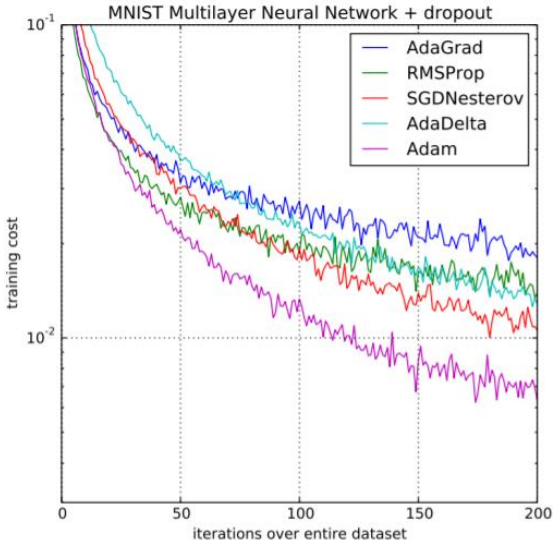

Figure 55: Performance of Adam optimiser against other well-known optimisers [40]

Adam adapts the learning rate based on the average of the first moment (mean) and the average of the second moment of the gradients. A study conducted has shown that the Adam optimiser outperforms other well-known optimisers such as the AdaGrad and RMSProp when used in a Multilayer Perceptron algorithm on the MNIST dataset (Figure 55) [40]. Its strong performance makes it the optimiser of choice.

5.3.4. Model Outputs

Each model (CNNModel 1, CNNModel 2 and CNNModel 3) contains 2 separately trained CNNs, each producing 1 output. Therefore, there are 2 outputs per input for each CNNModel, one for a prediction of the X velocity field at steady state while the other for a prediction of the Y velocity field at steady state. The outputs have dimensions (1, 128, 256, 1), which are the same as the inputs.

Chapter 6: Results and Discussions

6.1. Results of Model Training

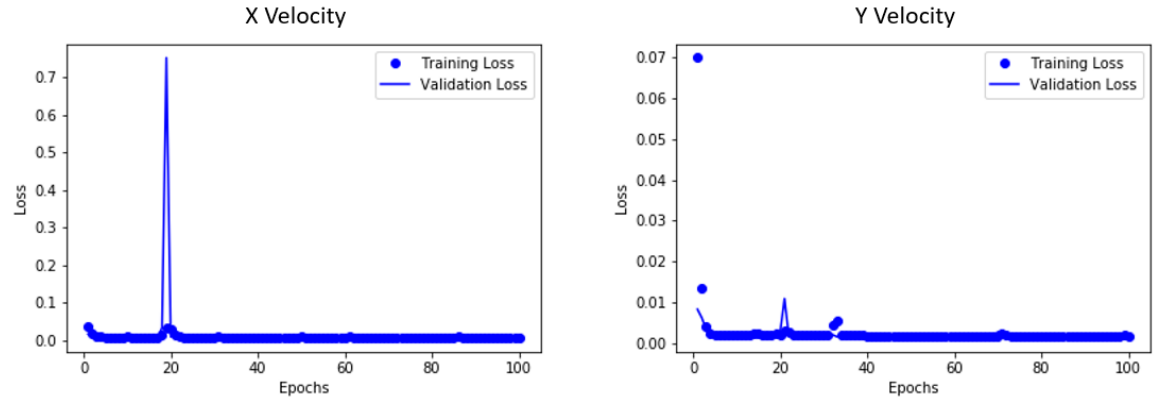

Figure 56: CNNModel 1 – Training and validation loss for fullConXVelocity (left) and fullConYVelocity (right)

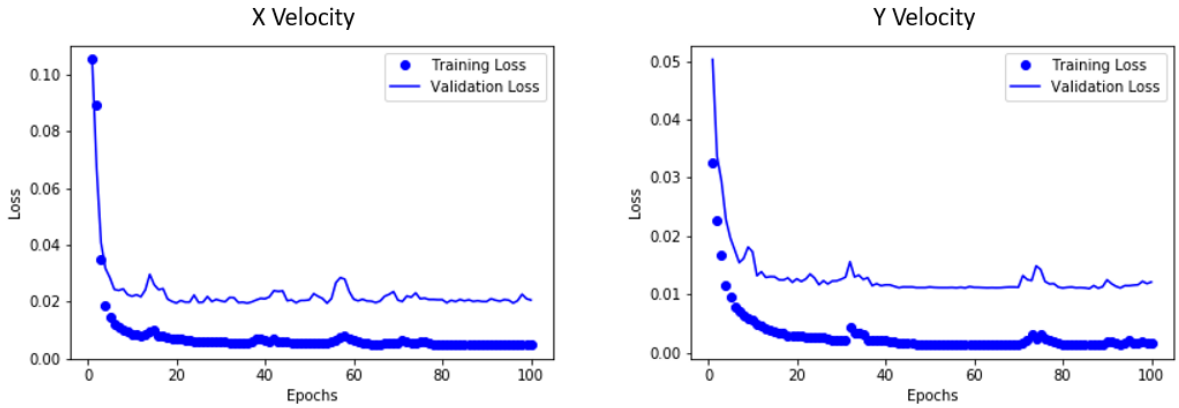

Figure 57: CNNModel 2 – Training and validation loss for encoderDecoderNoDenseXVel (left) and encoderDecoderNoDenseYVel (right)

