Machine learning approach towards predicting turbulent fluid flow using convolutional neural networks

---

Vinh Vu

June 9, 2023
Version: Final

University of Sheffield

Machine learning approach towards predicting turbulent fluid flow using convolutional neural networks

Vinh Vu

The School of Mathematics and Statistics, and the Department of Mechanical Engineering

Supervisors Robertus Erdelyi, Yi Li, Franck Nicolleau and Andrew Nowakowski

June 9, 2023

Vinh Vu

Machine learning approach towards predicting turbulent fluid flow using convolutional neural networks

June 9, 2023

Supervisors: Robertus Erdelyi, Yi Li, Franck Nicolleau and Andrew Nowakowski

University of Sheffield

The School of Mathematics and Statistics, and the Department of Mechanical Engineering

Abstract

Using convolutional neural networks, we present a novel method for predicting turbulent fluid flow through an array of obstacles in this thesis. In recent years, machine learning has exploded in popularity due to its ability to create accurate data-driven models and the abundance of available data. In an attempt to understand the characteristics of turbulent fluid flow, we utilise a novel convolutional autoencoder neural network to predict the first ten POD modes of turbulent fluid flow. We find that the model is able to predict the first two POD modes well although and with less accuracy for the remaining eight POD modes. In addition, we find that the ML-predicted POD modes are accurate enough to be used to reconstruct turbulent flow that adequately captures the large-scale details of the original simulation.

Acknowledgement

There are many people for who I am grateful for their support throughout my PhD life, to the extent that another thesis could be written about their support.

I would first like to thank my Masters and first Mechanical Engineering PhD supervisor, Dr Franck Nicolleau, for initially guiding my route to research and encouraging me to pursue my interest in fluid mechanics and turbulence. My interest in using computational methods would not have started without his encouragement. I would also like to thank Dr Andrew Nowakowski for his support in my work. I also would like to thank Professor Robertus Erdelyi for his support. Aside from support and the occasional funny stories and jokes, he has helped me in obtaining the funding for this PhD, which led me to start my PhD life. Finally, I would like to give my greatest thanks to Dr Yi Li, who has helped and guided me throughout my research. Yi has been very kind and patient, and I have been lucky to have him as a supervisor who could introduce me to advanced fluid dynamics research and interesting methods which have aided in my rapid development. I wish I could still learn more from him.

I would like to acknowledge both EPSRC and the Faculty of Science Doctoral Academy for their financial support. For this work, I had access to many different HPCs. I would like to acknowledge the IT Services at The University of Sheffield for the provision of services for High-Performance Computing. This work used the Cirrus UK National Tier-2 HPC Service at EPCC http://www.cirrus.ac.uk funded by the University of Edinburgh and EPSRC (EP/P020267/1). This project made use of time on Tier 2 HPC facility JADE, funded by EPSRC (EP/P020275/1). This project made use of time on Tier 2 HPC facility JADE2, funded by EPSRC (EP/T022205/1).

I would like to thank all the PhD students on H-Floor in Hicks building for their kind friendship and support. Particularly, Hope, Poppy and Callum who has been with me since I joined the Maths Department. Finally, I would like to express my gratitude to my grandparents, parents, my sisters and my girlfriend. Without their support and encouragement over the past few years, it would be impossible for me to complete my study.

Declaration

This thesis is the result of my own work and does not include anything that is the outcome of work done by or in collaboration with others, except where specifically indicated. All sources referred to are cited in the text accordingly

Vinh Vu

Contents

1 Introduction 1
1.1 Outline . . . 9

2 Machine Learning in Fluid Dynamics 11
2.1 Introduction . . . 11
2.2 Deep learning basics . . . 15
2.2.1 Feedforward Networks (FNN) overview . . . 15
2.2.2 Backpropagation . . . 17
2.2.3 Activation functions . . . 18
2.2.4 Deep Learning optimiser . . . 19
2.3 Convolutional neural networks . . . 20
2.4 Training . . . 23
2.4.1 Initialisation . . . 23
2.4.2 Learning rate and batch size . . . 24
2.4.3 Standardisation, Normalisation and Batch Normalisation . . . 25
2.4.4 Hyperparameters and Tuner . . . 27
2.4.5 Sample size . . . 28
2.4.6 Conclusion . . . 29

3 The Lattice Boltzmann Method 31
3.1 Introduction . . . 31
3.2 Boltzmann equation . . . 32
3.3 The Lattice Boltzmann Method . . . 34
3.3.1 Simulation process . . . 36
3.3.2 LBM Boundary Conditions . . . 38
3.3.3 Computational parallelism . . . 40
3.4 OpenLB . . . 41
3.5 Summary . . . 42

4 Reduced order modelling 43
4.1 Introduction . . . 43
4.2 Proper Orthogonal Decomposition . . . 43
4.3 POD Example . . . 45
4.4 Summary . . . 51

5 Predicting POD modes using convolutional neural network 53
5.1 Introduction . . . 53
5.2 Dataset Generation . . . 54
5.2.1 Outline . . . 54
5.2.2 Simulation Set up . . . 55
5.2.3 POD mode calculation . . . 58
5.2.4 Data Analysis tools . . . 59
5.3 Convolutional neural network . . . 61
5.3.1 Outline . . . 61
5.3.2 Model architecture search . . . 61
5.3.3 Model architecture . . . 62
5.3.4 Signed Distance Function . . . 64
5.3.5 Pre-processing and Post-processing . . . 65
5.3.6 Training . . . 65
5.4 Results . . . 68
5.4.1 Error evaluation . . . 68
5.4.2 Predictions and reconstruction . . . 73
5.4.3 Prediction discussion . . . 79
5.5 Future work with SSIM . . . 80

6 Conclusion 89

A Additional figures 93
A.1 Calculated POD mode example, for POD modes from three to ten . . . 93
A.2 Epoch Training graphs for POD modes 3 to 10 . . . 95
A.2.1 Full Epoch training graphs . . . 95
A.2.2 Cropped Epoch training graphs . . . 97
A.3 Model error histograms for POD modes 3 to 10 . . . 99
A.4 Error distribution compared against obstacle number plots for POD modes 3 to 10 . . . 108
A.5 Best POD mode predictions . . . 113
A.6 POD mode comparison . . . 116
A.6.1 Case 332 . . . 116
A.6.2 Case 33 . . . 119
A.7 Reconstruction snapshots . . . 122
A.7.1 Case 332 . . . 122
A.7.2 Case 33 . . . 122
A.7.3 Case 332 with rescaled POD modes . . . 122
A.8 Averaged POD modes . . . 129
A.9 SSIM . . . 131
A.9.1 SSIM comparison figures . . . 131
A.9.2 SSIM histogram error . . . 136

List of Figures

1.1 The Burj Khalifa. Based on the Hymenocallis Flower, the Burj Khalifa has three 'wings' which surface area progressively gets smaller as the building gets higher. As the cross-section of the building changes, this affects the wind flow and disrupts resonance from occurring. Picture taken by Shahin Ghanbari. 2

1.2 Picture taken by Landsat 7 shows the clouds off the coast of Chile, near the Juan Fernandez Islands (commonly known as the Robinson Crusoe Islands), on September 15, 1999. Here the von Karman vortex street is clearly seen. 3

1.3 The energy cascade. k is the wavenumber and $E(k)$ represents the energy of the wake. 4

1.4 Observed self-similarity in turbulent fluid flow. The image on the left is the original image, while the image on the right has been magnified, as shown in the image in the middle. 5

1.5 This image captured by the Hubble Space Telescope depicts Sh 2-106. A newly-formed star known as S106 IR is responsible for the hourglass-shaped gas cloud and the visible turbulence within. 5

1.6 The Lenaelva River in Norway, near Skrei. It is rotated to form a parallel with figure 1.5. The river's water is falling over a waterfall and causing turbulence. This type of pattern seems to occur everywhere in nature. 5

1.7 Fractals seen in nature. Notice how each branch looks similar to each sub-branch and that each progressive layer contains more similar detail. 6

1.8 The Sierpinski carpet after three iterations. A well-known fractal shape in which eight small squares are placed around each large square at each iteration. Each smaller square is a third the size of its larger counterpart. Following each iteration, eight smaller squares are placed around each small square, and this process is repeated onto infinity. . 7

2.1 A single artificial neuron. This diagram represents the mathematical function of $\vec{y} = \sigma(\vec{\omega}x + \vec{b})$. 16

2.2 A deep learning model with three layers, 1 input layer $\vec{x}$, 1 hidden layers and 1 output layer $\vec{y}$. Each layer does a weighed sum of the previous layer. 16

2.3 A convolution operation by a convolutional layer. The filter scans across the input data and selects a n by m array and applies the dot product with the values in the filter to produce a single value. In this case, the filter is 2 by 2 and the stride is 1 by 1. 21

2.4 A convolution operation by a convolutional layer with padding. Arbitrary values, in this case zeros are added outside the input data to prevent the reduction in dimension. 22

2.5 A transpose convolution operation conducted by a transpose convolutional layer. The extra padding increases the dimension of the input data, and a convolution operation is applied onto the input data with extra padding to produce a feature map with greater dimension compared to the input data. 22

2.6 The batch normalisation process. The output values produced after passing through a layer is standardised and fed into the next layer. . . 26

3.1 D2Q9 velocity set. Here the distribution functions f_i from i=1-9 travel to nodes connected to the centre node via the connection lines during the streaming step. f_0 remains in the centre node. 35

3.2 Bounce back boundary condition. Before the outbound flow crosses the wall during the streaming step, f_4, f_7 and f_8 are chosen to be redirected. 39

3.3 Bounce back boundary condition. Here, the bounce-back boundary condition is applied and the distribution functions f_4, f_7 and f_8 are redirected backwards. 39

3.4 Bounce back boundary condition. Here, after the bounce-back boundary condition is applied, the distribution functions f_4, f_7 and f_8 replace the former distribution functions f_2, f_5 and f_6 respectively as the new distribution functions after the streaming step. 40

4.1 The instantaneous streamwise velocity in 2D flow past a cylinder at a Re=100. 46

4.2 The instantaneous transverse velocity in 2D flow past a cylinder at a Re=100. 46

4.3 The instantaneous vorticity of the 2D flow past a cylinder at Re=100. . 46

4.4 The averaged streamwise velocity in 2D flow past a cylinder at a Re=100. 47

4.5 The averaged transverse velocity in 2D flow past a cylinder at a Re=100. 47

4.6 The first streamwise velocity POD mode. This dominant POD mode is identical to the averaged fluid flow, which is to be expected given that the majority of the fluid flow's energy should be within the simulation's centerline and not close to the wall. 48

4.7 The second POD mode for the streamwise velocity. Note the alternating patterns in the wake behind the cylinder, which tells us about the spatial correlation of the fluctuations of the flow in the area. The region with the highest values seems to be directly behind the cylinder and progressively becomes weaker. The alternating patterns also seems to diverge before converging. 48

4.8 The third POD mode for the streamwise velocity. This POD mode appears to exhibit the same pattern and strength as the second POD mode. The only discrepancy is that the distribution appears to initialise with a half-wake before continuing with the regular patterns observed in the second streamwise POD mode. 48

4.9 The first POD mode for the transverse velocity. The regions of high and low energies seems to correspond to the movement of the wake oscillating up and down along the centre line. 49

4.10 The second POD mode for the transverse velocity. This mode appears to depict the fluid flow being split by the cylinder in the simulation's centre, after which the fluid converges back to the centre line. This looks identical to the averaged transverse velocity. 49

4.11 The third POD mode for the transverse velocity. Here the POD mode shows a region with a series of rapidly alternating areas after the flow is split by the obstacle. This region has an alternating pair of positive and negative energies, suggest that the fluid oscillates between each region. The region is initially concentrated behind the obstacle but eventually spreads out across the domain, weakening in the process. Finally, the alternating series of fluctuating energies weakens to an extent that it is hardly visible in the figure. 50

4.12 The fourth POD mode for the transverse velocity. This POD mode is similar to the third transverse velocity POD mode, but it appears to be slightly shifted towards the right. Like before, the region shows areas with a strong fluctuating energies that dissipates throughout the domain. 50

5.1 An array with two obstacles. This is similar to the flow past a set of bluff bodies. 55

5.2 An array with 8 obstacles. This produces a fluid flow that is similar to the flow past a rough surface. 55

5.3 An array with 16 obstacles, is similar to the flow past a rough body. . . 55

5.4 An array with 22 obstacles, is similar to the flow past a rough body as well. 55

5.5 Graph of the distribution of the total number of obstacles in the dataset. 56

5.6 Diagram of the fluid domain. The edges around the fluid domain have periodic boundary conditions applied, and the entire domain has a forcing term applied to it. The obstacles are placed 0.5D away from the left and in between the top and bottom edges of the fluid domain. . . 58
5.7 Diagram of the array of obstacles where the POD modes for example figures 5.8 to A.8. 60
5.8 First POD mode of the simulated flow through an array of obstacles seen in figure 5.7. Here, we see a distinct horizontal separation between the region of positive energy and negative energy, indicating the flow is fluctuating between these areas. 60
5.9 Second POD mode of the simulated flow through an array of obstacles seen in figure 5.7. Similar to the first POD mode as seen in figure 5.8, two distinct horizontal regions are seen but shifted downwards. As before, this suggests the streamwise velocity alternates 60
5.10 The model architecture used for this project. On the top of the figure, we denote the operation between each feature map, and on the bottom, the dimensions of the feature map are labelled. We abbreviate the operations used, in this case, 'Conv' means convolution, 'T.Conv' means transpose convolution and 'Batch norm' means batch normalisation. The numbers in brackets next to 'Conv' and 'T.Conv' represents the filter size used. As for the dimensions, the values represent the height, width and depth of the feature map, respectively. 63
5.11 The SDF of an array with four obstacles. 64
5.12 The SDF of an array with ten obstacles. 64
5.13 Two figures comparing the affect of downsampling from $(303, 303)$ to $(64, 64)$. This downsample reduces the number of pixels in the image by 95% but still retains the essential information of the image. 66
5.14 Epoch loss graph for POD mode 1 67
5.15 Epoch loss graph for POD mode 2 67
5.16 Cropped epoch loss graph for POD mode 1 67
5.17 Cropped epoch loss graph for POD mode 2 67
5.18 Mean relative average error comparison between training set and validation set. 69
5.19 Standard deviation of MRAE comparison between the training set and validation set. 70
5.20 The normalised histogram showing the distribution of MRAE error for the first POD mode, separated between the training set and validation set. 70
5.21 The normalised histogram showing the distribution of MRAE error for the second POD mode, separated between the training set and validation set. 71

5.22 A histogram showing the error distribution against the number of obstacles in the array for the first POD mode. 71
5.23 A histogram showing the error distribution against the number of obstacles in the array for the second POD mode. 72
5.24 Best ML-predicted POD mode for POD mode 1. 73
5.25 Best ML-predicted POD mode for POD mode 2. 73
5.26 Simulation 332 74
5.27 Simulation 33 74
5.28 POD mode 1 comparison for array 332. 74
5.29 POD mode 2 comparison for array 332. 75
5.30 POD mode 1 comparison for array 33. 75
5.31 POD mode 2 comparison for array 33. 75
5.32 The averaged POD mode across the first POD mode. 80
5.33 The averaged POD mode across the second POD mode. 80
5.34 The SSIM measure, which is used to evaluate the likeness of the image and is compared to the MSE measure. Used with permission from Professor Wang, at the University of Waterloo. 83
5.35 This figure shows the comparison between the MSE loss function and the SSIM loss function for the first POD mode. 85
5.36 This figure shows the comparison between the MSE loss function and the SSIM loss function for the second POD mode. 85
5.37 The normalised histogram for the first POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 86
5.38 The normalised histogram for the second POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 87

A.1 Third POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 93
A.2 Fourth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 93
A.3 Fifth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 93
A.4 Sixth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 93
A.5 Seventh POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 94
A.6 Eighth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. 94

A.7 Ninth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. . . . 94
A.8 Tenth POD mode of the simulated flow through an array of obstacles seen in figure 5.7. . . . 94
A.9 Epoch loss graph for POD mode 3 . . . 95
A.10 Epoch loss graph for POD mode 4 . . . 95
A.11 Epoch loss graph for POD mode 5 . . . 95
A.12 Epoch loss graph for POD mode 6 . . . 95
A.13 Epoch loss graph for POD mode 7 . . . 95
A.14 Epoch loss graph for POD mode 8 . . . 95
A.15 Epoch loss graph for POD mode 9 . . . 96
A.16 Epoch loss graph for POD mode 10 . . . 96
A.17 Cropped epoch loss graph for POD mode 3 . . . 97
A.18 Cropped epoch loss graph for POD mode 4 . . . 97
A.19 Cropped epoch loss graph for POD mode 5 . . . 97
A.20 Cropped epoch loss graph for POD mode 6 . . . 97
A.21 Cropped epoch loss graph for POD mode 7 . . . 97
A.22 Cropped epoch loss graph for POD mode 8 . . . 97
A.23 Cropped epoch loss graph for POD mode 9 . . . 98
A.24 Cropped epoch loss graph for POD mode 10 . . . 98
A.25 The normalised histogram showing the distribution of MRAE error for the third POD mode, separated between the training set and validation set. . . . 100
A.26 The normalised histogram showing the distribution of MRAE error for fourth first POD mode, separated between the training set and validation set. . . . 101
A.27 The normalised histogram showing the distribution of MRAE error for the fifth POD mode, separated between the training set and validation set. . . . 102
A.28 The normalised histogram showing the distribution of MRAE error for the sixth POD mode, separated between the training set and validation set. . . . 103
A.29 The normalised histogram showing the distribution of MRAE error for the seventh POD mode, separated between the training set and validation set. . . . 104
A.30 The normalised histogram showing the distribution of MRAE error for the eighth POD mode, separated between the training set and validation set. . . . 105
A.31 The normalised histogram showing the distribution of MRAE error for the ninth POD mode, separated between the training set and validation set. . . . 106

A.32 The normalised histogram showing the distribution of MRAE error for the tenth POD mode, separated between the training set and validation set. . . . 107
A.33 A histogram showing the error distribution against the number of obstacles in the array for the third POD mode. . . . 108
A.34 A histogram showing the error distribution against the number of obstacles in the array for the fourth POD mode. . . . 109
A.35 A histogram showing the error distribution against the number of obstacles in the array for the fifth POD mode. . . . 109
A.36 A histogram showing the distribution of error against the number of obstacles in the array for the sixth POD mode. . . . 110
A.37 A histogram showing the distribution of error against the number of obstacles in the array for the seventh POD mode. . . . 110
A.38 A histogram showing the distribution of error against the number of obstacles in the array for the eighth POD mode. . . . 111
A.39 A histogram showing the distribution of error against the number of obstacles in the array for the ninth POD mode. . . . 111
A.40 A histogram showing the distribution of error against the number of obstacles in the array for the tenth POD mode. . . . 112
A.41 Best ML-predicted POD mode for POD mode 3. . . . 113
A.42 Best ML-predicted POD mode for POD mode 4. . . . 113
A.43 Best ML-predicted POD mode for POD mode 5. . . . 113
A.44 Best ML-predicted POD mode for POD mode 6. . . . 114
A.45 Best ML-predicted POD mode for POD mode 7. . . . 114
A.46 Best ML-predicted POD mode for POD mode 8. . . . 114
A.47 Best ML-predicted POD mode for POD mode 9. . . . 114
A.48 Best ML-predicted POD mode for POD mode 10. . . . 115
A.49 POD mode 3 comparison for array 332. . . . 116
A.50 POD mode 4 comparison for array 332. . . . 116
A.51 POD mode 5 comparison for array 332. . . . 116
A.52 POD mode 6 comparison for array 332. . . . 117
A.53 POD mode 7 comparison for array 332. . . . 117
A.54 POD mode 8 comparison for array 332. . . . 117
A.55 POD mode 9 comparison for array 332. . . . 117
A.56 POD mode 10 comparison for array 332. . . . 118
A.57 POD mode 3 comparison for array 33. . . . 119
A.58 POD mode 4 comparison for array 33. . . . 119
A.59 POD mode 5 comparison for array 33. . . . 119
A.60 POD mode 6 comparison for array 33. . . . 120
A.61 POD mode 7 comparison for array 33. . . . 120
A.62 POD mode 8 comparison for array 33. . . . 120
A.63 POD mode 9 comparison for array 33. . . . 120

A.64 POD mode 10 comparison for array 33. 121
A.65 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 1 for case 332. 123
A.66 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21 for case 332. 123
A.67 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41 for case 332. 123
A.68 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61 for case 332. 123
A.69 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81 for case 332. 124
A.70 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101 for case 332. 124
A.71 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121 for case 332. 124
A.72 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141 for case 332. 124
A.73 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 1 for case 33. 125
A.74 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21 for case 33. 125
A.75 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41 for case 33. 125
A.76 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61 for case 33. 125
A.77 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81 for case 33. 126
A.78 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101 for case 33. 126
A.79 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121 for case 33. 126
A.80 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141 for case 33. 126
A.81 An instantaneous snapshot of the turbulent fluid flow at timestep 1 for array 332. The ML-predicted POD modes are rescaled to match the true POD modes. 127
A.82 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21. 127
A.83 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41. 127
A.84 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61. 127

A.85 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81. . . . 128
A.86 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101. . . . 128
A.87 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121. . . . 128
A.88 An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141. . . . 128
A.89 The averaged POD mode across the third POD mode. . . . 129
A.90 The averaged POD mode across the fourth POD mode. . . . 129
A.91 The averaged POD mode across the fifth POD mode. . . . 129
A.92 The averaged POD mode across the sixth POD mode. . . . 129
A.93 The averaged POD mode across the seventh POD mode. . . . 129
A.94 The averaged POD mode across the eighth POD mode. . . . 129
A.95 The averaged POD mode across the ninth POD mode. . . . 130
A.96 The averaged POD mode across the tenth POD mode. . . . 130
A.97 This figure shows the comparison between the MSE loss function and the SSIM loss function for the third POD mode. . . . 132
A.98 This figure shows the comparison between the MSE loss function and the SSIM loss function for the fourth POD mode. . . . 132
A.99 This figure shows the comparison between the MSE loss function and the SSIM loss function for the fifth POD mode. . . . 133
A.100 This figure shows the comparison between the MSE loss function and the SSIM loss function for the sixth POD mode. . . . 133
A.101 This figure shows the comparison between the MSE loss function and the SSIM loss function for the seventh POD mode. . . . 134
A.102 This figure shows the comparison between the MSE loss function and the SSIM loss function for the eighth POD mode. . . . 134
A.103 This figure shows the comparison between the MSE loss function and the SSIM loss function for the ninth POD mode. . . . 135
A.104 This figure shows the comparison between the MSE loss function and the SSIM loss function for the tenth POD mode. . . . 135
A.105 The normalised histogram for the third POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. . . . 137
A.106 The normalised histogram for the fourth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. . . . 138

A.107 The normalised histogram for the fifth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 139
A.108 The normalised histogram for the sixth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 140
A.109 The normalised histogram for the seventh POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 141
A.110 The normalised histogram for the eighth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 142
A.111 The normalised histogram for the ninth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 143
A.112 The normalised histogram for the tenth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set. 144

List of Tables

4.1 POD mode strengths for the flow past a cylinder at Re=100. The first five streamwise POD modes contributes to 99% of the energy of the streamwise velocity component, whereas the first five transverse POD modes contributes to 89.65% of the transverse flow. 45

5.1 Simulation parameter table. 57

1 Introduction

> *The test of science is its ability to predict.*
>
> **— Richard Feynman**
> (Theoretical physicist)

Fluid flow is all around us. It is there when the clouds drift along the sky. It is there when the tides of the sea batter the beaches, and it is there when rockets launch for the stars, fluid flow exists all around us. Fluid dynamics, the study of fluid flow, is crucial to many industries and has contributed significantly to the advancement of human technology. There are still a lot of significant applications today that call for a deeper comprehension of fluid dynamics. According to the 2001 UK national fluid dynamics report by the UK Fluids Network [30], fluid dynamics is a £13.9 billion industry in the UK, employing 45,000 people across 2,300 businesses. If we include businesses that use fluid dynamics, this figure rises considerably to £200 billion and employs over 500,000 people, demonstrating how beneficial and important the discipline of fluid dynamics is to industry and the economy.

The fluid flow around high-rise buildings is an example of the usage of fluid dynamics for industrial purposes. Civil engineers must understand how the air flows around their buildings since the airflow can cause the building to wobble and suffer from resonance. When air flows past a single or a group of high-rise structures, the fluid flow's wake can leave behind a pattern of vortices shed from the building or buildings. This is known as the von Karman vortex street, and the pressure surrounding the structure can cause an oscillating force to be applied to the building. To minimise resonance, civil engineers must design their structures so that the natural frequency of the structure does not match the forcing frequency of the airflow. As seen in figure 1.1, the Burj Khalifa is a notable illustration of this, since the designers purposefully adjusted the cross-section of the building and tapered it such that the shape and size of the cross-sectional area changes as the skyscraper rises. Tapering the building permits the vortex shedding frequencies to alternate, disrupting the development of vortices in the wake of the fluid flow [1].

The von Karman vortex street is well seen in figure 1.2 where the terrain of the Juan Fernandez Islands is slowing the wind and forcing it to flow around the islands, so forming a series of alternating and oscillating regions of clear sky within a sea of

Fig. 1.1.: The Burj Khalifa. Based on the Hymenocallis Flower, the Burj Khalifa has three 'wings' which surface area progressively gets smaller as the building gets higher. As the cross-section of the building changes, this affects the wind flow and disrupts resonance from occurring. Picture taken by Shahin Ghanbari.

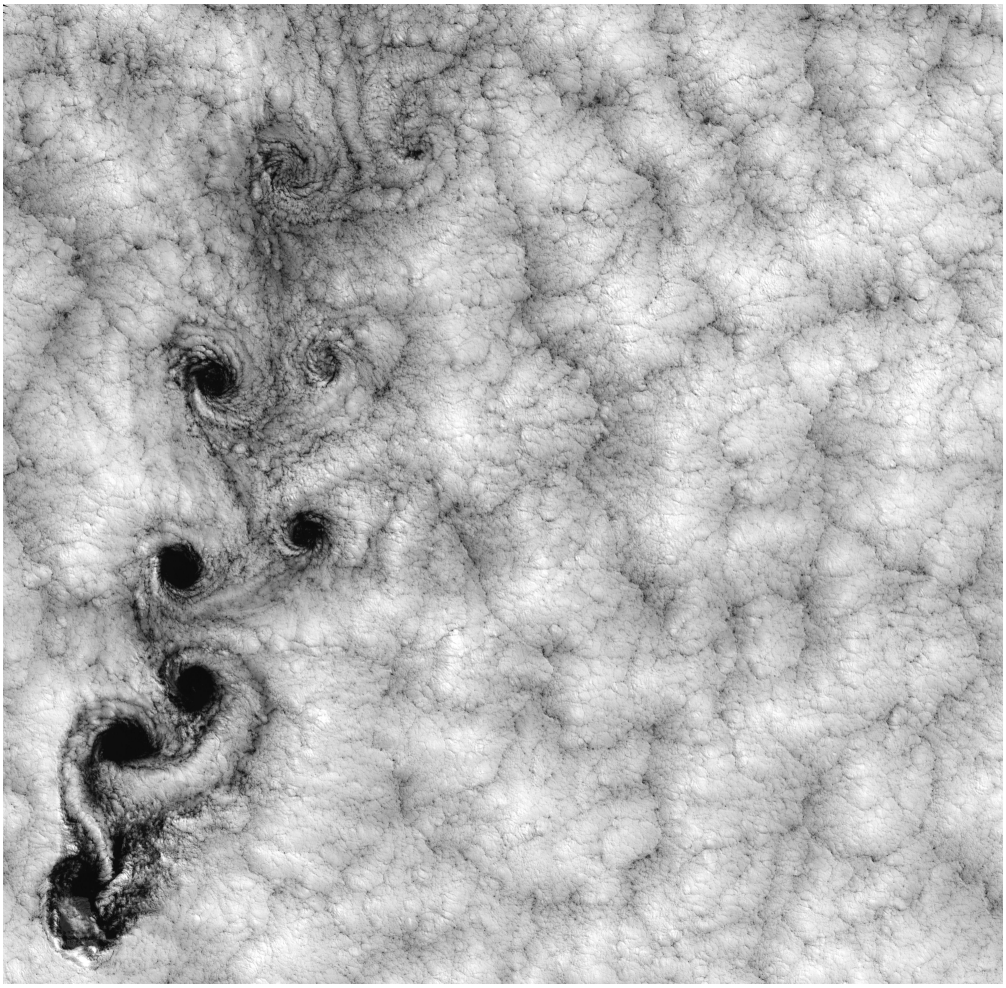

Fig. 1.2.: Picture taken by Landsat 7 shows the clouds off the coast of Chile, near the Juan Fernandez Islands (commonly known as the Robinson Crusoe Islands), on September 15, 1999. Here the von Karman vortex street is clearly seen.

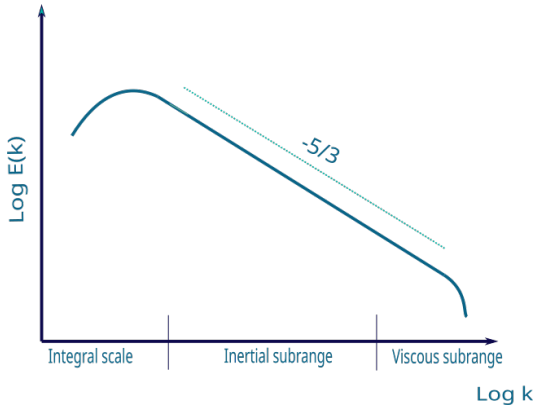

Fig. 1.3.: The energy cascade. k is the wavenumber and $E(k)$ represents the energy of the wake.

clouds. As each region of clear sky gets farther and further away, the wakes dissipate steadily into entropy. This figure illustrates the energy cascade, one of the primary pillars of turbulence. (sometimes called the Richardson-Kolmogorov cascade). The energy cascade is beautifully described in this poem by Richardson.

> ***"Big Whirls have little whirls,***
> ***that feed on their velocity;***
> ***and little whirls and lesser whirls,***
> ***and so on to viscosity."***
> **- Lewis Fry Richardson**

The energy cascade describes the energy transfer between larger and smaller wakes. As each wake fragments into smaller wakes, its energy follows the energy cascade, as seen in Figure 1.3. At inertial subranges, the effect of dissipation is extremely weak, so wake disintegration is caused by vortex stretching. As each wake dissipates into ever-smaller wakes, the effect of dissipation eventually increases until the wake is so small that it dissipates into entropy. One key result was derived by Onsager [52] [12] which found that the energy cascade flows down in the power of $-5/3$. Onsager's work was built on Ricardson's description of the energy cascade and Kolmogorov's [32] derivation the power-law forms describing the energy cascade in 1941.

An additional lesson we can learn from this figure is the self-similarity effect. A object that looks roughly the same at any length scale is said to be self-similar. We observe these effects on larger scales, such as galaxies as seen in figure 1.5, and smaller scales, such as the flow of small streams as seen in figure 1.6. This self-similarity effect is prevalent in turbulent fluid flow, but it can be observed in other parts of nature as well such as coastlines of a country or how the branches of a tree branch out. As one looks at the tree as seen in figure 1.7, and gradually zoom

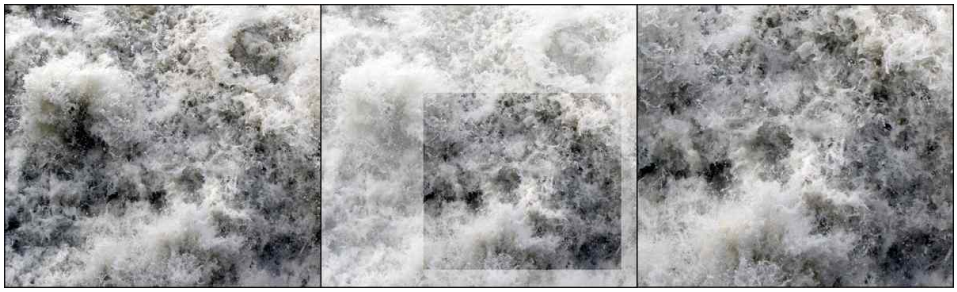

Fig. 1.4.: Observed self-similarity in turbulent fluid flow. The image on the left is the original image, while the image on the right has been magnified, as shown in the image in the middle.

Fig. 1.5.: This image captured by the Hubble Space Telescope depicts Sh 2-106. A newly-formed star known as S106 IR is responsible for the hourglass-shaped gas cloud and the visible turbulence within.

Fig. 1.6.: The Lenaelva River in Norway, near Skrei. It is rotated to form a parallel with figure 1.5. The river's water is falling over a waterfall and causing turbulence. This type of pattern seems to occur everywhere in nature.

closer into the branches, the level of detail increases, replicating the features seen at larger scales. We call these structures fractals, a term coined by Mandelbrot in 1975 [44].

Fractals are characterised, in part, by the fractal dimension, as described in appendix **??**, and the fractal dimension exhibits a power law distribution, the same as the energy cascade. The scale invariance of the power law distribution is useful for modelling turbulent fluid flow [13] as turbulence tends to be scale invariance as well. Due to this, fractals has played a role in producing models describing the fragmentation of larger wakes to smaller wakes such as the β model or the multifractal model [16].

Everywhere we look in nature, we tend to see fractals everywhere. Yet, man-made objects do not tend to be fractals, and the flow around these objects is frequently

Fig. 1.7.: Fractals seen in nature. Notice how each branch looks similar to each sub-branch and that each progressive layer contains more similar detail.

examined in detail. What is intriguing, however, are the turbulent flow through fractal geometries. Marlow, Brevis, and Nicolleau [45], as well as Higham and Vaidheswaran [27], studied the effect of turbulent wakes generated by the flow through fractal multi-scale structures. Marlow et al. experimentally examined turbulent flows through three-dimensional, multi-scale porous obstacles, specifically the flow through the Sierpinski carpet as seen in figure 1.8, and discovered a significant deviation from Kolmogorov's $-5/3$ power law. They performed a power spectrum density analysis on the data and discovered that the flows through the Sierpinski carpet obstacles instead followed a -2 or $-7/3$ power law. Similar results have been reported earlier such as the work by Mazellier and Vassilicos [46]. The imposition of boundary conditions in a particular configuration appears to transform Kolmogorov's $-5/3$ law into either a -2 or $-7/3$ power law.

Concerning the geometric structure of an obstacle undergoing turbulent fluid flow, there are two key questions. Which geometries exhibit these behaviours and why is this the case? How can we parameterize these geometries so that they can be utilised in mathematical analysis?

Marlow et al. attempted to quantify the fractal geometric structure using four unique geometric parameters. The first geometric geometric parameter is the void fraction ϕ which is defined as

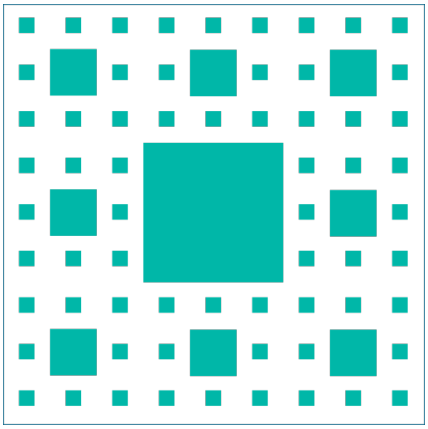

Fig. 1.8.: The Sierpinski carpet after three iterations. A well-known fractal shape in which eight small squares are placed around each large square at each iteration. Each smaller square is a third the size of its larger counterpart. Following each iteration, eight smaller squares are placed around each small square, and this process is repeated onto infinity.

$$\phi = \frac{\text{Volume of voids in the structure}}{\text{Total volume}} \tag{1.1}$$

which represents the porosity of the structure. These measurements may only have secondary roles for analysing turbulent fluid flow as they only parameterise the fractal structures and have no correlation with fluid dynamics at this time. Intuitively, these measurements could be relevant when the flow between obstacles is of interest. The frontal surface area, total perimeter, total number of edges, and many other methods of measuring the fractal structure may also be applicable measures for turbulent fluid flow.