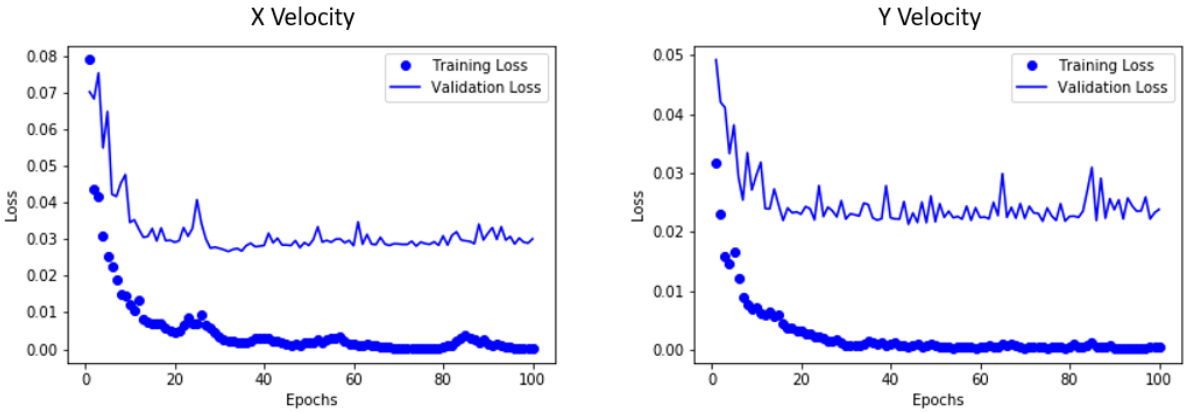

Figure 58: CNNModel 3 – Training and validation loss for encoderDecoderWithDenseXVel (left) and encoderDecoderWithDenseYVel (right)

From Figure 56, at the 100th epoch, there are no visible divergences between the training and validation losses for CNNModel 1. Since both training and validation losses are low, the model is a good fit. From Figures 57 and 58, at the 100th epoch, the validation loss is significantly higher than the training loss for both velocities in CNNModel 2 and CNNModel 3. In both models, the training and validation losses decreased and stabilised after a certain number of epochs, with the validation loss stabilising at a higher value than the training loss and the difference between the validation and training losses relatively constant up till the 100th epoch. It is also noted that the training of CNNModel 2 for both velocities converged faster than the training of CNNModel 3. The low training loss and high validation losses for CNNModel 2 and CNNModel 3 implies that the models are unable to generalise well on cases it has not encountered before but are able to sufficiently capture the features on the data set they are trained on.

6.2. Case Study using Trained Model

Case studies using sample obstacles allow visualisation of the result of training the models. This section compares the performance of the model using 3 different sample obstacles that the model has not encountered before.