This complicates the investigation of these non-equilibrium turbulent flows. Although there are numerous known fractal structures that can be analysed, there are an infinite number of ways in how these structures can be arranged. These additional complexity poses significant new modelling challenges for these flows. While efforts are concentrated in multiple areas, we propose developing a neural network based on Deep Learning that can understand the structure's geometric data and predict turbulence.

In recent years, the expansion of data has been pervasive across all scientific and commercial fields, and as a result, the use of machine learning has become increasingly popular [9]. Machine learning is a subfield of artificial intelligence that focuses on the creation of models and algorithms that can create approximate predictive functions from data [61]. And a popular subset of Machine learning is Deep Learning, which has become the poster child of machine learning methods. Deep learning is gaining popularity due to the accuracy of the predictions and the

simplicity of the model. Deep learning models consist of a series of 'layers' that, individually, can learn simple relationships but, collectively, can learn complex relationships.

Traditionally, fluid dynamics has generated an abundance of data, either from experiments, field measurements or simulations. This synergy between fluid dynamics and machine learning has motivated researchers to tackle challenges in fluid dynamics using machine learning models and applying them to situations such as turbulence closure modelling, reduced-order modelling, or accelerating computational fluid dynamic (CFD) simulations.

One such example is the work by Ling et al. (2016) [42] who used a multilayer preception (MLP) network, a type of deep neural network, to model the Reynolds stress anisotropy tensor used in Reynolds-averaged Navier-Stokes (RANS) simulations. The aim is to augment the use of the turbulence models in RANS simulations and they found that their model made significant improvement to the simulation compared to linear eddy viscosity and nonlinear eddy viscosity models. Miyanawala and Jaimana (2017), [49] presents an efficient method for model reduction of the Navier-Stokes equations for unsteady flow problems using deep learning. They show the predictive ability of the model they trained, which predicts the force coefficients for a set of bluff bodies and determined the predicts with a maximum relative error of less than 5% whilst being faster than conventional methods by almost an order of four magnitudes.

An interesting application of machine learning in fluid dynamics is the work by Guo et al. [22]. Using a convolutional neural network, Guo et al. predicted the steady flow past a series of regular and irregular shaped bluff bodies. They showed that the model predicted the laminar flow around the bodies with a high accuracy whilst achieving a 2-4 orders of magnitude speed up compared to traditional CFD methods. Although Guo et al.'s work was for steady flow, this result motivated us to develop a similar model but capable of predicting certain features of the turbulent flow instead.

Another work with convolutional neural networks is by Murata et al. [**murata_fukami_fukagata_2020**] who showed that a Convolutional autoencoder can be used to decompose flow fields into more efficient nonlinear mode decompositions that has lower L2 reconstruction losses compared to traditional methods like POD. They used a convolutional neural network with 28 layers and 10,000 samples for the dataset, and showed that convolutional neural networks has the potential to be used for developing more efficient methods of controlling flows.

Convolutional autoencoders can only produce a snapshot of the velocity field, hence their work on predicting steady flow. We wish to understand how information concerning the geometric arrangements of obstacles affect turbulent flows. As turbulence is transient and the convolutional autoencoder produces a snapshot, we plan to devote our efforts to build a model to predict the proper orthogonal decomposition (POD) modes of turbulence from a series of still snapshots containing the geometric information of the obstacles. It is then possible to recreate recreate a surrogate flow which capture some key features of the turbulent flow around an object given a series of POD modes as shown by Baldi and Hornik (1989) [3].

The application potential of this model is vast. In addition to assisting us in comprehending how a particular arrangement of obstacles modifies the fundamental pillars of turbulence, it enables us to rapidly examine turbulent flow past a variety of geometries and predict the flow characteristics. If this model replicates the increase of speed up by 2-4 magnitudes like Guo et al.'s model, the computational resources and time intensity of the work will be drastically reduced.

We intend to create this model by first creating the dataset using CFD and then determining the POD modes of the dataset. Due to its efficiency and speed, simulations will be conducted using the Lattice Boltzmann method (LBM), and the POD modes will be calculated using singular value decomposition (SVD).

The geometric information of the obstacles in each simulation will be parameterised and stored in a separate dataset. After generating the two datasets, we train the model to learn the mapping between the geometry and the POD modes of the turbulent fluid flow.

1.1 Outline

The outline for this thesis are as follows,

- In chapter 2, the basics of machine learning and convolutional autoencoders will be discussed. We will discuss how these models work, to provide background information to the predictive model.

- In chapter 3, the foundations of the Lattice Boltzmann method will be discussed. The quality of the prediction of the model is dependent on the quality of the simulations used to produce them so understanding the methods underpinning the simulation is important.

- In chapter 4, we discuss how the POD modes are calculated and how they can be used to characterise the structures of the turbulent fluid flow with some examples.

- In chapter 5, we will discuss how we combined all these methods to produce our predictive model and discuss the results of the model produced.

- Finally in chapter 6, we will conclude this thesis.

2 Machine Learning in Fluid Dynamics

> *People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world.*
>
> **— Pedro Domingos**
> (Professor Emeritus of Computer Science and Engineering at the University of Washington)

2.1 Introduction

In 2015, the Go world champion Lee Sedol was defeated in a thrilling five-game series by the computer programme AlphaGo, 4-1. Due to the complexity of the game at the time, it was widely believed that machine learning models could not yet defeat humans in a game of Go. Unlike previous older computer programs such as Deep Blue, which beat the World Chess Champion Garry Kasparov in 1997, AlphaGo is different in that it did not solely use a brute force method to search for the most optimal move as Deep Blue did[1]. Instead, the distinction between AlphaGo and Deep Blue was that AlphaGo calculated the probability of optimal moves and ignored losing ones in its search algorithm. This enabled AlphaGo to reduce the complexity of its search algorithm and concentrate its limited computational resources on determining the optimal move. This distinction in approach is notable. It may be computationally costly to use brute-force methods to obtain direct results, depending on the task at hand; therefore, it may be pragmatic to use an approach that approximates the ideal solution. Deep learning enables the model to understand the representation of the data as opposed to memorising the data, enabling it to make much more accurate predictions than other machine learning techniques.

[1]Deep blue used an Alpha-beta pruning search algorithm. The algorithm attempts to maximise a predefined score while disregarding moves that result in a lower score. Due to the fact that Chess is not a solved problem, i.e there is no 'ideal game', the algorithm used is a deterministic method for determining the optimal moves.

Deep learning's success is a result of the continued growth of computing power and the abundance of large datasets. There are three main sub-branches to deep learning, supervised learning, unsupervised learning and reinforcement learning. Supervised learning consists of algorithms used to train a model given a known input and output. Unsupervised learning consists of algorithms that do not require an output dataset, and instead obtain useful results such as the Principal component analysis (PCA) components or clusters using the K-means clustering algorithm. The majority of machine learning applications are founded on supervised learning.

Deep neural networks (DNN) can be seen as a model with typically a large number of parameters which takes some input and approximates the output. These parameters are built using a series of 'artificial neurons' which individually aims to solve a simple equation,

$$\vec{y} = \sigma(\vec{\omega}\vec{x} + \vec{b}), \tag{2.1}$$

where $\vec{x}$ and $\vec{y}$ represents the input and output respectively, ω represents the weight, b represents the bias of the neuron and σ represents the activation function. The activation function is typically a non-linear function. Each individual artificial neuron can be stacked into a 'layer' which acts as

$$\vec{y_i} = \sigma \sum (\vec{\omega}\vec{x}_{ij} + \vec{b_j}). \tag{2.2}$$

Each layer can be stacked on top of each other to form a series of layers and given a DNN a sufficient number of layers and sufficient number of artificial neurons per layer, it can be shown via the the universal approximation theorem that it is possible to construct a neural network that can predict a given function if a DNN has a sufficient number of layers of artificial neurons or if the width of the layer is sufficiently large [26]. This is the strength of Deep Learning and this strength has contributed to Deep learning's recent popularity.

However, there are two substantial disadvantages. The first is that the universal approximation theorem does not instruct how to construct an ideal neural network so there is no information regarding the optimal method for determining these parameters. A second weakness is the curse of dimensionality. Richard Bellman [4] coined the term "curse of dimensionality" to describe the difficulty of using a brute-force algorithm to optimise a function with many variables. Due to the curse of dimensionality, as the number of required dimensions increases, so does the amount of training data required, as the data becomes more "sparse" as the number of required dimensions increases. Therefore, caution is required when developing a predictive deep learning model, as the sparsity of the dataset may prevent the model from training.

Previously, we contrasted the deterministic approach of the Deep Blue program with the probabilistic approach of AlphaGo. These approaches can also be viewed through the lens of computational fluid dynamics. CFD simulations are used to model turbulent fluid flow, with Direct Numerical Simulation (DNS) being the most accurate method. These simulations are computationally intensive because they must predict the movement of the fluid at all scales, including the smallest which inevitably means that the resolution of the simulation has to be dense. This is comparable to the deterministic brute force algorithm mentioned earlier.

In contrast, one can approach this problem though the probabilistic point of view.

Here, machine learning models can play an important role. Typically, a disadvantage of machine learning methods is that they are data-driven and therefore require an abundance of data to produce accurate models. In the case of fluid dynamics, however, this limitation is neglected by the abundance of available simulation and experimental data. As shown by a review paper by Brunton et al. (2020) [9], machine learning is currently being used to solve many problems in CFD such as solving the closure problem in Reynolds-averaged Navier-Stokes equations by predicting the Reynolds stress [37]. As a result, we can approach this from a probabilistic point of view.

Application of deep learning techniques to CFD appears promising. DeepMind's work (Ravuri et al.) [55] is an example of the use of deep learning models in the context of fluid dynamics. Due to the computational costs required to simulate high quality simulations in real time, conventional near time precipitation nowcasting simulations cannot predict immediate and near future precipitation accurately in real time.

Using convolutional neural networks, Guo et al. [22] demonstrated that it was possible to reproduce the steady laminar flow field around buff bodies. Their models produced the results of streamwise velocity component and transverse velocity component separately, which they later combined to produce the final result. They introduced the use of the signed distance function (SDF) to characterise the geometry in the context of fluid dynamics, the use of the encoder to analyse the geometry, and the use of the decoder to understand the encoded data and predict the steady flow. We explain in more detail about the SDF in subsection **??** and the encoder and decoder in subsection 2.3.

They experimented with two model architectures, a 'shared' encoder model which used one encoder to produce a encoding that is used to predict both the streamwise and transverse velocity components, and a 'separate' encoder model where two models were used to independently predict either the streamwise or transverse

velocity components without using the same encoder. The two models were trained on two separate datasets, containing either 100,000 samples of five unique 2D primitives: triangles, quadrilaterals, pentagons, hexagons and dodecagons, or a 2D dataset containing car geometries and airfoils. The number of samples for the 2D dataset containing car geometries and airfoils is not known.

The accuracy of the models varied. Using a root mean relative squared error to calculate the error across the entire velocity field, defined as

$$\text{err}^n(i,j) = \frac{\sqrt{(v_{ij}^x - \hat{v_{ij}^x})^2 + (v_{ij}^y - \hat{v_{ij}^y})^2}}{\sqrt{(v_{ij}^x)^2 + (v_{ij}^y)^2}}, \tag{2.3}$$

where v_{ij}^x and v_{ij}^y represents the velocity values at each pixel, and $\hat{v_{ij}^x}$ and $\hat{v_{ij}^y}$ represents the predicted velocity values at each pixel. The accuracy varied from 1.76% to 3.08% for the dataset containing primitive shapes, and 9.04% to 16.53% for the 2d dataset containing car geometries and airfoils. Their method allows for the prediction of fluid flow given a certain shape and gives a speed increase of three to four orders compared to CFD simulation. Guo et al. showed that the use of the SDF to represent the geometry compared to binary reduced the error significantly, with an error reduction of 80.71% to 3.08% being the most drastic difference. The reason behind the increase in effectiveness of the SDF is that the SDF representation contains global geometric information, compared to the binary representation which only contains local geometric information.

Bhatnagar et al. [7] also had a similar idea in creating an CNN model but were interested in creating a model that could predict velocity flow fields and pressure field given varying airfoil shapes, Reynolds numbers and angles of attack. They used four different high Reynolds numbers, three unique airfoils and twenty-one different angle of attacks and also adopted a similar model architecture to Guo et al., using a SDF for the input, an encoder to interpret the SDF and decoder to decode the output into the velocity or pressure field. Their dataset contained 252 samples which contained the results of the RANS simulations conducted for the work. A noteworthy experiment conducted by Bhatnagar et al. is that they used a shared encoder shared decoder architecture, rather than the shared encoder separate decoder and separate encoder separate decoder architecture experimented by Guo et al. They showed that the shared decoder architecture performed better compared to the separate decoder architecture, where the MAPE error reduces by more than 10%, from 24.97% to 14.82% for the pressure field error in the wake region of the flow field. They noticed that their results varied depending on the region of the flow field with the model struggling with predicting the wake region as accurately compared to the rest of the flow field. Overall they report an MSE of less than

0.1 over the entire flowfield for all their predictions. Like before, Bhatnagar et al. reports a speed up of a magnitude of 4 orders compared to using a RANS solver.

In this chapter, we will discuss the the fundamentals of deep learning and the different architectures of deep learning that may be applicable to our model. Finally, we will describe the basics of the encoder and decoder of the convolutional neural network.

2.2 Deep learning basics

As stated previously, the goal of deep learning models is to estimate the mapping between an input $\vec{x}$ and an output $\vec{y}$ using a series of artificial neurons. There are other models for deep learning, but we will begin with the feedforward neural network (FNN) because its architecture is straightforward and provides a solid foundation for understanding other neural networks. Later in this chapter, the convolutional neural network will be described in detail.

2.2.1 Feedforward Networks (FNN) overview

The FNN employs a network of artificial neurons arranged in layers called fully connected layers. Each neuron utilises the equation,

$$\vec{z} = \vec{\omega}\vec{x} + \vec{b}, \quad \vec{y} = \sigma(\vec{z}). \tag{2.4}$$

where $\vec{\omega}$ represents the weight, $\vec{b}$ represents the bias and σ represents a non-linear function. The weight $\vec{\omega}$ and bias $\vec{b}$ are referred to as the parameters of a neuron. The intermediate term z is the output of the neuron without the activation function and performs a unique role in training the model, which is mentioned in subsection 2.2.2. For Deep Learning models to satisfy the universal approximation theorem, non-linearity is required so non-linear terms σ are frequently employed to bring non-linearity into the model. Nonlinearity being defined as a function that violates the superposition principle and/or homogeneity. Multiple layers may be added to the model to increase the number of possible 'connections' per layer, hence boosting the predictive capacity of the model. The input $\vec{x}$ is altered each time it passes a layer, from the input to the output of the model. This is referred to as forward propagation and is depicted in the image **??**. The number of weights and biases are equal to the number of connections between the the two adjacent layers.

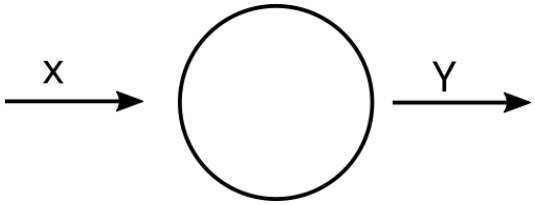

Fig. 2.1.: A single artificial neuron. This diagram represents the mathematical function of $\vec{y} = \sigma(\vec{\omega}x + \vec{b})$.

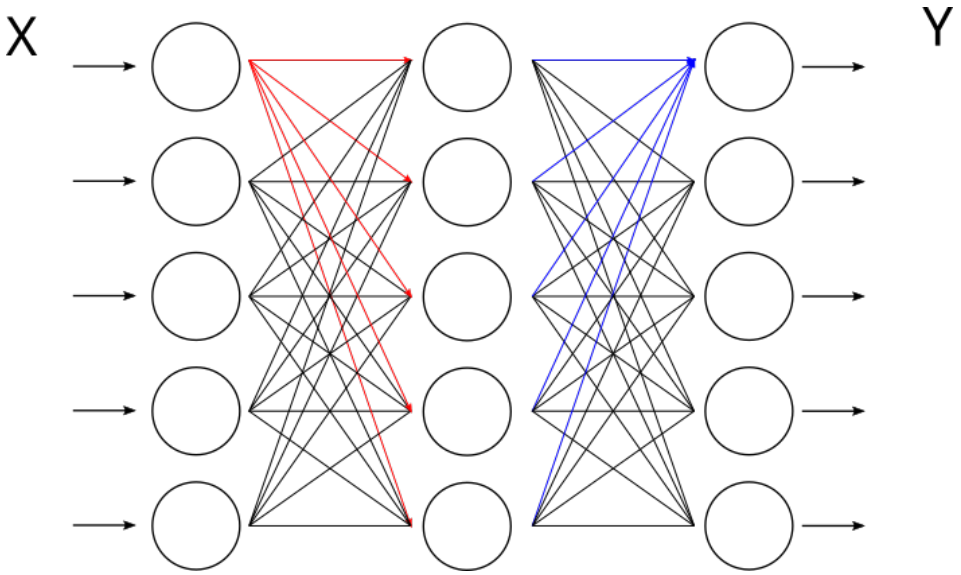

Fig. 2.2.: A deep learning model with three layers, 1 input layer $\vec{x}$, 1 hidden layers and 1 output layer $\vec{y}$. Each layer does a weighed sum of the previous layer.

We know from the universal approximation theorem [11] that a network of neurons with a sufficient number of layers and a non-polynomial activation function may approximate any Borel-measurable function [40]. As a result, choosing the optimal parameters for all neurons is the primary concern. This becomes an optimization problem where the model can be trained using an algorithm known as back-propagation [56] and a deep learning optimiser.

2.2.2 Backpropagation

Backpropagation is a training method used to adjust the weights and biases of every neuron in a network using the chain rule. As the goal is for the model to interpret the input X and generate a prediction $\hat{Y}$ such that it is as close as possible to the desired output Y, a loss function can be employed and mathematical optimisation methods can be used to adjust the parameters of every neuron. Many loss functions can be used, such as the squared mean error (MSE)

$$\mathcal{L}_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}(Y - \hat{Y})^2, \tag{2.5}$$

or the mean absolute error (MAE),

$$\mathcal{L}_{MAE} = \frac{1}{n}\sum_{i=1}^{n}|Y - \hat{Y}|. \tag{2.6}$$

Each loss function has its own advantages and disadvantages. For instance, the MSE loss function is more effective than the MAE when training models with a dataset with high variance, whilst MAE is more effective for training models with lower variance [29] . The reason for this is due to the squared term in the MSE. If the difference between $\hat{Y}$ and Y is large, the loss reported by the MSE loss function will be greater than MAE. The opposite is also true, if the difference between $\hat{Y}$ and Y is exceedingly small then the loss reported by the MAE loss function is higher than the MSE loss function.

Once an appropriate loss function has been defined, back propagation can be used to modify the weights ω and biases b of each neuron in the model. Essentially, this procedure follows the chain rule. The objective is to adjust the parameters such that the rate of change of the loss function with respect to the weight or bias is minimised. The weights and biases are adjusted by the equations,

$$\omega_{i+1} = \omega_i - \alpha\frac{\partial\mathcal{L}}{\partial\omega}, \tag{2.7}$$

and

$$b_{i+1} = b_i - \alpha \frac{\partial \mathcal{L}}{\partial b}, \tag{2.8}$$

where α represents the learning rate. The learning rate is a variable that adjusts the rate of change for the weight and bias. The learning rate α can be adjusted by deep learning optimisers as shown in subsection 2.2.4. The lower the learning rate, the slower the training process, but the greater the likelihood that the model will converge on a minimum.

We aim to determine the gradients $\frac{\partial \mathcal{L}}{\partial \omega}$ and $\frac{\partial \mathcal{L}}{\partial b}$. To update each neuron, the chain rule can be used to derive the relation,

$$\frac{\partial \mathcal{L}}{\partial \omega} = \frac{\partial \mathcal{L}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial z} \frac{\partial z}{\partial \omega}, \tag{2.9}$$

which provides the gradient for the weights. For the biases, a similar equation is used,

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial z} \frac{\partial z}{\partial b}. \tag{2.10}$$

The term $\frac{\partial \mathcal{L}}{\partial \hat{Y}}$ can be trivially derived from the loss function and the term $\frac{\partial \hat{Y}}{\partial z}$ can be derived from the activation function. Finally the terms $\frac{\partial z}{\partial \omega}$ and $\frac{\partial z}{\partial b}$ can be derived from equation 2.4 and becomes x and 1 respectively.

2.2.3 Activation functions

The activation function, as mentioned previously, introduces nonlinearity into the model. As demonstrated above, the term $\frac{\partial \hat{Y}}{\partial z}$ is dependent on the activation function used, so in addition to selecting the activation function based on the function's characteristics, one must also select the function based on the activation function's derivative. A list of typical activation functions are listed below. The rectified linear unit (ReLu),

$$\sigma(z) = \max(0, z), \quad \frac{\partial \sigma(z)}{\partial z} = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}, \tag{2.11}$$

the leaky Relu,

$$\sigma(z) = \max(0.1z, z), \quad \frac{\partial \sigma(z)}{\partial z} = \begin{cases} \text{-0.1}, & z < 0 \\ 1, & z \geq 0 \end{cases}, \tag{2.12}$$

the Exponential Linear Units function (elu),

$$\sigma(z) = \begin{cases} \alpha(e^z - 1), & z < 0 \\ z, & z \geq 0 \end{cases}, \tag{2.13}$$

$$\frac{\partial \sigma(z)}{\partial z} = \begin{cases} \alpha e^z, & z < 0 \\ 1, & z \geq 0 \end{cases}, \tag{2.14}$$

the sigmoid function,

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \frac{\partial \sigma(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2}, \tag{2.15}$$

and the tanh function,

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \frac{\partial \sigma(z)}{\partial z} = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2}. \tag{2.16}$$

Note that the discontinuous nature of the Relu activation function and its variants means that those functions are nonlinear as it does not obey the superposition principle.

2.2.4 Deep Learning optimiser

In the final step of the process, the learning rate α is adjusted using a 'learning optimiser'. In practise, neural networks are composed of thousands or millions of neurons, each with its own weights, biases, and the gradients of each neuron will have to be calculated. Using the gradient descent method, defined as

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta) \tag{2.17}$$

can be computationally intensive. The computational cost is $O(m)$ and due to the data driven nature of Deep Learning, the computational cost can be prohibitive. Therefore this motivates the use of learning optimisers. A second reason for employing learning optimisers is to direct the training procedure towards the optimal local minimum. If the learning rate is too high, the model will be trained to achieve a suboptimal minimum in a shorter amount of time. Similarly, if the learning rate is low, the model will be trained to reach a minimum that is more optimal, but training will take longer. There are numerous learning optimisers available, and each employs a vastly different strategy to better the training process.

The stochastic gradient descent (SGD) algorithm is the standard optimisation technique for deep learning. In this instance, the gradient is viewed as an expectation that can be approximated by determining the gradient of a subset of the dataset as shown by,

$$g = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(x^{(i)}, y^{(i)}, \theta). \tag{2.18}$$

This is typically found by splitting the dataset into 'batches' and obtaining the gradient of one batch. If the size of the batch is equal to the size of the dataset itself then the SGD method is equivalent to gradient descent.

2.3 Convolutional neural networks

Convolutional neural networks (CNN) are a type of deep learning architecture designed to analyse high-dimensional data, such as images or video sequences by analysing the proximity of the data [38].

The traditional feed-forward deep learning models used to process these types of data are typically very large in size. This is because, as depicted in figure **??**, each neuron in a FNN is connected to all neurons in the subsequent layer, and so on. The number of connections between neurons can therefore increase by $O(m \times n)$, where m and n represent the number of neurons in layer i and layer $i+1$, respectively.

CNNs are based on the premise that important information from a large dataset can be inferred from a small subset of the dataset. Therefore, it is ideal to filter out unnecessary information so that the model only handles statistically significant data. The model does this by employing 'filters' (also known as a kernel) which is smaller than the input data and scans through the input data, moving from one locality to the next. The filters contains values that acts similarly to the parameters of a neuron, and these values applies the dot product to each local area to obtain the output data which is called the feature map. This is represented as,

$$Y_{i,j} = \sigma(\sum_{k} X_{i,k} \cdot \omega_{k,j}), \tag{2.19}$$

where $Y_{i,j}$ is local value in the feature map, $X_{i,k}$ is the input data, σ is the activation function and $\omega_{k,j} f$ is the weights of the filter. This method provides very interesting strengths to the neural network. In particular these three concepts, sparse interactions, parameter sharing and equivariant representations [21]. As CNNs restrict the number of possible connections based on the local information in the input data as seen in figure 2.3, the computational demand for a large dataset is reduced as fewer parameters are required and the statistical efficiency of the model is improved.

This method also permits the sharing of parameters, as each parameter now has multiple functions in the model, as opposed to a single function in a feedforward network. Finally equivariance, defined as $f(g(x)) = g(f(x))$, is the ability to understand representations in the data without being linked to the coordinate system.

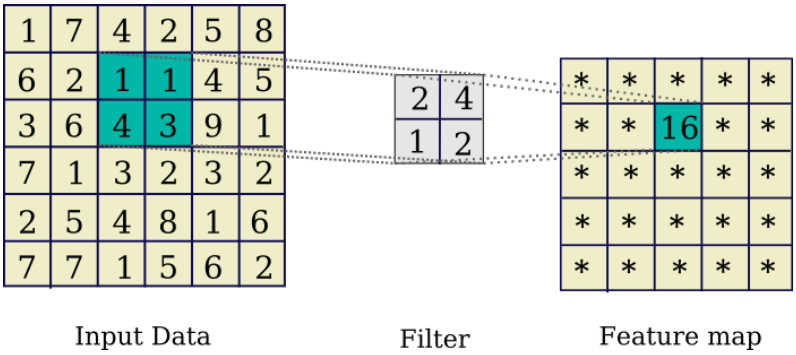

Fig. 2.3.: A convolution operation by a convolutional layer. The filter scans across the input data and selects a n by m array and applies the dot product with the values in the filter to produce a single value. In this case, the filter is 2 by 2 and the stride is 1 by 1.

This means that if a model has learned a particular pattern, it is more likely to understand the same pattern if it is in a different location, which is advantageous for pattern recognition models where the feature might be located in different areas of the image.

In equation 2.19, the use of the bias is not mentioned as there are three methods to handling the bias. The first is to avoid using a bias, as many prominent CNN models, such as ResNet, have chosen to do [24]. The second approach employs "tied biases," which introduces the bias as a scalar such that,

$$Y_{i,j} = \sigma[(\sum_{k} X_{i,k} \cdot \omega_{k,j}) + b], \tag{2.20}$$

where b is the bias. The final method, termed "untied bias," introduces the bias as both a scalar and a tensor, such that

$$Y_{i,j} = \sigma[(\sum_{k} X_{i,k} \cdot \omega_{k,j} + b^t_{i,j}) + b], \tag{2.21}$$

where $b^t_{i,j}$ is the bias for each output location. The size of the filter and the distance it travels (known as the stride) are both adjustable. Increasing the stride enables the filter to bypass certain neurons, thereby reducing the computational demand as fewer neurons are processed and reducing the amount of memory required. Note that the model is applicable for any input data that is arranged in a regular grid structure.

Note the change in dimension between the input data and the resulting feature map as a result of the convolution operation. As shown in figures 2.3 and 2.4, one can either retain this change in dimension or pad the input data with values such as zero to prevent the dimension reduction. These padding are called valid padding and same padding respectively.

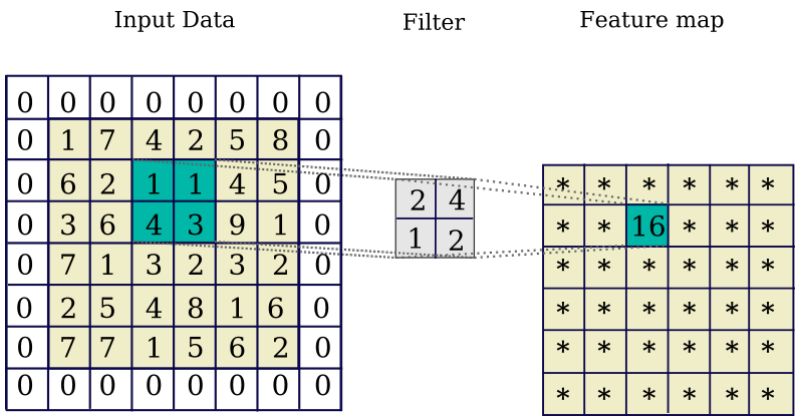

Fig. 2.4.: A convolution operation by a convolutional layer with padding. Arbitrary values, in this case zeros are added outside the input data to prevent the reduction in dimension.

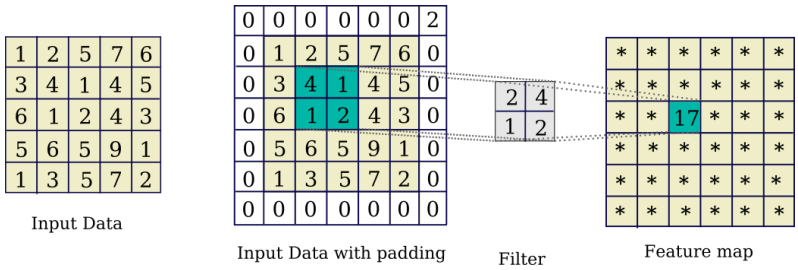

Fig. 2.5.: A transpose convolution operation conducted by a transpose convolutional layer. The extra padding increases the dimension of the input data, and a convolution operation is applied onto the input data with extra padding to produce a feature map with greater dimension compared to the input data.

The change of dimension is represented by the mathematical relationship,

$$\frac{W_i - K + 2P}{S} + 1 = W_{i+1} \tag{2.22}$$

where W_i is the input dimension, W_{i+1} is the output dimension, K is the kernel size, P is the padding and S is the stride.

The opposite operation to convolutional layers, transpose convolutional layers (also known as deconvolution layers) are also possible to implement. The transpose convolutional layers identically to the convolutional layers aside from the first few steps. First, the input image is padded. This is usually done with zeros. If a stride is used then the input data is spaced out as shown in figure 2.5.

The filters then work identically to the deep learning convolution operation shown in equation 2.19. The increase in dimension can be calculated using

$$W_{i+1} = (W_i - 1) * S - 2 * P + (K - 1) + 1. \tag{2.23}$$

2.4 Training

In the previous section, the individual action of each neuron is described. However, the method of determining the parameters for the entire neural network is a different matter.

2.4.1 Initialisation

As a quick review, mathematical optimization methods are used to determine the optimal values for the parameters of each neuron through back-propagation employing gradient descent or a variant of it. These methods can be vectorized such that they are applicable to all neurons. To initiate the training procedure, the weights are initialised by randomly determining their values according to various probabilistic distributions. If the parameters are initialised with an optimal set of values, it significantly reduces training time by preventing the neurons from obtaining extreme parameters which could have a detrimental effect on the model training process. Ideal parameter initialisation also provides numerical stability during training. During training, gradient descent is capable of producing parameters that either explode or vanish to zero. This results in the neuron being either ineffective or too dominant in the model, which reduces the efficacy of connected neurons and, consequently, the model's accuracy. Using the ideal parameters to initialise training prevents this from happening.

Currently, research is still ongoing to understand which probabilistic distribution is ideal so several probabilistic distributions are currently in use by the Deep Learning community. The probabilistic distributions can be classed into three categories, Gaussian distributions, uniform distributions and constant. In addition, the mean and variance of the distribution can be altered. To prevent the parameters from exploding or vanishing, the variance of the parameters can be measured to ensure it does not increase or decrease drastically. This resulted in the creation of two distinct initialisation methods used in Deep learning: the Xavier (also known as the Glorot initialisation) initialisation [20], and later on, the Kaiming (also known as He) distribution [25].

The Xavier initialisation sets the bounds that the initial weight values generated, where the distribution is centered on zero and the standard deviation is bounded by the expression,

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{2.24}$$

where n_i and n_{i+1} is the number of incoming network connections and number of outgoing network connections for a uniform distribution. For a normal distribution, the distribution is again set to zero and the standard deviation is bounded by,

$$\pm\frac{\sqrt{2}}{\sqrt{n_i + n_{i+1}}}. \tag{2.25}$$

Experimentally, the Xavier initialisation is known to be particularly effective for models with a symmetric activation function such as the tanh function or sigmoid function. However, the Xavier initialisation does not work well for non-symmetric activation functions such as ReLU so the Kaiming distribution is often used instead. Here, the standard deviation of the Kaiming uniform distribution are centred on zero and the standard deviation bounded by the equation,

$$\pm\frac{\sqrt{6}}{\sqrt{n_i}}. \tag{2.26}$$

and the standard deviation for the zero-centred Kaiming normal distribution is expressed as,

$$\pm\frac{\sqrt{2}}{\sqrt{n_i}}. \tag{2.27}$$

2.4.2 Learning rate and batch size

Once the weight initialisation is set, obtaining the ideal learning rate is the next step. Adjusting the learning rate can result in a lower generalisation error at a cost of higher computational resources or a higher generalisation error with fewer computational resources used. The dataset used to train the model becomes the space that the model aims to predict in. To confirm that the model is generalising on the dataset and not learning the dataset, the dataset is split into a training set and validation set. The ratio of this split varies between applications and is subject to numerous rules of thumbs.

The training set can be further subdivided into batches. As the dataset is primarily processed by graphical processing units (GPUs), the memory capacity of DRAMs is limited and can only store a limited amount of data. Additionally, it typically takes time to transfer data from the storage space to the DRAM. Therefore, it is prudent to select a batch size that is small enough to fit in the DRAM but not so small as to increase processing time. The batch size can also affect certain algorithms such as the SGD where the algorithm performs gradient descent on a batch rather than the entire training set.

After the entire training dataset is processed and gradient descent is performed, this process is looped again and again, where each loop is called an epoch. After each epoch, the model parameters will adjust and the overall performance of the model may gradually move towards the ideal generalised error, or away from it. After many epochs, the model performance will tend towards some minimum where it may start to overfit on the data.

2.4.3 Standardisation, Normalisation and Batch Normalisation

Scaling the dataset can improve the training process as many learning optimisers and weight intialisations assume that the data in the dataset ranges from $[-1, 1]$ with a variance of 1. Two scaling methods are used, which are called 'normalisation' and 'standardisation'. Normalisation involves scaling the data by dividing by the range of the dataset such that the range of values in the dataset fall between $x \in [-1, 1]$ as mentioned previously. This is expressed as,

$$y = \frac{x - x_{min}}{x_{max} - x_{min}}. \tag{2.28}$$

Normalisation can also refer to scaling the dataset from $[0, 1]$ depending on the properties of the dataset.

The alternative method is standardisation involves shifting the mean of the data to zero, and scaling the standard deviation of the dataset to 1 dividing the dataset by the standard deviation σ as represented as,

$$y = \frac{x - \bar{x}}{\sigma}. \tag{2.29}$$

Standardisation allows a better distribution of the dataset, which improves the training process of the model.

Batch Normalisation is another method that can be used to improve the accuracy of the model. Rather than scaling the dataset, one can scale the values before or after they are processed by the neurons. Confusingly, this method is similar to standardisation rather than normalisation. During each epoch, batch normalisation is applied using equation 2.29 as shown in figure 2.6. This minimises the number of parameters either exploding or vanishing by normalising the parameters by the standard deviation.