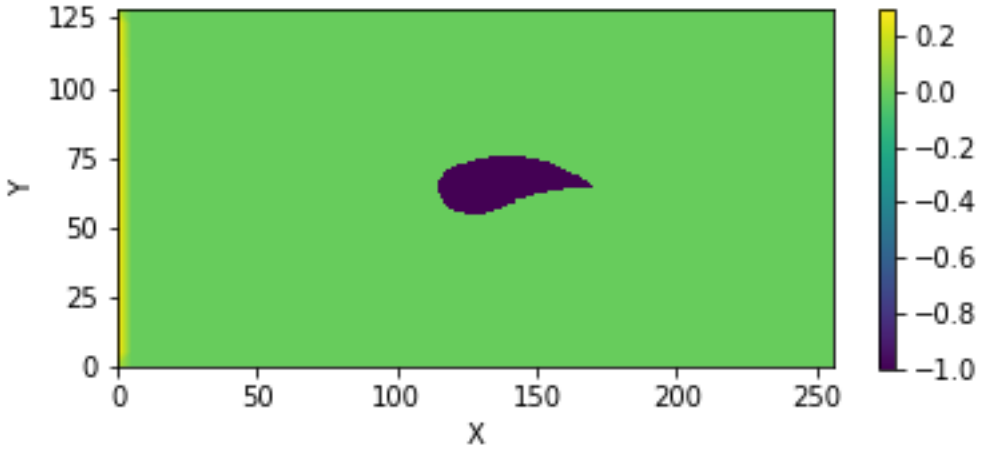

Figure 59: Sample 1 – Input

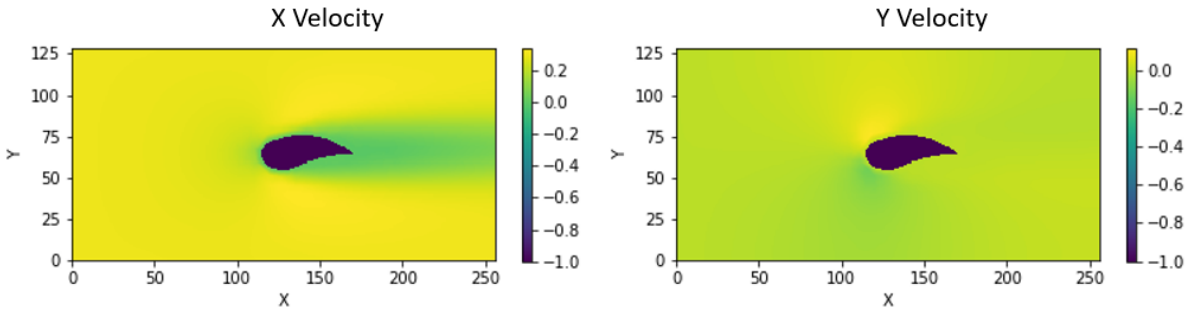

Figure 60: Sample 1 – Ground truth

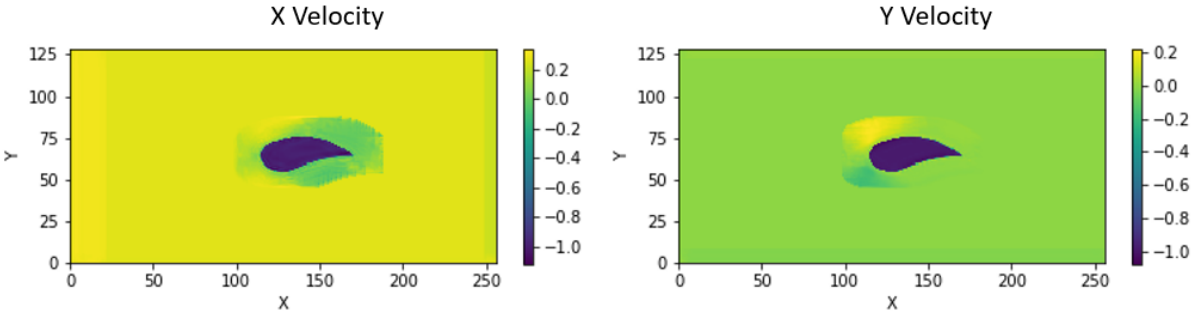

Figure 61: Sample 1 – Predictions of CNNModel 1

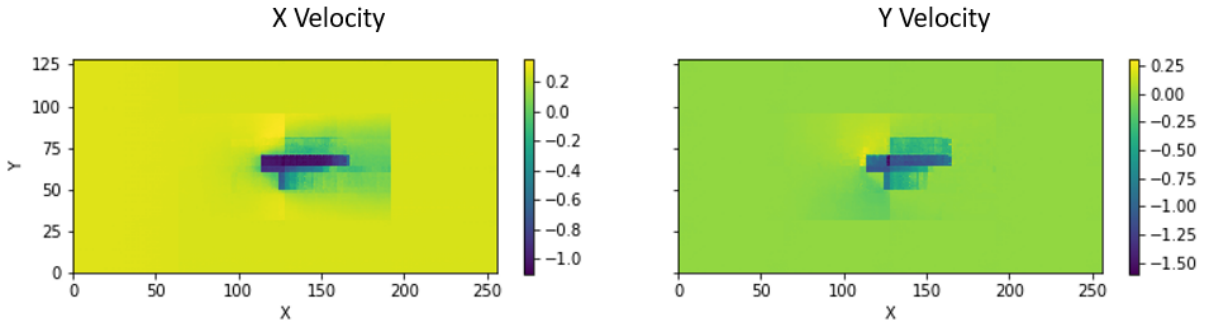

Figure 62: Sample 1 – Predictions of CNNModel 2

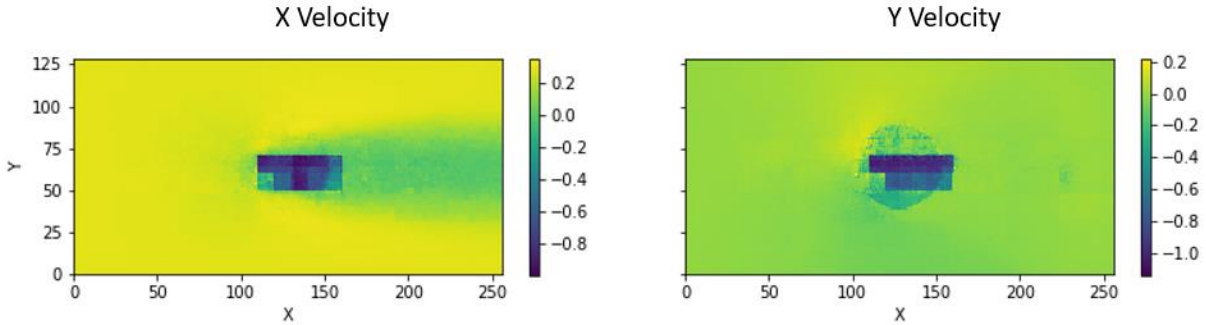

Figure 63: Sample 1 – Predictions of CNNModel 3

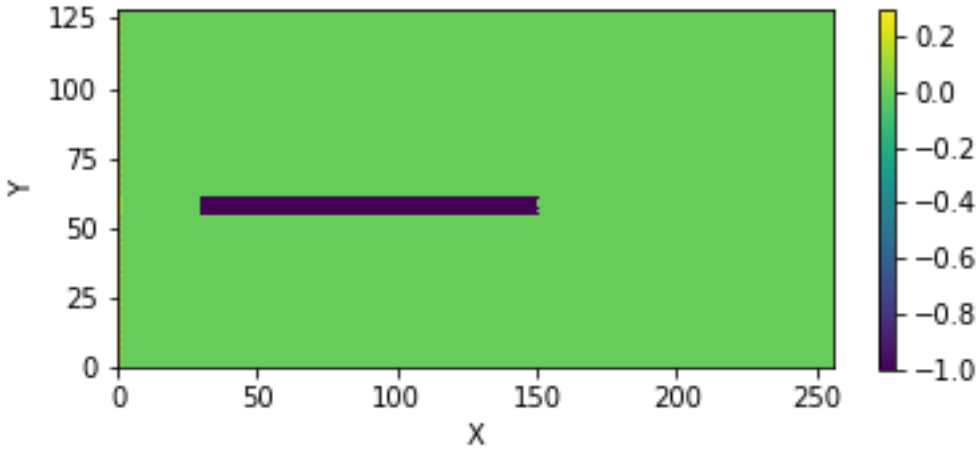

Figure 64: Sample 2 – Input

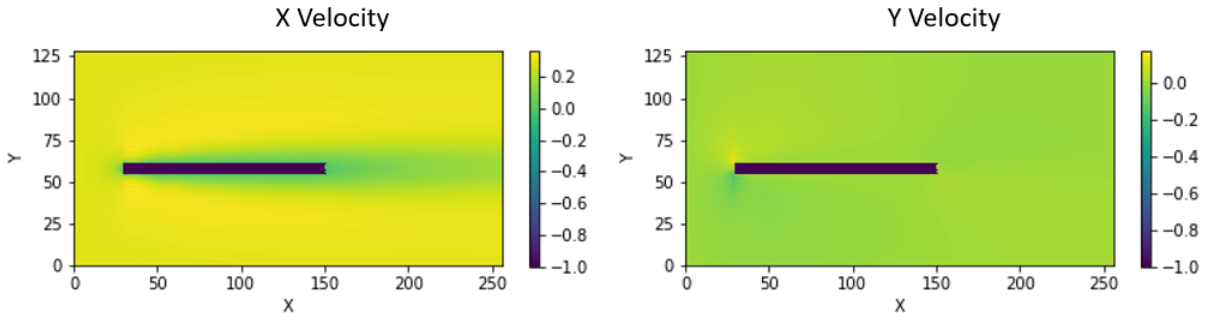

Figure 65: Sample 2 – Ground truth

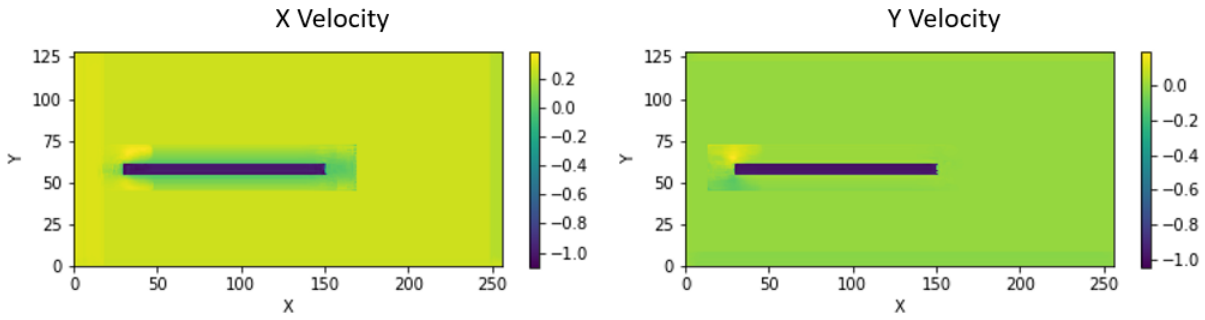

Figure 66: Sample 2 – Predictions of CNNModel 1

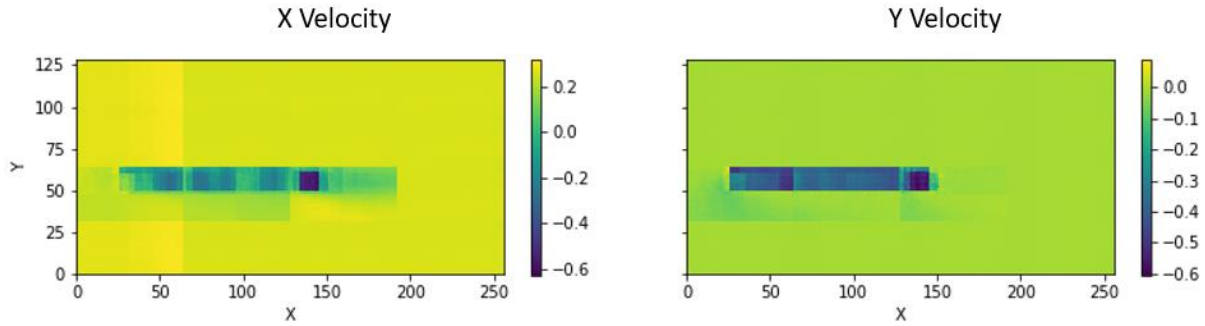

Figure 67: Sample 2 – Predictions for CNNModel 2

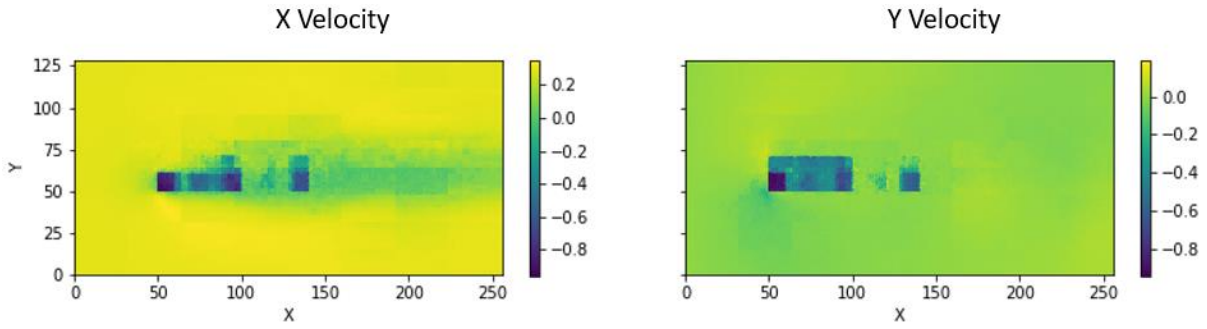

Figure 68: Sample 2 – Prediction for CNNModel 3

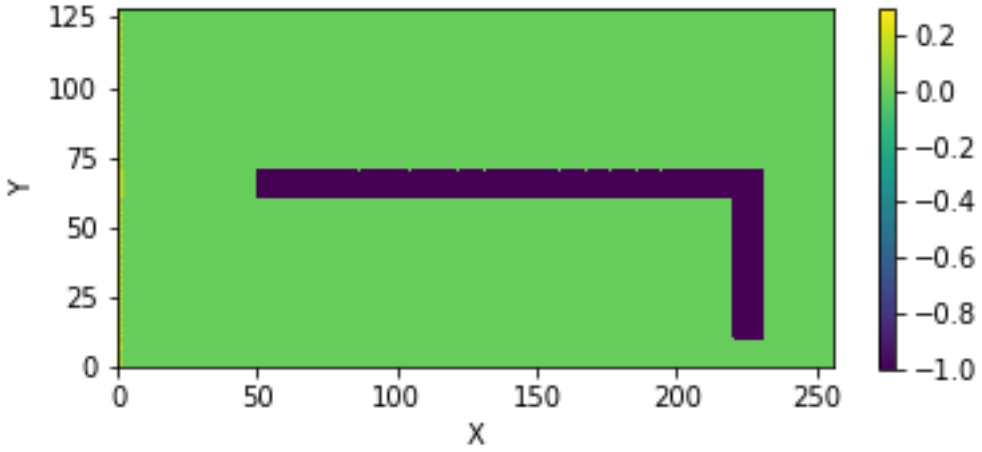

Figure 69: Sample 3 – Input

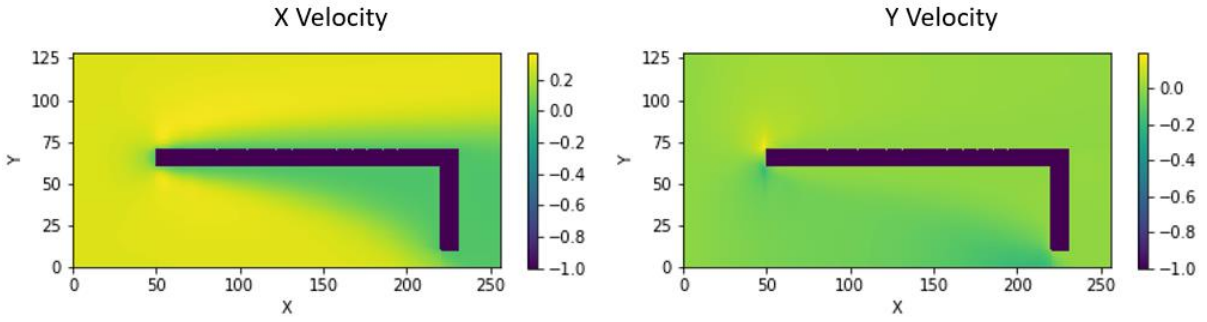

Figure 70: Sample 3 – Ground truth

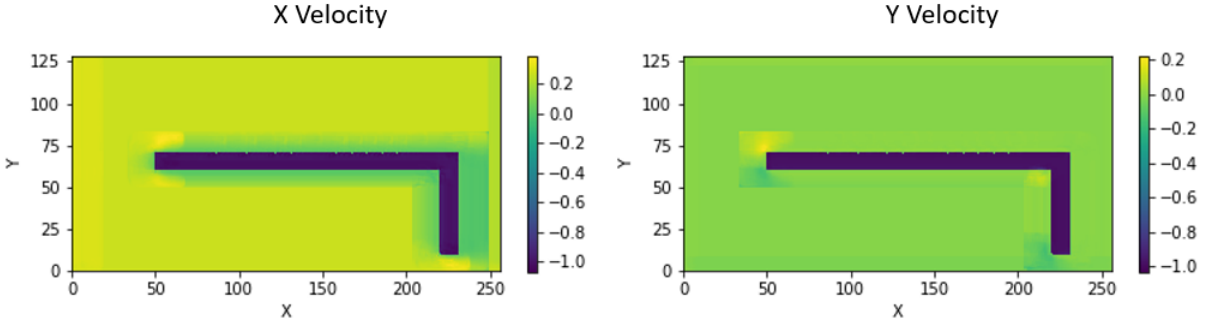

Figure 71: Sample 3 – Predictions of CNNModel 1

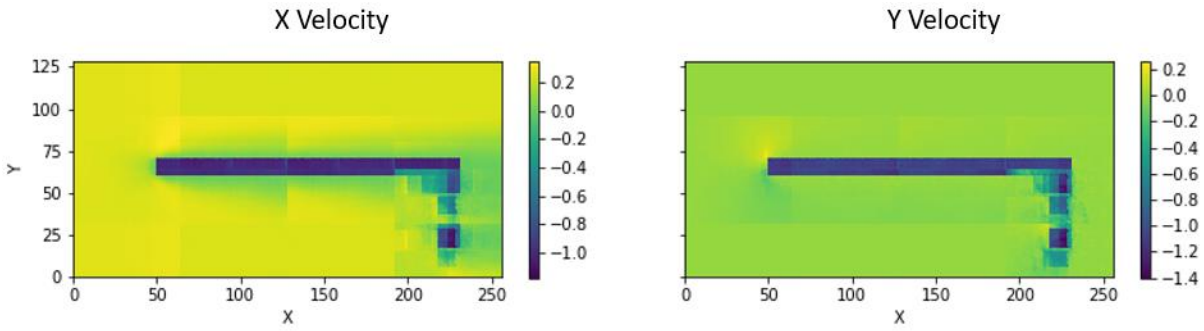

Figure 72: Sample 3 – Predictions of CNNModel 2