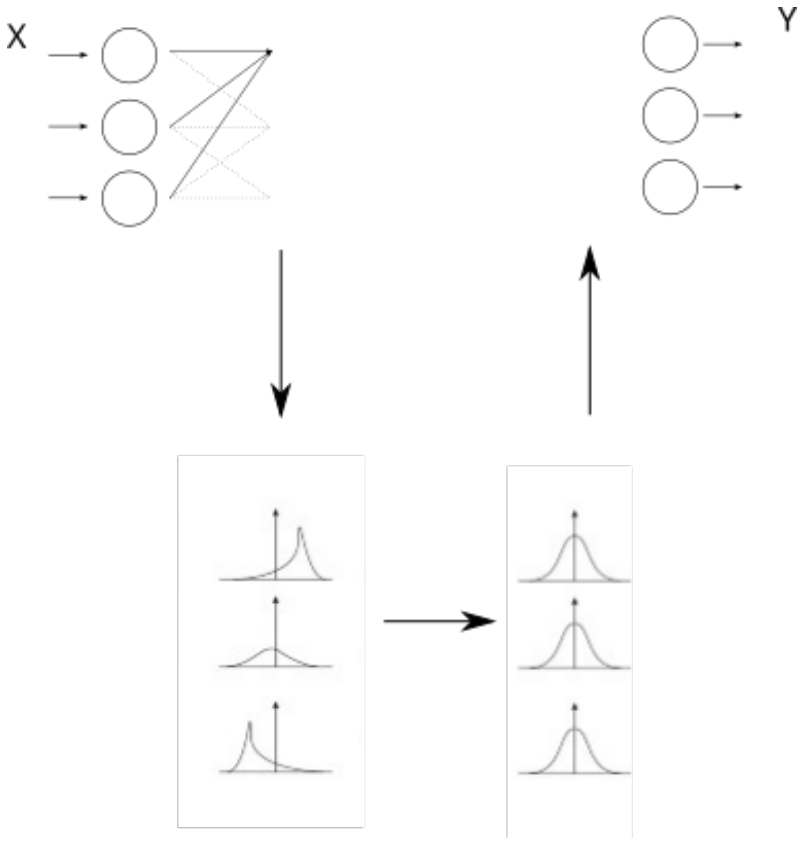

Fig. 2.6.: The batch normalisation process. The output values produced after passing through a layer is standardised and fed into the next layer.

2.4.4 Hyperparameters and Tuner

Optimising the ideal deep learning model relies on experimenting with the model architecture to understand how it is making predictions, and improving either the model architecture or the training process. There are many ways to do this, and the parameters associated with this process are called hyperparameters. Hyperparameters are parameters that are set outside the training process. Example of hyperparameters include the number of neurons in a fully connected layer, the learning rate, deep learning optimisers, the number of filters in a convolutional layer or a transpose convolutional layer. Optimising these hyperparameters can have different affects on the model's predictive ability.

For instance, increasing the number of neurons in a fully connected layer or the number of filters in a convolutional network can increase the model's capacity, or the ability to predict more complex features. This comes at a cost of an increase in computational requirements and the increased likelihood of overfitting. As a result, depending on the training dataset, the model architecture and training process, this can either improve the overall accuracy of the model, or decrease it if overfitting occurs. Similarly, the opposite is also true, reducing the number of neurons will reduce the predictive ability of the model and generalise the prediction, which may be advantageous for overfitted models but simplifies the prediction output of the model. Therefore obtaining the ideal number of neurons or number of filters is a balancing act.

This can be accomplished either manually by examining the model's performance, hypothesising the model architecture's limitations, and improving on them. Manual search provides researchers with an understanding of the model architecture's capability without requiring additional technical methods. However, this method has two significant disadvantages. The first is that it is a time consuming method. Researchers must spend time to analyse the model and hypothesise the potential adjustments to improve the model. The second disadvantage is the difficulty in reproducing results, as the method is not systematic and relies on 'intuition' which is not same for each individual[5].

Adjusting the hyperparameters can be accomplished systematically using a 'tuner'. Tuners are hyperparameter optimisation methods that can be used to determine the ideal hyperparameter values for the model, in other words, 'tuning' the model. There are a few methods for determining the ideal hyperparameters such as grid search, random Search and Hyperband.

Grid search [39] is a method that involves specifying a search space, and intervals to search within the search space for range of values or individual methods for each hyperparameter, and training the model on all possible combinations. This method can be tedious but allows a thorough investigation of the model architectures capability. However, it should be noted that grid search suffers from the curse of dimensionality. For each hyperparameter being adjusted, the computational demand required to tune the model increases by an order of magnitude. This compels the use of other tuners.

Random search [5] is another method has been shown to work better than grid search when the number of hyperparameters is high. Once a search space is defined, random combinations are chosen to train on the model. The reasoning behind why this method is more optimal than grid search is that in practice, many hyperparameters do not alter the performance of the model. Random search directly bypasses the determination of the effectiveness of these parameters and therefore reduces the computational demand required overall. Although the most ideal hyperparameter values may not be found, some acceptable value can be found that produces good results for the model.

The Hyperband [41] tuner is an interesting hyperparameter optimisation method based on random search and uses the concept of resource allocation. The assumption underlying Hyperband is that the most optimal models typically train more quickly in the earlier stages of the training process than in the later stages. Once a search space has been specified, Hyperband will conduct a random search using fewer epochs. The algorithm will then reduce the search space based on the results of the half-trained models and repeat the process using an increasing number of epochs each time. Following a specified amount of search space reduction, the algorithm will then conducts a standard random search to determine the ideal model.

2.4.5 Sample size

For data driven models, the size of the dataset required to train the model is typically large. But it is unknown how many sample are required to train for the model. One reason is the curse of dimensionality. There have been many studies on determining the ideal size of the training set, such as papers from (Figueroa et al. 2012) [14], (Jain and Chandrasekaren 1982) [28], and (Raudys and Jain, 1991)[54]. Though a widely accepted practical method has not emerged, many rules of thumb have emerged. One example is Goodfellow et al.'s book 'Deep Learning' [21], which recommends at least 5000 observations per category for acceptable performance and at least 10,000,000 observations per category to match or beat human performance.

2.4.6 Conclusion

To conclude this chapter, we have explained the basics of deep learning, and introduce the mechanisms that the deep learning model will employ. The aim of this chapter was to provide the foundational context to the convolutional neural network that we aim to use, to predict the POD modes of turbulent fluid flow in order to gain a deeper understanding of how the geometry of the array affects turbulent flow.

3 The Lattice Boltzmann Method

> *Bring forward what is true. Write it so that it is clear. Defend it to your last breath!*
>
> — **Ludwig Boltzmann**

3.1 Introduction

The Lattice Boltzmann method (LBM) is a unique CFD scheme that simulates fluid flow by employing the Boltzmann equation instead of the Navier-Stokes equations. Evolving from the lattice gas automata method, the LBM is based on understanding the molecular dynamics of particles to solve the macroscopic behaviour of fluid flow [10]. What differs LBM from lattice gas automata methods in that it tracks the distribution of particles compared to individual particles, which are tracked in lattice gas methods. In a sense, if the Navier-stokes equations model the macroscopic behaviour of the fluid flow and the dynamics of individual particles are the microscopic behaviour of the fluid flow, then the LBM aims to predict the fluid flow on the mesoscopic scale, in between the microscopic and macroscopic levels. Therefore it is reliant on the Boltzmann equation which is derived from the Kinetic theory of diluted gases, the study of the mesoscopic properties of a large number of particles.

This section introduces how the Boltzmann equation was derived and how the Lattice Boltzmann Method (LBM) works. By understanding how the fundamental equation was derived, it is possible to deduce the LBM's characteristics, which in turn helps us understand the strengths and weaknesses of the method. We avoid lengthy mathematical derivations as more comprehensive explanations are readily available such as the LBM textbook 'The Lattice Boltzmann Method' by Kruger et al. [34].

3.2 Boltzmann equation

The Boltzmann equation is derived from Newtonian mechanics, which assumes quantum mechanical effects are negligible. Within a control volume μ, there exists N discrete particles that move and collide randomly with nearby particles in the phase space. Here we consider the position $\boldsymbol{r}$ and velocity v of the particles as coordinates. The particle density or distribution function as it is more commonly known, is an important property in the kinetic theory of gases. The distribution function is a function of the position of the particles denoted as $\boldsymbol{r}$, the velocity of the particle denoted as $\boldsymbol{v}$ and the time t, such that

$$N = \int f(\boldsymbol{r}, \boldsymbol{v}, t) d\mu \tag{3.1}$$

where $f(\boldsymbol{r}, \boldsymbol{v}, t)$ is the distribution function and N is the total amount of particles inside a volume μ. This distribution is represented by the Maxwell-Boltzmann distribution function shown in equation 3.2,

$$f(\boldsymbol{v}) = 4\pi \Big(\frac{m}{2\pi KT} \Big)^{\frac{3}{2}} v^2 e^{(\frac{mv^2}{2KT})}, \tag{3.2}$$

where m is the particle mass, k is the Boltzmann constant and T is the temperature.

On a microscopic scale, the distribution function varies as particles enter and exit the control volume. The movement of particles within the control volume can be divided into two distinct aspects: particle streaming and particle collision. Invoked during the streaming step, Liouville's theorem states that the particle density function is conserved along a particle's trajectory. This permits the use of the following equation to model the particle streaming terms:

$$f(\boldsymbol{r}, \boldsymbol{v}, t) = f(\boldsymbol{r} + \boldsymbol{v}\delta t, \boldsymbol{v} + \frac{F}{m}\delta t, t + \delta t) \tag{3.3}$$

where v is the particle velocity, F is the force, m is the mass of the particle and f is the particle density. Using the Taylor expansion, we find the movement of particles obeys the following mathematical equation,

$$\frac{\partial f}{\partial t} + v\nabla_x f + \frac{F}{m}\nabla_v f = 0. \tag{3.4}$$

We implement a collision term in equation 3.4, which is affected by thermodynamic properties like temperature and pressure. When its effects are included, we obtain the Boltzmann equation, which is often symbolically written as

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \nabla_x f + \frac{\boldsymbol{F}}{m} \cdot \nabla_v f = \left(\frac{\partial f}{\partial t}\right)_{coll}. \tag{3.5}$$

where the term on the right-hand side represents the effects of collisions. The expansion of the collision term depends on the phenomenological assumptions one introduces. An often-used expression is

$$\left(\frac{\partial f}{\partial t}\right)_{coll} = \iint \sigma(\Omega) \mid \boldsymbol{v} - \boldsymbol{v_1} \mid \cdot [f(\boldsymbol{v'}) \cdot f(\boldsymbol{v_1}') - f(\boldsymbol{v}) \cdot f(\boldsymbol{v_1})] d\Omega d\boldsymbol{v_1}. \tag{3.6}$$

where $\boldsymbol{v_1}$ is the velocity of the other particle, $\sigma(\Omega)$ is the cross-section of the collision, $\boldsymbol{v}$ and $\boldsymbol{v_1'}$ are the velocities after collision which is determined by $\boldsymbol{v}$, $\boldsymbol{v_1}$ and Ω according to momentum conservation. The Boltzmann equation is a non-linear integral-differential equation which is difficult to solve numerically. Therefore, the LBM approximates the collision term instead. There are numerous methods for approximating this term, but it must obey the conservation laws, conserving mass and momentum.

Most LBM models use an approximation called the BGK approximation [6] to approximate the collision term. The BGK approximation is a simple model to implement as it is a linear approximation and assumes that after a collision, the particle density f will relax towards an equilibrium particle density f_{eq}, with a rate proportional to the deviation of f from f_{eq}. The BGK approximation is mathematically represented as

$$\left(\frac{\partial f}{\partial t}\right)_{coll} = \frac{1}{\tau}(f_{eq} - f_{local}) \tag{3.7}$$

where τ is the relaxation time, f_{eq} represents the equilibrium particle density function. With equation 3.7, the Boltzmann equation becomes

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \nabla_x f + \frac{\boldsymbol{F}}{m} \cdot \nabla_v f = \frac{1}{\tau}(f_{eq} - f) \tag{3.8}$$

Note that the equation 3.8 has the underlying assumption is that the external forces acting on the particle are negligible. This simplifies equation 3.8 and removes the last term on the left-hand side. Therefore, the common form of the Boltzmann equation used in the LBM is

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \nabla_x f = \frac{1}{\tau}(f_{eq} - f). \tag{3.9}$$

One can derive the macroscopic equations using the Chapman-Enskog expansion and discover that $\frac{1}{\tau}$ has a unique relationship with viscosity. Suppose the collision rate $\frac{1}{\tau}$ is increased. In that case, the local particle distribution function should return to its original form much more quickly, which is a characteristic of high Reynolds number flows. This relationship is shown to be

$$\nu = c^2 \frac{\Delta t}{3}(\tau - \frac{1}{2}). \tag{3.10}$$

After deriving the simplified Boltzmann equation, calculating the macroscopic properties of the flow is straightforward. As f represents the particle density function, the density is the sum of all particle density functions. Similarly, momentum density is the sum of the product of the particle density functions and their velocities.

$$m \int f = \rho \tag{3.11}$$

$$m \int f\boldsymbol{v} = \rho\boldsymbol{v} \tag{3.12}$$

3.3 The Lattice Boltzmann Method

These ideas can be used to simulate fluid flow. As mentioned earlier, the main principle of the LBM is to simulate and understand the change in the distribution function f as it travels through a discretised solution domain. We discretise this form into a discrete-velocity distribution function $f_i(\boldsymbol{x}, t)$ where the density ρ and momentum density $\rho\boldsymbol{v}(\boldsymbol{x}, t)$ can be found using

$$\rho(\boldsymbol{x}, t) = \sum_i f_i(\boldsymbol{x}, t), \tag{3.13}$$

and,

$$\rho\boldsymbol{v}(\boldsymbol{x}, t) = \sum_i \boldsymbol{c}_i f_i(\boldsymbol{x}, t), \tag{3.14}$$

where $\boldsymbol{c}_i$ is the velocity set and $\boldsymbol{x}$ are the points where f_i is located in space. As the discrete distribution function f_i travels through the solution domain, it travels from one 'node' to another 'point'. Therefore, we can determine which 'connections' the distribution function can take from node to node in 2D and 3D space. The higher the number of 'connections', the higher the accuracy and the higher the computational demand. Therefore, the velocity set contains the velocity information of f_i as it travels along these connections.

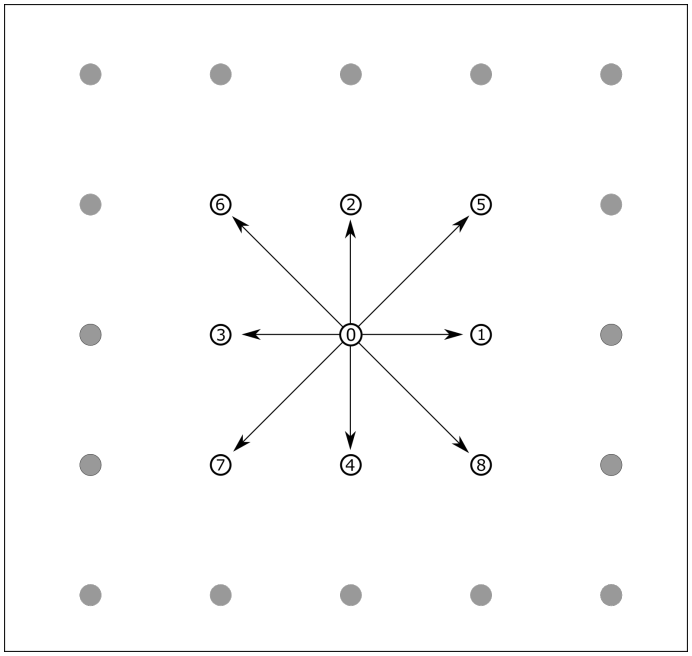

Fig. 3.1.: D2Q9 velocity set. Here the distribution functions f_i from i=1-9 travel to nodes connected to the centre node via the connection lines during the streaming step. f_0 remains in the centre node.

An example of this is the D2Q9 velocity set, where D2 represents 2D space and Q9 represents the number of connections from one node to another. Many velocity sets are available such as the D1Q3, D2Q7, D3Q15, D3Q19 and D3Q27 velocity sets. An example diagram showing the D2Q9 is shown in figure 3.1.

Each node in the lattice Boltzmann method stores its own probability distribution function and interacts with neighbouring nodes via connection lines. There are three processes that can be used to summarise the interaction between the nodes. The process of streaming, the collision process, and the computation of density and momentum density,

The streaming process is a discretised variant of 3.3 and dictates the movement of the f_i along the connection lines. The new macroscopic density and velocity are then computed in order to calculate the new probability distribution functions during the collision process. This effectively updates the node values, and the process is repeated until the simulation is terminated. Calculating the density and momentum density is necessary only when calculating the flow's velocity and pressure fields. Consequently, the calculation step is frequently skipped in order to reduce computational cost until it is required.

The discretized Boltzmann equation with BGK approximation is represented as

$$\frac{\partial f_i}{\partial t} + e_i \cdot \nabla f_i = \frac{1}{\tau}\left(f_i - f_i^{eq}\right) \tag{3.15}$$

where f_i represents the equilibrium probability distribution function at a node t is the time, and e_i represents the discrete velocity vector at the connection linkages. The expansion of the Maxwell-Boltzmann distribution can be used to obtain the local equilibrium distribution function.

$$f_i^{eq} = w_i \rho \left(1 + \frac{c_i u}{c^2} + \frac{(c_i u)^2}{2c^4} - \frac{u^2}{2c^2} \right) \tag{3.16}$$

where $\boldsymbol{u}$ is the local velocity vector, c is the lattice velocity and w_i is the weight of the connection link. The weight w_i is dependent on the velocity set used.

We explain these steps in further detail in subsection 3.3.1, and later we will describe how boundary conditions are applied in subsection 3.3.2.

3.3.1 Simulation process

The simulation algorithm involves a few simple steps

- Lattice Initialisation
- Collision step
- Streaming step
- Calculation step
- Loop back to collision step

This algorithm is applied to all of the inner nodes in the domain, with the exception of the nodes implementing certain boundary conditions, which are discussed in the subsection on boundary conditions. The subsequent paragraphs explain each step in greater detail.