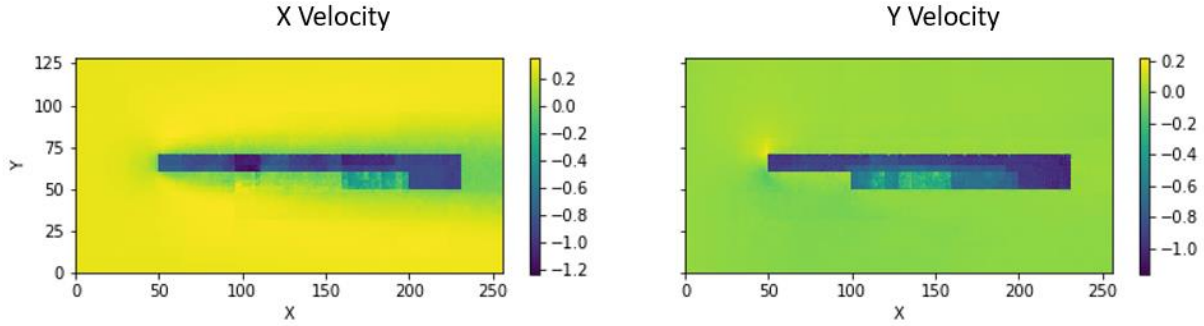

Figure 73: Sample 3 – Predictions of CNNModel 3

CNNModel 1 is unable to make accurate predictions on the velocity field in the region after the obstacle (the wake region). This is most evident in Figure 66 where the green shade following the trailing edge of the obstacle abruptly changes colour. CNNModel 1 is also unable to make accurate predictions on the velocity field that lies too far from the obstacle. However, the model accurately predicts the geometry of the obstacle and the velocity field that lies adjacent to the obstacles.