Lattice Initialisation

The Lattice-Boltzmann method's lattice has distinctive characteristics that distinguish it from the computational grids of other CFD schemes. When the solution domain is discretized, a collection of nodes are positioned equidistantly from one another, and connection links are created between them. The choice of this structure or the velocity set determines the number of symmetries preserved in the LBM scheme. These configurations assume that a node's particles only flow to the

nodes to which it is directly connected. This indicates that the higher the number of linkages, the greater the degree of freedom and, therefore, the more accurate the solution. This has the disadvantage of requiring more computational resources.

Depending on the type of boundary condition used, additional boundary nodes are either placed along the area of interest or existing nodes are converted to boundary nodes after the domain is discretized. After the boundary nodes are assigned, the lattice is initialised by assigning density and momentum density to the domain's nodes. The initial probability distribution function is then computed using equation 3.16.

Collision step

This step involves determining the collision operator using the updated density and velocity values. This allows the node to calculate the new distribution function. This is accomplished by

$$f_i^*(\boldsymbol{x},t) = f_i(\boldsymbol{x},t) - \frac{\Delta t}{\tau}\Big(f_i(\boldsymbol{x},t) - f_i^{eq}(\boldsymbol{x},t)\Big), \tag{3.17}$$

where f_i^* represents the distribution function after collision and after f_i^{eq} is found using equation 3.16.

Streaming step

During the streaming step, the particle densities are shifted along the nodes via the connection links during the streaming step. The following equation provides a summary of this step.

$$f_i^*(x,t) = f_i(\boldsymbol{x} + \boldsymbol{c_i}\Delta t, t + \Delta t). \tag{3.18}$$

The new post-probability distribution function can then be used in the simulation algorithm for the subsequent collision.

Calculation step

Using the new probability distribution function, the macroscopic properties of velocity and pressure can be determined at this stage.

$$\rho = \sum_{i=0}^{Q-1} f_i \tag{3.19}$$

$$\boldsymbol{v} = \frac{1}{\rho} \sum_{i=0}^{Q-1} \boldsymbol{c_i} f_i, \tag{3.20}$$

where Q represents the number of linkages, ρ is the macroscopic density, and $\boldsymbol{c_i}$ is the lattice speed. Using the calculated values of density and velocity, the collision step can calculate the new equilibrium distribution function.

These three processes highlight the simple foundation behind the LBM.

3.3.2 LBM Boundary Conditions

Like all fluid problems, boundary conditions are required to define the fluid flow. Unlike the Navier-Stokes equations where the velocity or pressure can be set towards a certain value, the LBM works by manipulating the distribution function f_i. For example, the periodic boundary conditions are relatively easy as one redirects the distribution function leaving the solution domain to reenter the solution domain on the opposite side.

There exist numerous methods for implementing solid boundary conditions [19], but these can be categorised into two distinct families, the link-wise and wet-node boundary conditions. Imposing solid boundary conditions using the LBM can be quite tricky as the degrees of freedom associated with the system of mesoscopic variables is much higher compared to the corresponding macroscopic system. Intuitively, the boundary conditions should be defined using macroscopic moments such as density or momentum density, but determining the distribution function f_i using the macroscopic moments is not straightforward as many different mesoscopic approaches exist to obtain the same macroscopic boundary conditions.

These approaches are defined easier as the link-wise boundary conditions and wet-node boundary conditions after the methods used to impose the mesoscopic boundary conditions. The difference between these boundary conditions depends on how the distribution function f_i is manipulated, usually by redirecting the distribution functions. The location where this redirection happens is either between the nodes, which is the family that the link-wise boundary conditions fall under or on a set of special nodes called wet nodes, which is the family under which the wet-node boundary conditions fall.

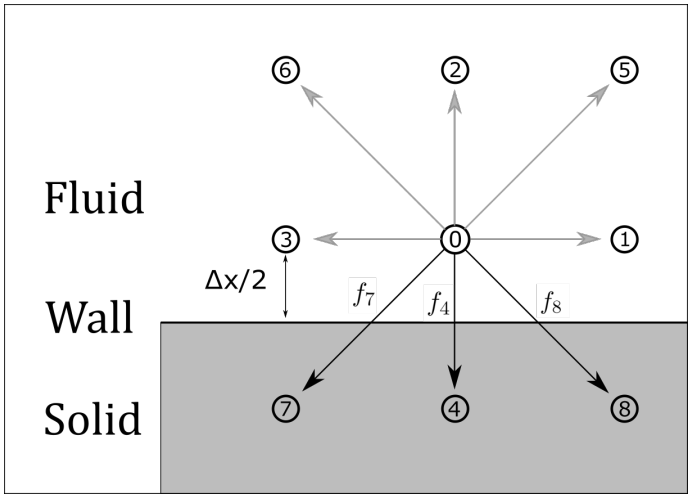

Fig. 3.2.: Bounce back boundary condition. Before the outbound flow crosses the wall during the streaming step, f_4, f_7 and f_8 are chosen to be redirected.

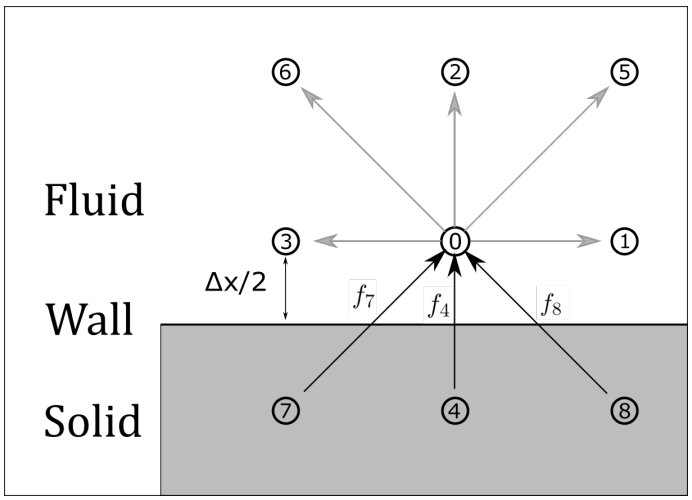

Fig. 3.3.: Bounce back boundary condition. Here, the bounce-back boundary condition is applied and the distribution functions f_4, f_7 and f_8 are redirected backwards.

One of the most popular boundary conditions to use is the bounce-back boundary condition which has been studied by various authors such as Zou and He [64], Ginzburg and Adler [18], Bouzidi et al [8] and d'Humières and Ginzburg [19] and imposes the Dirichlet boundary condition onto the fluid. Assuming the wall does not move, the bounce-back boundary condition determines the distribution functions travelling past a pre-defined wall and re-diverts the distribution function back along the connection line and replaces the outbound distribution function opposite it. This is seen in figures 3.2, 3.3 and 3.4.

The advantage of using the Bounce-back boundary condition is that it is a stable numerical scheme, which is especially close to the instability limit, as $\tau \to \Delta t/2$. Furthermore, this boundary condition does not alter the distribution functions and, therefore explicitly obeys mass conservation.

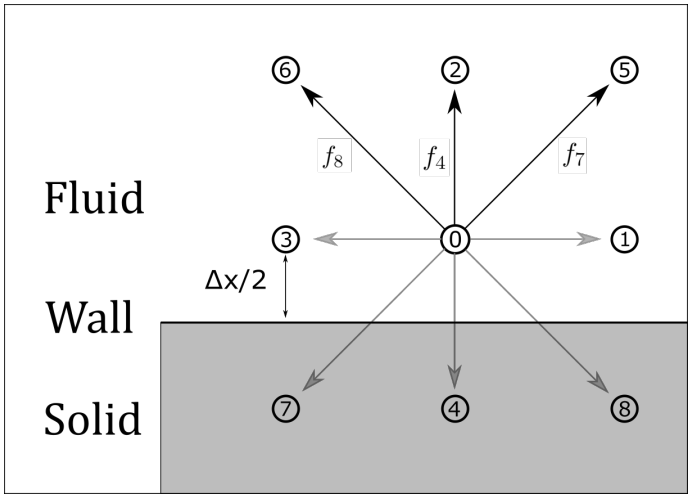

Fig. 3.4.: Bounce back boundary condition. Here, after the bounce-back boundary condition is applied, the distribution functions f_4, f_7 and f_8 replace the former distribution functions f_2, f_5 and f_6 respectively as the new distribution functions after the streaming step.

However, this boundary condition is dependent on having a regular square grid. To closely model a curve, we must approximate curves using 'stair-case' shapes, which has the indirect effect of reducing the order of accuracy from second-order accurate to first-order accurate. Finally, the location of the no-slip boundary condition is viscosity-dependent if the model is used with the BGK collision model. This causes issues as it implies that the hydrodynamic solution will differ even if the governing parameters, such as the Reynolds numbers, are fixed [15].

3.3.3 Computational parallelism

The LBM is particularly enticing to use as it does not need to solve the Poisson equation, unlike Navier-Stokes solvers. The Poisson equation is a significant computational restriction imposed on the incompressible Navier-Stokes equations. Typically, the equation of state can be applied to compressible Navier-Stokes equations to obtain the pressure, as it requires the temperature parameter from the energy equation. However, the incompressible Navier-Stokes equation lacks an energy equation, so pressure must be calculated differently. The alternative method uses the Poisson equation, which is derived by finding the divergence of the Navier-Stokes equation.

$$\nabla^2 P = -\rho \frac{\partial^2}{\partial x_i \partial x_j} V_i V_j \tag{3.21}$$

For conventional CFD methods, the computation of the Poisson equation dictates that the entire velocity field must be calculated prior to calculating the pressure at a single point. As velocity and pressure are intrinsically coupled, this constrains the parallelisation of conventional CFD methods. In contrast, the LBM lacks this limitation as the algorithm is 'local' meaning that each node processes data unique to that node and does not rely on information from other nodes. This allows multiple processors to independently process each node without requiring extensive coordination. Without concern for external variables, the LBM algorithm is able to allocate processors to different nodes independently, allowing it to utilise the computational power of many processors to simulate fluid flow exceptionally well.

3.4 OpenLB

As alluded to in the introduction, we choose to simulate these simulations using the Lattice Boltzmann method (LBM). Using the LBM to generate the dataset has some advantages for us :

- The simulations are almost embarrassingly parallel[1], allowing them to run very rapidly on an HPC [53] [2]. The vast majority of the computation is local in space, allowing the code to be easily localised into individual cores [35]. This complements the trend in computational power whereby the hardware used in HPCs has seen more use of multicore and manycore processors in recent years[17].

- The LBM does not involve the Poisson equation as the pressure and velocity can be recovered independently. This improves the computational efficiency even more as the equations are local rather than non-local[50].

- The LBM is well suited for simulating fluid flow in complex geometries[50].

- Automating these simulations is relatively straightforward. The partial differential equation can be easily discretised and implemented in each node. Additionally, it is simple to implement the boundary condition, and the entire code can be automated using a Python script.

We use the OpenLB C++ library to build the code of the simulations. OpenLB solves the Lattice Boltzmann equation (LBE) as seen in equation 3.15. OpenLB is an open-

[1]Although all the calculations are done locally, communication between neighbouring nodes are still required for each timestep.

source library that was first published by Mathias Krause at the Karlsruhe Institute of Technology and published under the GNU GPLv2 license.

Using OpenLB is advantageous in several ways,

- OpenLB has MPI and OpenMP built-in to parallelise the simulations on high-performance computers and has checkpointing features to allow resumable program executions.
- Implementing the LBM on OpenLB is intuitive due to the interfaces and templates provided, which helps makes the code modular.
- OpenLB has been tested and validated over several publications [33].
- Detailed documentation is widely available.

Detailed documentation is the main reason why this LBM software was chosen. The user guide written by Kummerländer et al. [36] is detailed. OpenLB also provides various velocity sets for 2D or 3D simulations such as D2Q9 or D3Q19 and a wide range of LBM collision models such as the BGK or the multiple-relaxation-time (MRT). OpenLB automatically generates the mesh once the domain is initialised and implements a wide range of boundary conditions on the domain. The OpenLB library allows complex geometry creation through code or a computer-aided design (CAD) file. We automate this process for the dataset generation stage by using OpenLB's geometry creation methods. We output these results in the VTK file format which is used by Paraview to display spatial and temporal data.

3.5 Summary

To summarise, the LBM is a simple and powerful pseudocompressible CFD scheme that can be used to simulate fluid flow. The majority of the computation is local, allowing the method to be easily parallelisable and scalable. However, the LBM is memory intensive as it requires a vast amount of information to be stored temporarily for calculation purposes. Due to the scalability of these simulations, it is appealing to simulate turbulent fluid flow on HPCs using the LBM and therefore is the cause why we intend to generate the dataset using the LBM.

4 Reduced order modelling

> *Our life is frittered away by detail. Simplify, simplify, simplify.*
>
> — **Henry David Thoreau**
> (American naturalist, essayist, poet, and philosopher)

4.1 Introduction

The quantity of data presents one of the greatest challenges when analysing turbulent flows. To produce simulations with accurate high Reynolds number flows, a dense mesh of grids is required to capture the features of the fluid flow, leading to large datasets. To simplify and condense the data into a more practical form and reduce computational costs, reduced-order modelling is often used to simplify the dataset and obtain interesting insights. Broadly speaking, reduced order models achieve cost reduction by focusing on large scales of the coherent structures in fluid flow. In this thesis, we employ the Proper Orthogonal Decomposition (POD) to identify dominant modes, although it is worth noting that many other potential reduced order methods exist, such as Dynamic Mode Decomposition (DMD). The POD method is chosen due to it's simplicity.

4.2 Proper Orthogonal Decomposition

Proper orthogonal decomposition (POD) is a well-known technique for decomposing a set of data into the optimal basis in the least-squares sense. It was first introduced by J. Lumley [43] into the field of fluid dynamics to extract coherent structures in the fluid flow, and it has since become a popular technique for analysing turbulent fluid flow. Notably, the POD is also known as the Principal Component Analysis (PCA) and is a variation of the Karhunen-Loeve expansion. It is the most efficient method for decomposing a fluctuating signal into lower-dimensional approximations in the sense of least squares. The first few terms of the decomposition contain the majority of the flow's energy and can therefore be used to reconstruct a

representation which captures the majority of the energy of the fluid flow. Due to these benefits, it can be utilised as both a model order reduction (MOR) and data exploration method [59]. We will mathematically describe the POD below.

Let $\boldsymbol{u}(\boldsymbol{x},t)$ be the velocity of the fluid field where $\boldsymbol{x}$ and t are the spatial vector and the temporal vector, respectively. We can use the Reynolds decomposition $\boldsymbol{u}(\boldsymbol{x},t) = \overline{\boldsymbol{u}(\boldsymbol{x},t)} + \boldsymbol{u}'(\boldsymbol{x},t)$ to decompose the velocity into a mean velocity $\overline{\boldsymbol{u}(\boldsymbol{x},t)}$ and fluctuating velocity $\boldsymbol{u}'(\boldsymbol{x},t)$. We assume that the velocity fluctuation $\boldsymbol{u}'(\boldsymbol{x},t)$ can be decomposed into an infinite series containing the basis functions that are dependent on space and time. We call the basis functions dependent on space the POD modes, and the basis functions dependent on time, the time coefficients. This infinite series can be represented as,

$$\boldsymbol{u}'(\boldsymbol{x},t) = \sum_{n=1}^{\infty} a_n(t)\phi_n(\boldsymbol{x}) \tag{4.1}$$

where n represents the number of terms in the decomposition [63]. There exist many other similar decompositions 4.1, such as the Fourier series, where the basis functions are in terms of sine and cosine. Here, the POD method distinguishes itself by imposing the condition of orthogonality onto the basis functions such that

$$\int_{\boldsymbol{x}} \phi_i(\boldsymbol{x})\phi_j(\boldsymbol{x})dx = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}. \tag{4.2}$$

The orthogonality property is useful as each time coefficient $a_n(t)$ is only dependent on a corresponding POD mode $\phi_n(\boldsymbol{x})$. We aim to determine the nature of the corresponding POD mode $\phi_n(\boldsymbol{x})$. Let ψ be a function that we use to approximate ϕ such that

$$\max \frac{\langle |(\boldsymbol{u'},\psi)|)^2 \rangle}{(\psi,\psi)} = \frac{\langle |(\boldsymbol{u'},\phi)|^2 \rangle}{(\phi,\phi)}, \tag{4.3}$$

where $\langle . \rangle$ is the time-averaging operator and (a,b) is the inner product of the functions a and b defined as

$$(a,b) = \int ab \ dx. \tag{4.4}$$

We can take the Lagrangian multipliers approach [31] to determine ψ and minimise the difference between ϕ and ψ using the L2 norm. Using this approach, we find that the POD modes can be obtained by the eigendecomposition of the covariance of the fluctuating velocity $u'(x,t)$. The covariance R is calculated using,

$$R = \sum_{i=1}^{m} \boldsymbol{u'}(t_i)\boldsymbol{u'}^T(t_i) = U'U'^T \in \mathbb{R}^{n\times n}. \tag{4.5}$$

When the domain is discretised with n grid points, the eigenvalue and eigenvectors are determined by the mathematical relationship,

$$R\phi_j = \lambda_j \phi_j, \quad \phi_j \in \mathbb{R}^n, \quad \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n, \tag{4.6}$$

where the eigenvalues are sorted in descending order and the eigenvectors represent the POD modes. Finally, the time coefficients can be obtained by computing the dot product of the matrix $\boldsymbol{U}$ and the POD modes $\boldsymbol{\phi}$.

4.3 POD Example

Below is an example of the use of the POD method for the 2D flow past a cylinder at a Reynolds number of 100. Figures 4.1, 4.2 and 4.3 shows a snapshot of the streamwise and transverse velocity components and vorticity of the unsteady flow. POD analysis is then applied onto the velocity in the x and y direction respectively as shown in figures 4.6 and 4.7, and figures 4.11 and 4.12. The strengths of each POD mode are shown in table 4.1. Here the POD modes are normalised by the sum of the eigenvalues and multiplied by 100 to obtain the percentage.

POD Mode Strength %	Streamwise velocity (X)	Transverse velocity (Y)
Mode 1	94.85	32.44
Mode 2	2.29	30.63
Mode 3	0.89	19.15
Mode 4	0.83	3.75
Mode 5	0.42	3.68

Tab. 4.1.: POD mode strengths for the flow past a cylinder at Re=100. The first five streamwise POD modes contributes to 99% of the energy of the streamwise velocity component, whereas the first five transverse POD modes contributes to 89.65% of the transverse flow.