The addition of a fully connected layer in CNNModel 3 allows the model to better capture information that lies further away from the obstacle. This is most evident in Figures 63 and 68 where there is a continuation of flow beyond the trailing edge of the obstacle. The flow fields predicted by CNNModel 3 are much smoother than those predicted by CNNModel 1 and CNNModel 2. This is possibly because each node in the fully connected layers are connected to every other node in the preceding and subsequent convolution layer.

However, the presence of a fully connected layer has resulted in the model having difficulty in picking up information on the geometry of the obstacle. This is evident in Figure 73, where the vertical portion of the obstacle is truncated.

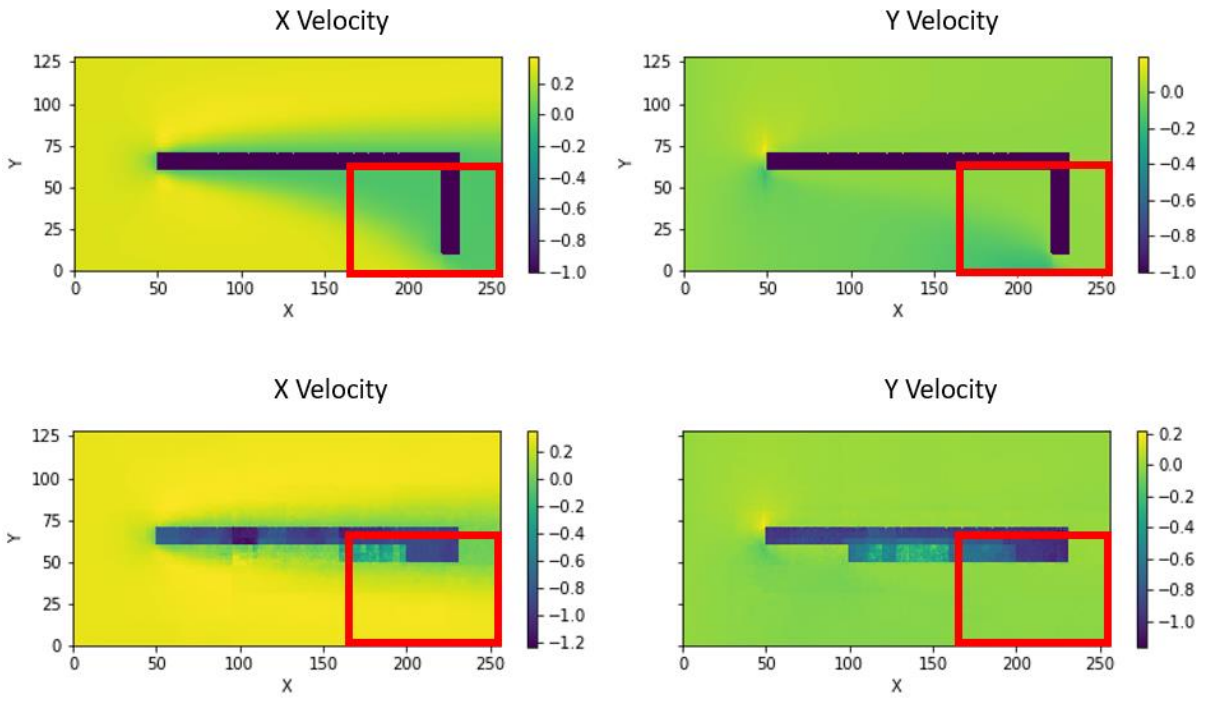

Figure 74: Comparison between ground truth (top) and CNNModel 3 (bottom) for Sample 3

From Figure 74, it is observed that the velocity field in the area around the vertical portion of the ground truth (in red highlighted box) does not have any resemblance to the ground truth. Given the input geometry shown in Figure 69, the model failed to capture the vertical portion of the obstacle, but it was expected to replicate the flow field in that region to predict a velocity field somewhat similar to the ground truth. However, CNNModel 3 predicted a velocity field for the obstacle that the model detected instead of the obstacle that was fed to the model. The failure to capture the obstacle resulted in a poor prediction of the velocity field and is the most likely cause of the relatively high mean square error for its prediction on Sample 3's input.

A model with an encoder-decoder architecture without a fully connected layer, such as in CNNModel 2, has a performance in between that of CNNModel 1 and CNNModel 3. It is unable to capture the obstacle's geometry as well as CNNModel 1, but better than CNNModel 3. CNNModel 2 is able to capture the velocity field of a larger region surrounding the obstacle compared to CNNModel 1. However, it also produces a slightly pixelated velocity field and is unable to capture the velocity field that lies too far from the obstacle, most evident in Figures 62 and 67.

		CNNModel 1	CNNModel 2	CNNModel 3
Sample 1	X Velocity	0.0041670	0.011569	0.016099
	Y Velocity	0.00070011	0.0075151	0.012646
Sample 2	X Velocity	0.0029360	0.021424	0.023357
	Y Velocity	0.00028578	0.013637	0.019486
Sample 3	X Velocity	0.0050623	0.013630	0.033407
	Y Velocity	0.0014418	0.0090934	0.029206

Table 7: Mean square error for each sample for each model

The errors between the ground truth output and predictions by each model is given in Table 7. CNNModel 1 yields the lowest MSE across all samples, followed by CNNModel 2, then CNNModel 3. However, it will be erroneous to nominate CNNModel 1 as the best and CNNModel 3 as the worst model since CNNModel 2 and CNNModel 3 are not trained to their optimum points.

For CNNModel 1, the prediction on Sample 2 has the lowest MSE, followed by Sample 1 then Sample 3, which has the highest MSE. This is expected since CNNModel 1 is only able to make accurate predictions for the velocity field near the obstacle. Its ability to predict worsens as the boundary layer surrounding the obstacle becomes larger.