The POD modes show the location where the majority of the energy are as demonstrated by figures 4.6 to 4.12. As the first two POD modes contain the majority of the energy as shown in table 4.1, we can simplify the transient von Karman vortex street, into two simple still snapshots. However, the POD method has two significant downsides. The first is that decomposing the flow to a set of basis modes that are chosen due to how much energy each mode contains is not necessarily a viable method of analysing the flow. It is possible to have dynamical modes that have small amounts of energy and are still highly significant. In the process of producing the POD modes, we must use an averaging process to obtain the second order statistics which dictates only partial information is utilised. The second downside is that the POD modes do not contain any temporal information. The POD modes can only represent the spatial distribution of the energies of the fluid flow.

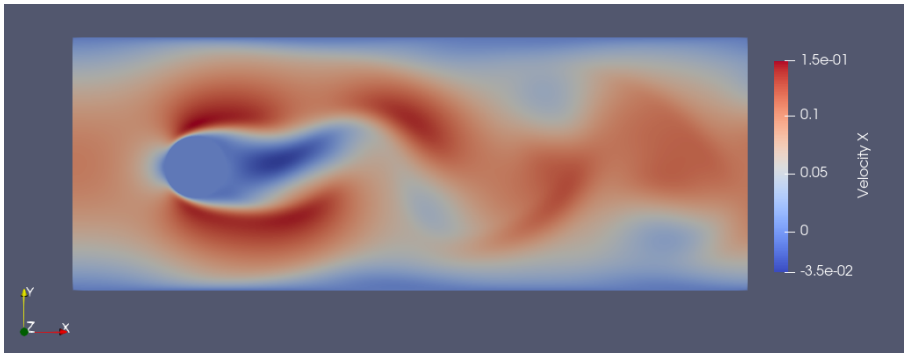

Fig. 4.1.: The instantaneous streamwise velocity in 2D flow past a cylinder at a Re=100.

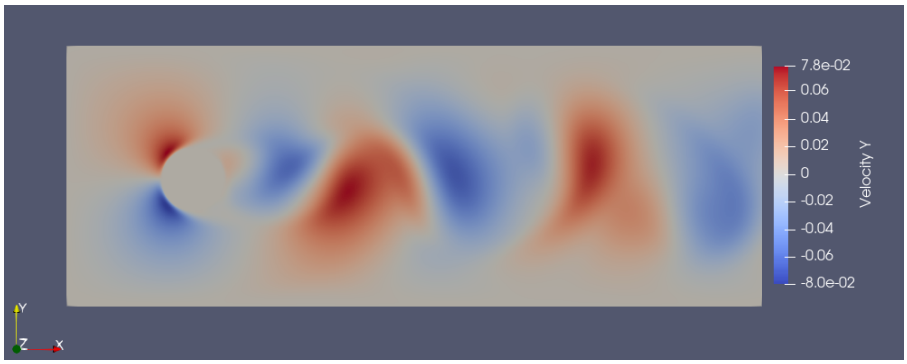

Fig. 4.2.: The instantaneous transverse velocity in 2D flow past a cylinder at a Re=100.

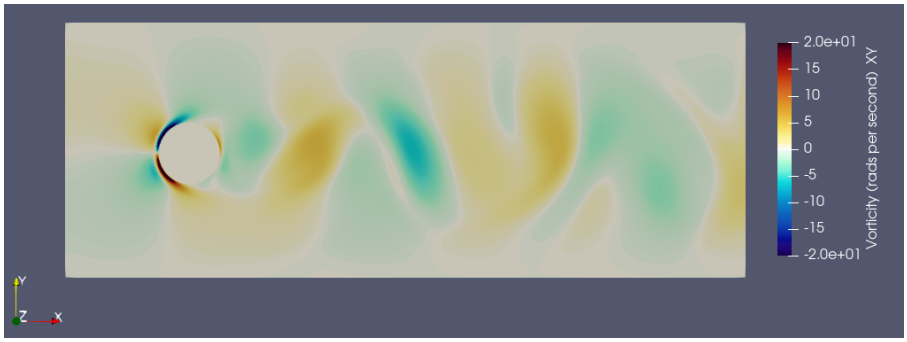

Fig. 4.3.: The instantaneous vorticity of the 2D flow past a cylinder at Re=100.

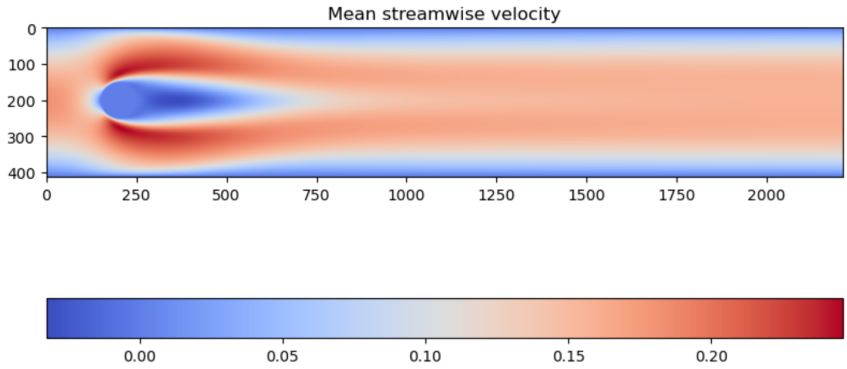

Fig. 4.4.: The averaged streamwise velocity in 2D flow past a cylinder at a Re=100.

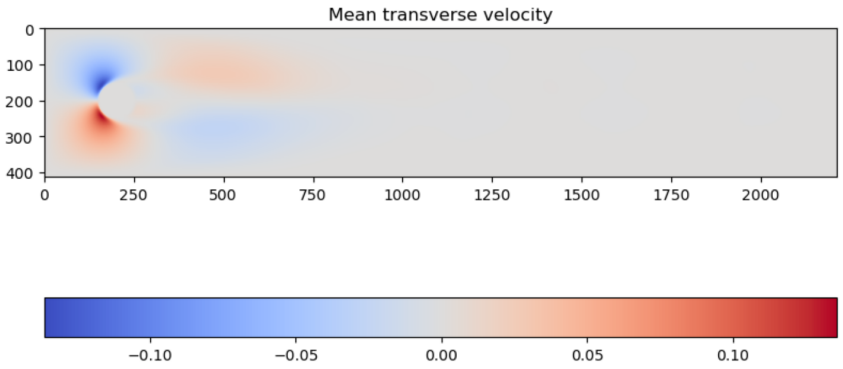

Fig. 4.5.: The averaged transverse velocity in 2D flow past a cylinder at a Re=100.

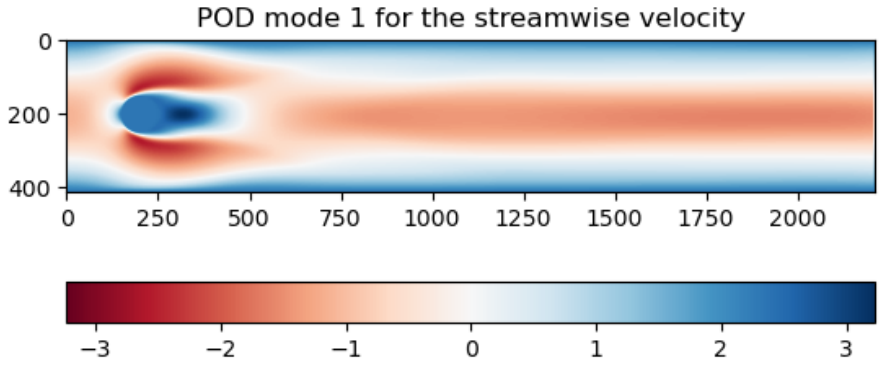

Fig. 4.6.: The first streamwise velocity POD mode. This dominant POD mode is identical to the averaged fluid flow, which is to be expected given that the majority of the fluid flow's energy should be within the simulation's centerline and not close to the wall.

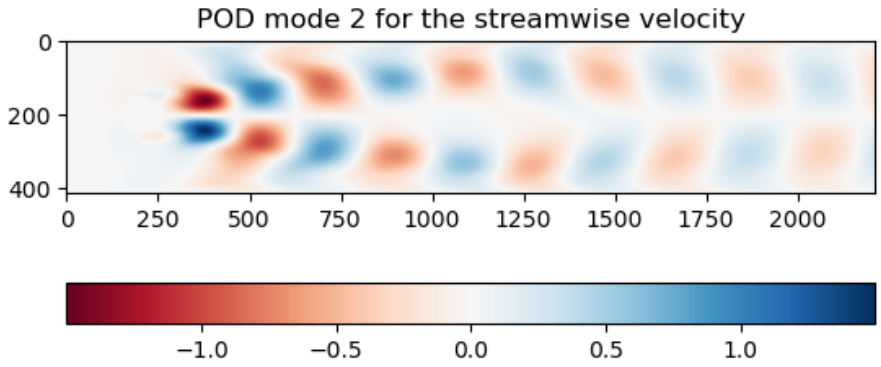

Fig. 4.7.: The second POD mode for the streamwise velocity. Note the alternating patterns in the wake behind the cylinder, which tells us about the spatial correlation of the fluctuations of the flow in the area. The region with the highest values seems to be directly behind the cylinder and progressively becomes weaker. The alternating patterns also seems to diverge before converging.

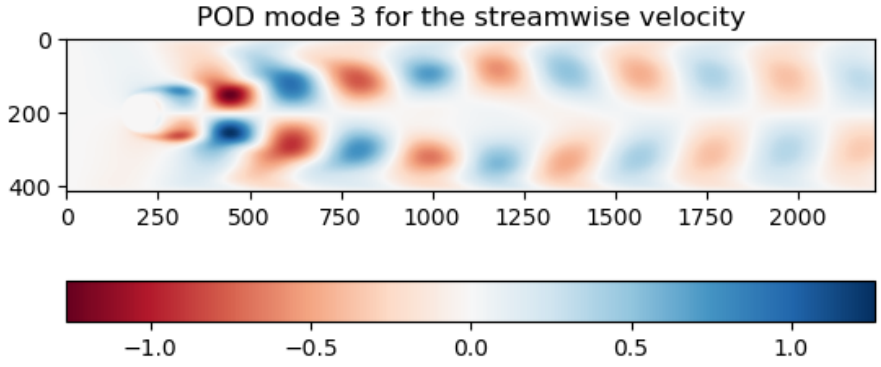

Fig. 4.8.: The third POD mode for the streamwise velocity. This POD mode appears to exhibit the same pattern and strength as the second POD mode. The only discrepancy is that the distribution appears to initialise with a half-wake before continuing with the regular patterns observed in the second streamwise POD mode.

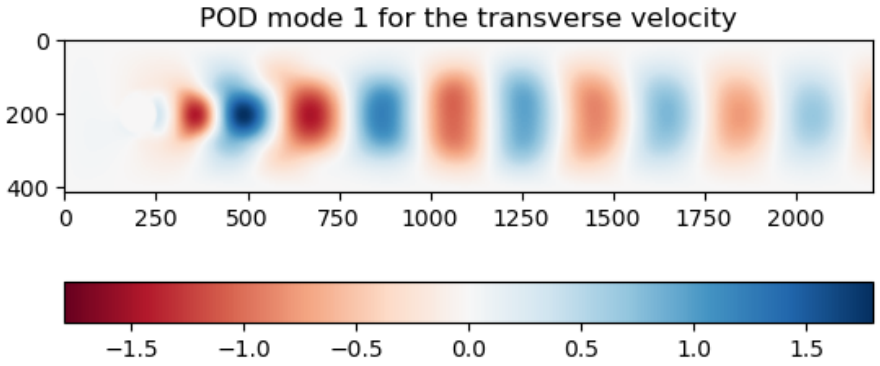

Fig. 4.9.: The first POD mode for the transverse velocity. The regions of high and low energies seems to correspond to the movement of the wake oscillating up and down along the centre line.

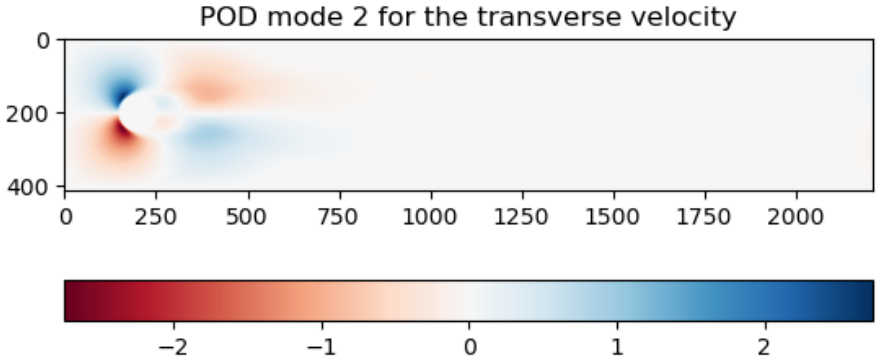

Fig. 4.10.: The second POD mode for the transverse velocity. This mode appears to depict the fluid flow being split by the cylinder in the simulation's centre, after which the fluid converges back to the centre line. This looks identical to the averaged transverse velocity.

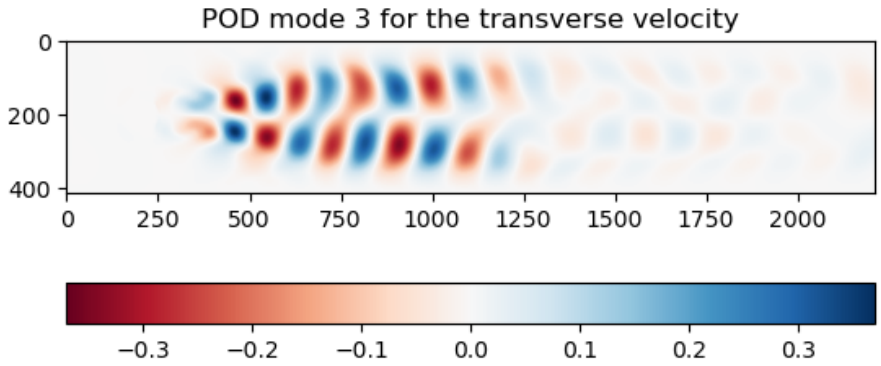

Fig. 4.11.: The third POD mode for the transverse velocity. Here the POD mode shows a region with a series of rapidly alternating areas after the flow is split by the obstacle. This region has an alternating pair of positive and negative energies, suggest that the fluid oscillates between each region. The region is initially concentrated behind the obstacle but eventually spreads out across the domain, weakening in the process. Finally, the alternating series of fluctuating energies weakens to an extent that it is hardly visible in the figure.

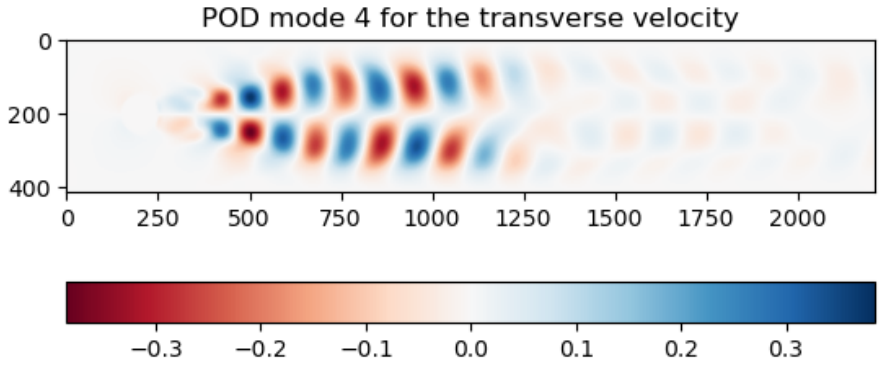

Fig. 4.12.: The fourth POD mode for the transverse velocity. This POD mode is similar to the third transverse velocity POD mode, but it appears to be slightly shifted towards the right. Like before, the region shows areas with a strong fluctuating energies that dissipates throughout the domain.

4.4 Summary

We show the derivation for the POD and show it's use in understanding the dynamics of complex flow. We aim to use a Convolutional Neural Network to predict these POD modes, and thereby understand the complex dynamics behind the fluid flow. As shown above, we can reconstruct the original flow using the time coefficients and POD modes.

5 Predicting POD modes using convolutional neural network

> *If we want computers to discover new knowledge, then we must give them the ability to truly learn for themselves.*
>
> **— Demis Hassabis**
> (CEO of DeepMind Technologies)

5.1 Introduction

In this section, we simulate turbulent flow using convolutional neural networks. We devise a method to predict the POD modes of the turbulent fluid flow using the geometric data of the obstacles, which could be used to understand the characteristics of the turbulent flow and reconstruct the turbulent fluid flow. This is similar to Guo et al. [22] work on predicting the steady flow around bluff obstacles, but we adjust this method to predict the POD mode such that it can be used to reconstruct turbulent flow.

As noted in section 4, reconstructing the original data requires both the POD modes and the time coefficients. However this work only aims to predict the POD modes, and therefore we use the time coefficients from the original dataset to test our approach.

This study serves as a proof-of-concept to determine if it is possible to use a CNN to interpret geometric information, and predict the modes of the proper orthogonal decomposition modes of turbulent flow. One can interpret this in another way, is it possible for a CNN to understand the boundary conditions imposed on the fluid domain, and predict how the flow will change with a new boundary condition.

In this section, we present our methodology for reconstructing turbulent flow. We first generate simulation data using CFD, and then calculate the first ten POD modes for all simulations to generate the dataset for the ML model. Then, we train the model using a CNN and analyse the POD modes predicted by the model. Finally,

the turbulent flow is reconstructed using the predicted POD modes and compared to the original data. For the sake of brevity, the figures used in this section are for the first two POD modes. The remaining eight other POD modes are located in the appendix, in section A.