For CNNModel 2, the prediction on Sample 1 has the lowest MSE, followed by Sample 3 then Sample 2, which has the highest MSE. CNNModel 2's ability to capture a larger region of velocity field surrounding the obstacle allows it to achieve a satisfactory performance even when the boundary layer is larger.

For CNNModel 3, the prediction on Sample 1 has the lowest MSE, followed by Sample 2 then Sample 3, which has the highest MSE. Despite CNNModel 3 being able to predict the velocity fields of the boundary layer and wake region more effectively, its failure to accurately capture the obstacle geometry in Sample 3 resulted in significantly poorer predictions for that sample.

The pure convolution layers are able to capture some physical information on the velocity fields, however, the model is unable to make predictions that are sufficiently accurate to approximate the non-linear Navier Stokes equations especially in regions further from obstacles. The use of strided convolution in an encoder-decoder architecture without fully connected layers results in capturing a larger region of the velocity field, but the model still fails to accurately predict the velocity fields further away from the obstacle. The addition of a fully connected layer to the encoder-decoder architecture enables the model to predict the

velocity fields in the boundary layer and wake region of the obstacle much more accurately. The predicted velocity field is also smoother when visualised. However, the accuracy of its prediction is contingent on its ability to capture the obstacle's geometry effectively.

In general, there is greater loss of obstacle geometry information when strided convolutions (encoder-decoder architectures) are used. As the model downsamples, it is observed that more spatial features are lost (this is also an issue documented by Long et al. [12]), but a wider region of information can be captured. The use of a fully connected layer enhances this ability. Unstrided convolution layers appear to better retain spatial features mainly because the absence of downsampling means that spatial information is not lost as they are passed through the layers. However, the absence of downsampling reduces the size of the region that subsequent convolution kernels cover and hence, less information is captured.

The use of strided convolutions with and without a fully connected layer shows potential. The information lost through the downsampling process can probably be recovered through techniques such as skipped connections or the fusing of outputs from intermediate layers. The optimal methodology will have to be discovered via experiment but it is expected that such techniques can improve the models' performances.

Chapter 7: Recommendations and Conclusion

7.1. Recommendations for Future Work

7.1.1. Improvements to Project Model

This project focused on the use of different architectures. The other variables that are involved in the construction of the whole machine learning model are kept consistent across CNNModel 1, CNNModel 2 and CNNModel 3 as far as possible. As mentioned in Chapter 6, CNNModel 2 and CNNModel 3 are not trained to their optimum point. With 3 optimally trained models, a fair comparison can then be done to determine the best performing model architecture.

7.1.2. Modification of the Project Model Architecture

The models used in this project are very basic in nature. Techniques such as skipped connections [41] or fusing outputs of various layers [12]. These methods have shown to improve the performance of CNNs using an encoder-decoder architecture.

7.1.3. Characterisation Studies

It can be seen from the earlier sections that a neural network has a lot of hyperparameters. The modification of a hyperparameter can possibly lead to significantly better performance of the model. There are numerous permutations that can be explored, however, due to time and resource constraints, most of the efforts were directed to developing a model that is able to achieve satisfactory results.

In addition to hyperparameters, the network architecture can also affect the performance of the model. Simple changes such as the addition of another convolution layer may even lead to better performance. The model used in this project is a simple model using stacked convolution layers with the Sequential object. More complex network architectures can be constructed using Keras' Functional API. Many newer modern machine learning network architectures are a result of creativity and empirical studies and these may pave the way towards solving even more difficult data problems in the future.

7.1.4. Improve Data Set Coverage

Data forms the basis of machine learning. It is generalised that larger data sets can lead to the production of models that are more accurate. This can be done via generating more data sets or using data augmentation techniques. However, as mentioned earlier, data augmentation techniques traditionally used for CNNs in computer vision and classification problems are not likely to work for fluid simulation problems. Therefore, data augmentation techniques that are

applicable to fluid simulation problems is an area worth looking into, since the generation of training data can be rather time consuming.

7.1.5. Using Different Types of Input Data

This project used inputs and outputs where the cells corresponding to the obstacle are given a value of -1. Other variations that can be explored include assigning a value of 0 to the cells corresponding to the obstacle or using more complex functions such as the signed distance function.

7.1.6. Development for Practical Applications

Besides the development of a machine learning model that makes accurate predictions on fluid flow fields around an object, creative uses for this technology can eventually lead to shifts in industry trends. Perhaps in a decade, fluid flows around a 3D object can be rendered in real time with high accuracy just by pointing a mobile camera at the object. The tools for building such technological capabilities already exist and it is up to mankind's creativity to develop new uses and push the boundaries.

7.2. Conclusion

Machine learning is an extremely broad field and this project barely scratches the surface of what can be achieved. Nonetheless, this project has had some successes in using machine learning to predict the velocity field of a fluid flow around an obstacle and provided some insights into the performance of certain network architectures. The results were not spectacular, but they pinpointed some directions for further improvement. Building a successful model involves experimentations, a high level of technical understanding, creativity and perhaps, some luck. With enough time and resources devoted into the field of machine learning, there is no limit on what mankind can achieve in the future.

References

[1] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” Nov 1990. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[2] Y. LeCun, C. Cortes and C. J. Burges, “The MNIST Database of Handwritten Digits,” [Online]. Available: http://yann.lecun.com/exdb/mnist/index.html.

[3] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” 2012. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[4] J. Tompson, K. Schlachter, P. Sprechmann and K. Perlin, “Accelerating Eulerian Fluid Simulation With Convolutional Networks,” 2017.

[5] C. Yang, X. Yang and X. Xiao, “Data-driven projection method in fluid simulation,” *Computer Animation and Virtual Worlds,* no. 27, pp. 415-424, 2016.

[6] M. Chu and N. Thuerey, “Data-Driven Synthesis of Smoke Flows with CNN-based Feature Descriptors,” *ACM Transactions on Graphics,* vol. 36, 2017.

[7] X. Guo, L. Wei and F. Iorio, “Convolutional Neural Networks for Steady Flow Approximation,” San Francisco, CA, USA, 2016.

[8] X. Jin, P. Cheng, W.-L. Chen and H. Li, “Prediction model of velocity field around circular cylinder over various Reynolds numbers by fusion convolutional neural networks based on pressure on the cylinder,” *Physics of Fluids,* vol. 30, 2018.

[9] J. Na, K. Jeon and W. B. Lee, “Toxic gas release modeling for real-time analysis using variational autoencoder with convolutional neural networks,” *Chemical Engineering Science,* vol. 181, pp. 68-78, 2018.

[10] M. Kano, S. Hasebe, I. Hashimoto and H. Ohno, “A new multivariate statistical process monitoring method using principal component analysis,” *Computers and Chemical Engineering* , vol. 25, pp. 1103-1113, 2001.

[11] H. G. E. and S. R. R., “Reducing the Dimensionality of Data with Neural Networks,” *Science,* vol. 313, pp. 504-507, 2006.

[12] J. Long, E. Shelhamer and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[13] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *International Conference on Learning Representations*, 2015.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going deeper with convolutions,” 17 Sep 2014. [Online]. Available: https://arxiv.org/pdf/1409.4842.pdf.

[15] S.-H. Tsang, “Review: FCN—Fully Convolutional Network (Semantic Segmentation),” 6 Oct 2018. [Online]. Available: https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1.

[16] F. Ning, D. Delhomme, Y. LeCun, F. Piano, L. Bottou and P. E. Barbano, “Toward Automatic Phenotyping of Developing Embryos from Videos,” 2005. [Online]. Available: https://leon.bottou.org/publications/pdf/tip-2005.pdf.

[17] SimScale, “What is Aerodynamics?,” [Online]. Available: https://www.simscale.com/docs/content/simwiki/cfd/what-is-aerodynamics.html.

[18] COMSOL Multiphysics, “Navier-Stokes Equations,” 15 Jan 2015. [Online]. Available: https://www.comsol.com/multiphysics/navier-stokes-equations.

[19] W. Zuo, “Introduction of Computational Fluid Dynamics,” [Online]. Available: http://wwwmayr.informatik.tu-muenchen.de/konferenzen/Jass05/courses/2/Zuo/Zuo_paper.pdf.

[20] E. V. Magnus, “The lattice Boltzmann method: Fundamentals and acoustics,” Trondheim, 2014.

[21] S. Mukherjee, “[Infographic] Applications of Artificial Intelligence (AI) in business,” 20 Jul 2018. [Online]. Available: https://www.hackerearth.com/blog/innovation-management/applications-of-artificial-intelligence/.

[22] McKinsey & Company, “Artificial Intelligence The Next Digital Frontier?,” Jun 2017. [Online]. Available: https://www.mckinsey.com/~/media/McKinsey/Industries/Advanced%20Electronics/Our%20Insights/How%20artificial%20intelligence%20can%20deliver%20real%20value%20to%20companies/MGI-Artificial-Intelligence-Discussion-paper.ashx.

[23] G. Yufeng, “The 7 Steps of Machine Learning,” 1 Sep 2017. [Online]. Available: https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e.

[24] M. Mody, D. Kumar, P. Swami, M. Mathew and S. Nagori, “Low Cost and Power CNN/Deep Learning Solution for Automated Driving,” *IEEE*, 2018.

[25] A. Deshpande, “A Beginner's Guide To Understanding Convolutional Neural Networks,” 20 Jul 2016. [Online]. Available: https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/.

[26] S. Sharma, “Activation Functions: Neural Networks,” 6 Sep 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[27] Computer Science Wiki, “Max-pooling / Pooling,” [Online]. Available: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[28] G. Ognjanovski, “Everything you need to know about Neural Networks and Backpropagation—Machine Learning Easy and Fun,” 15 Jan 2019. [Online]. Available: https://towardsdatascience.com/everything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a.

[29] H. Tao and X. Lu, “On Comparing Six Optimization Algorithms for Network-based Wind Speed Forecasting,” in *37th Chinese Control Conference*, Wuhan, China, 2018.

[30] D. Mishkin, N. Sergievskiy and J. Matas, “Systematic evaluation of CNN advances on the ImageNet,” 2016.

[31] W. Koehrsen, “Overfitting vs. Underfitting: A Complete Example,” 28 Jan 2018. [Online]. Available: https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765.

[32] J. Brownlee, “How to Diagnose Overfitting and Underfitting of LSTM Models,” 1 Sep 2017. [Online]. Available: https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/.

[33] M. Heller, “What is CUDA? Parallel programming for GPUs,” 30 Aug 2018. [Online]. Available: https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html.

[34] Keras, “Why use Keras?,” [Online]. Available: https://keras.io/why-use-keras/.

[35] C. Greenshields, “OpenFOAM v6 User Guide: 5.3 Mesh generation with blockMesh,” 10 Jul 2018. [Online]. Available: https://cfd.direct/openfoam/user-guide/v6-blockmesh/.

[36] OpenFOAM, “4.5 Mesh conversion,” [Online]. Available: https://www.openfoam.com/documentation/user-guide/mesh-conversion.php.

[37] OpenFOAM Wiki, “OpenFOAM guide/The PISO algorithm in OpenFOAM,” 7 Sep 2013. [Online]. Available: https://openfoamwiki.net/index.php/OpenFOAM_guide/The_PISO_algorithm_in_OpenFOAM.

[38] SuperDataScience, “Convolutional Neural Networks (CNN): Step 3 - Flattening,” 18 Aug 2018. [Online]. Available: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening.

[39] S. Sharma, “Activation Functions in Neural Networks,” 6 Sep 2017. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[40] D. P. Kingma and J. L. Ba, “Adam: A Method for Stochastic Optimization,” 2015. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf.

[41] J. C. Ye and W. K. Sung, “Understanding Geometry of Encoder-Decoder CNNs,” 22 Jan 2019. [Online]. Available: https://arxiv.org/pdf/1901.07647.pdf.