5.2 Dataset Generation

5.2.1 Outline

This section outlines the method used to generate the dataset of POD modes that are used to train the model. We generate the simulation data using the Lattice Boltzmann method due to its computational efficiency and then process the simulation data to obtain the POD modes. We are interested in understanding how the geometry of obstacles affects the turbulent fluid flow so we will vary the geometric configuration of the arrays and keep all other parameters constant.

We call a group of obstacles in a specified location, an array of obstacles. There are almost an infinite number of configurations in that an array of obstacles can be arranged. Multiple factors are involved, such as the size, shape, number, and location of each obstacle, which can vary in an array of obstacles. As a result, due to the problem's complexity, we decide to reduce the number of possible geometric combinations to a reasonable level. We choose to develop a framework in which a five-by-five grid is placed in the centre of the simulation domain. Then, we generate obstacles and position them within the grid. There are 25 possible locations where each equally sized square obstacle is placed. For the case with adjacent obstacles, they are merged to form one larger obstacle.

An example is given in figures 5.1 to 5.4. This framework limits the number of possible combinations to 33,554,431[1] and keeps the size and shape of each obstacle constant. This grid approach can also produce complex geometric arrays. Referring back to figures 5.1 to 5.4, as the number of obstacles increases, the flow regime can be seen to steadily change from the flow past bluff bodies to the flow through a series of obstacles, to the flow around a rough body.

As it is not possible to determine how many simulations are needed to produce a broad dataset, we aim to simulate as many simulations as possible. As computational resources are limited, we simplify the simulations to reduce the computational demand, allowing more simulations to be conducted. As a result, we opt to use the Lattice Boltzmann method to produce the simulation data. The LBM is

[1]This value was calculated by $2^{25} - 1$. The case with no obstacle is not considered.

nearly embarrassingly parallel [34] which allows each simulation to be conducted quickly using multiple cores and more efficiently. As a result, the use of LBM synchronises well with GPUs or HPCs.

5.2.2 Simulation Set up

We run 2D LBM simulations on the Tier-2 High-Performance Computer and data science service 'Cirrus' (HPC). Cirrus is an HPE/SGI 8600 system and contains 280 standard compute nodes and 38 GPU compute nodes. Each standard computes node contains two 2.1 GHz, 18-core Intel Xeon (Broadwell) processors and 256GB of memory. Cirrus also provides GPU compute nodes where each GPU node contains two 2.4 GHz, 20-core Intel Xeon (Cascade Lake) processors and four NVIDIA Tesla V100-SXM2-16GB GPU accelerators. All nodes are connected via a single InfiniBand fabric. We conduct the simulations using the open source C++ LBM library OpenLB, where the details of the library are discussed in subsection 3.4. The simulations were parallelised using the OpenMPI protocol and compiled using the Intel C++ compiler, version 19.0.0.117.

As previously mentioned, we generate a series of obstacles within a five-by-five grid, where an obstacle is considered a single square within the five-by-five grid, and we call an array a group of obstacles. A series of examples are shown in figures 5.1, 5.2, 5.3 and 5.4 representing the generation of obstacles within a five-by-five grid, where the shaded areas represent the size and location of each obstacle in the array. For example, in figure 5.2, the total number of obstacles in this array is 8.

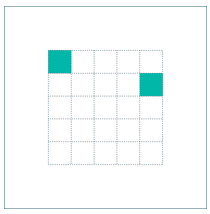

Fig. 5.1.: An array with two obstacles. This is similar to the flow past a set of bluff bodies.

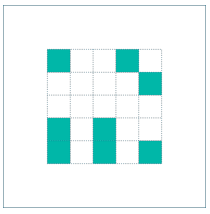

Fig. 5.2.: An array with 8 obstacles. This produces a fluid flow that is similar to the flow past a rough surface.

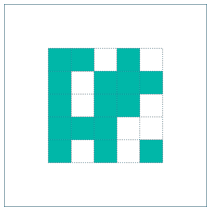

Fig. 5.3.: An array with 16 obstacles, is similar to the flow past a rough body.

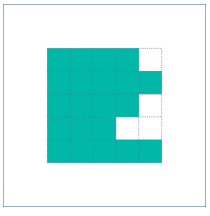

Fig. 5.4.: An array with 22 obstacles, is similar to the flow past a rough body as well.

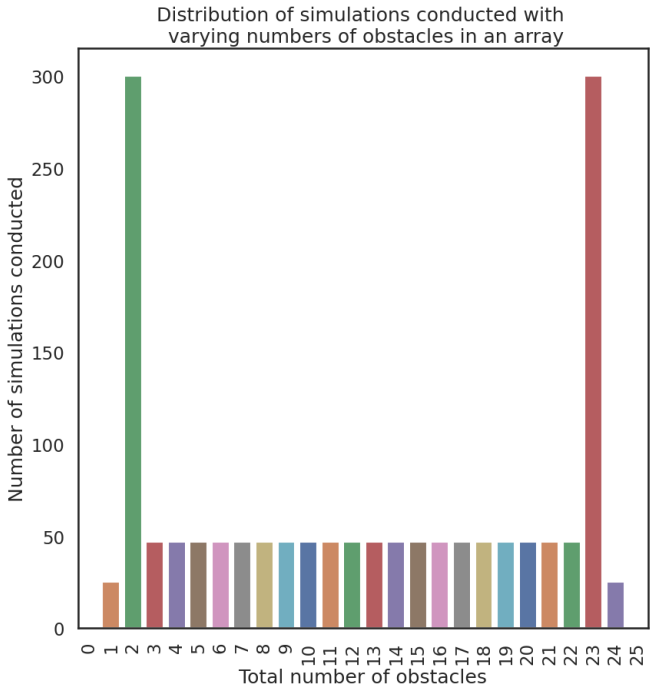

Fig. 5.5.: Graph of the distribution of the total number of obstacles in the dataset.

Two methods were used for generating the locations of these obstacles depending on the total number of obstacles contained in an array. The reason for this is that if we solely chose to generate the location of obstacles randomly then the set of randomly generated obstacles would have been unlikely to contain arrays with very low or very high numbers of obstacles. As mentioned earlier, the type of simulation data used will affect the accuracy of the model predictions. To emphasise the model learning the POD modes around these obstacles, we generate all possible geometric configurations for arrays with a total number of obstacles numbering from 1 to 2 and 23 to 24. Finally, we randomly generate the configurations for arrays with the total number of obstacles ranging from 3 to 22, where the number of configurations for each total number of obstacles is constant. The distribution of the number of arrays generated for the dataset is shown in figure 5.5. In total, 1590 arrays were generated, where 650 arrays were generated for arrays with a total number of 1, 2, 23 and 24 obstacles, and 940 for the remainder.

We choose our simulation parameters and boundary conditions based on reducing the computational resources required per simulation whilst maintaining the accuracy and precision needed for turbulent fluid flows. Table 5.1 has the simulation parameters used for these simulations.

We run simulations with a Reynolds number of 400 for a single obstacle and 2000 for a five-by-five domain. We show a typical simulation in figure 5.6, where the fluid

Reynolds number	400 for each obstacle, 2000 across the 5-by-5 grid
Total Time	106 pass-through time
Recorded data interval	21T-106 pass-through time
Domain Size	1Dx1D
Grid points	303x303
Obstacle size	1/40D
Five by Five grid size	1/8D
Collision operator	BGK
Velocity set	D2Q9
Mach Number	0.2
Relaxation time	0.502273

Tab. 5.1.: Simulation parameter table.

flows from the left boundary of the domain to the right. The simulations are run for 106 pass-through times, where one pass-through time is defined as the time it takes for the mean streamwise velocity to travel from the left to the right side of the domain. The turbulent fluid flow is measured between 21 and 106 pass-through times. The data for the first 21 pass-through times were not recorded because we were interested in observing developed turbulent flow; omitting these data reduces storage and computational requirements.

The total domain size is 1D by 1D, where 1D represents the side of the square solution domain, and the number of nodes used in the simulation is 303 by 303. The centre of the array is located in the centre of the domain, so 0.5D away from the entrance and 0.5D away from the height of the domain. The size of the array in comparison with the domain is $\frac{1}{8}$D, and the size of each obstacle is $\frac{1}{40}D$. We used the BGK collision model as the collision operator as it is computationally efficient and provides sufficient accuracy. The D2Q9 velocity set scheme was used, and the Mach number used in the simulation is 0.2 to reduce compressibility effects. The relaxation time used for these simulations was 0.502273. This value was obtained by using equation 5.1 [34],

$$\frac{Re}{Ma} = \frac{N}{\sqrt{3}(\tau - 0.5)} \tag{5.1}$$

where Ma is the mach number, Re is the Reynolds number, τ is the relaxation number and N is the number of nodes along the characteristic length.

The fluid is driven by a forcing term, and periodic boundary conditions are applied along the edge of the domain. Periodic boundary conditions were chosen as it improved the numerical stability of simulations, which allows the size of the fluid

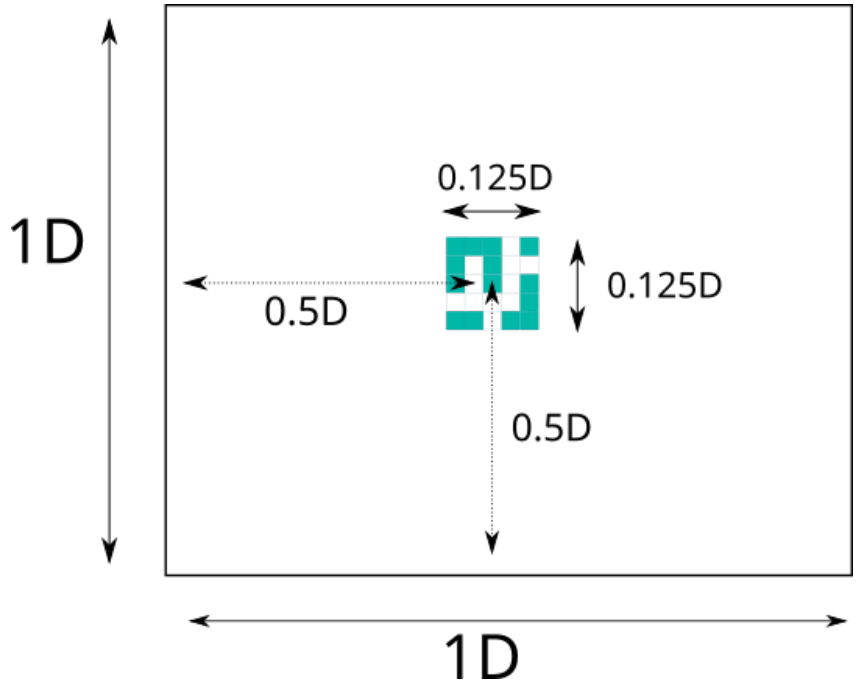

Fig. 5.6.: Diagram of the fluid domain. The edges around the fluid domain have periodic boundary conditions applied, and the entire domain has a forcing term applied to it. The obstacles are placed 0.5D away from the left and in between the top and bottom edges of the fluid domain.

domain to be reduced and decreases the computational resources needed for each simulation. Using periodic boundary conditions alters the nature of the type of simulations, rather than it being the flow past a single set of obstacles, it becomes similar to the flow past an infinite 2D plane of an array of obstacles. This type of flow is a good approximation of what we would expect to see in large cities and allows turbulent flows to developing which allows more advanced coherent structures to form during the simulation. We apply no-slip boundary conditions for the edge of the obstacles in the array, using OpenLB's implementation of a bounce-back boundary condition. No turbulence model was used for these simulations.

5.2.3 POD mode calculation

After simulating all the desired cases, we calculate the POD modes. The POD modes were calculated using SVD and were conducted on the Tier-2 HPC 'Cirrus'. We determine the fluctuating velocity values for each simulation's transverse and streamwise velocity and calculate the fluctuating values by finding the difference between the instantaneous velocity values from the mean. We use Welford's online pass to calculate the mean velocity as the method allows an iterative 'piecewise' calculation of the mean, whereby we can load a slice of the dataset into the RAM and compute an approximation of the mean. In comparison, the naïve method of calculating the mean demands the entire simulation data to be loaded onto the

RAM. Although Welford's online pass is not as fast compared to the traditional method, it is a stable algorithm that allows the processing of large datasets.

The mathematical equation to calculate the mean using Welford's online pass is,

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}, \tag{5.2}$$

where $\bar{x}_n$ is the running mean, $\bar{x}_{n-1}$ is the previous mean, x_n is the sample value and n is the running number of samples. This algorithm initialises $\bar{x_n} = 0$ and $n = 1$ then iterates to process the rest of the dataset. Welford's online pass is typically used to calculate variance and higher-order statistical moments, although we repurpose the algorithm to reduce the RAM demand required for this operation.

We use the SVD function from the Python library NumPy [23] was used to calculate the POD modes. This function is a Python wrapper for the LAPACK routine "DGESDD". Once calculated, we save the first 10 POD modes for both the streamwise and transverse velocity directions into an H5 file.

5.2.4 Data Analysis tools

We employ the Python libraries NumPy, Pandas, and PyVista [58] to handle the data analysis of this project. NumPy [23] is a Python library containing high-level mathematical functions that operate on python objects called NumPy Arrays. NumPy functions are typically wrappers for C or FORTRAN functions which allows fast processing of data whiles maintaining Python's ease of use. Pandas [60] [47] is a data analysis library used to handle data. Pandas use a python object known as a Pandas DataFrame, and it is built on top of NumPy. Pandas can be used to clean data, fill data, normalise, merge and join, visualise, inspect, load and save data.

We used the PyVista Python library to open and scan through the VTK files produced by the OpenLB simulations, and the desired data was then converted into a Pandas Dataframe and later to a NumPy array for data analysis and processing. We use the PyVista library to simplify the Python code as the code is more 'Pythonic' than ParaView's implementation of Python and VTI. An example of this simplicity is shown in **??**, where the code written below is used to open the VTI files. In PyVista, once a list containing the location of all files is identified, we use the 'read' function to open the data for each block and append it onto a list which we then turn into an array. In comparison, this is the code to open a VTI file using Paraview's implementation of Python, PVserver. Paraview's Python implementation uses many unique functions and objects that are only applicable to PVserver for the data analysis and this complicates the coding process.

An example of the first two POD modes are shown in figures 5.8 and 5.9 for the streamwise velocity for the streamwise flow through an array shown in figure 5.7. We leave the figures for the remaining eight POD modes are in section A.1. The patterns seen in the figures containing the POD modes show how unique the distribution of energies can be, and provides a view of what POD modes are expected to be seen throughout the 1590 simulations conducted. Generally what we see is that for the first two POD modes, the POD modes tend to be similar across all the simulations, but the remaining POD modes vary drastically.

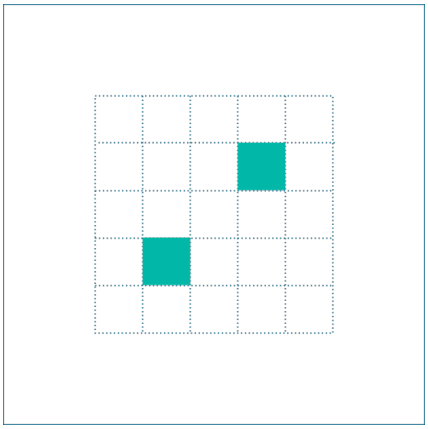

Fig. 5.7.: Diagram of the array of obstacles where the POD modes for example figures 5.8 to A.8.

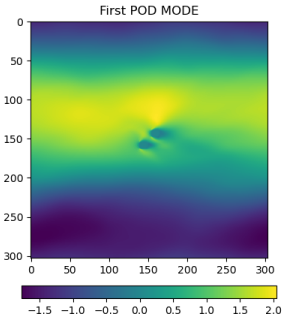

Fig. 5.8.: First POD mode of the simulated flow through an array of obstacles seen in figure 5.7. Here, we see a distinct horizontal separation between the region of positive energy and negative energy, indicating the flow is fluctuating between these areas.

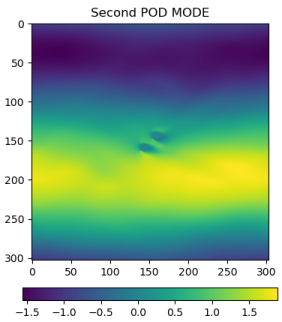

Fig. 5.9.: Second POD mode of the simulated flow through an array of obstacles seen in figure 5.7. Similar to the first POD mode as seen in figure 5.8, two distinct horizontal regions are seen but shifted downwards. As before, this suggests the streamwise velocity alternates

5.3 Convolutional neural network

5.3.1 Outline

In this subsection, we will discuss how we obtained our CNN model architecture, the CNN model, and how we trained the model. As we are attempting to understand how the geometry of the obstacles can alter the turbulent flow, it is reasonable to expect that a given array of obstacles would generate a unique set of POD modes. Therefore we designate the input of the model to be the geometry of the arrangement of the obstacles, and the output is the corresponding POD modes.

Overall, the model can be described in four steps. The first is the use of the signed distance function to generalise the geometry in subsection 5.3.4. The second is down-sampling the dataset such that the computational demand of the model can be reduced when training. This is discussed in subsection 5.3.5. The third step is training using the model architecture listed in subsection 5.3.3. Finally, we up-sample the output of the mode, using the methods discussed in subsection 5.3.5.

5.3.2 Model architecture search

Obtaining the model architecture for this project was a process of trial and error. As we were inspired by Guo et al. [22] on their work of predicting steady flow using the geometry, we initially replicated their model and modified the dimensions to fit the dataset. We then used the Keras hyperparameter tuner to see explore which hyperparameters were ideal for the model. This involved increasing the number of layers in the encoder and decoder, changing the activation functions, altering the convolutional kernel sizes, different loss functions and strides and the number of neurons in a layer. Mean squared error and mean absolute error was experimented and we found that the mean squared error converged faster and had a lower mean relative absolute error compared to the mean absolute error.

We found that increasing the number of layers in the encoder improves the performance up to the third layer. Any additional layers in the encoder grants negligible improvement in accuracy. A similar effect is also seen when increasing the number of layers in the decoder but up to the fourth layer. More than 200 models were trailed and the chosen model architecture was chosen based on performance and model size.

5.3.3 Model architecture

The project's convolutional autoencoder architecture is depicted in figure 5.10. The convolutional neural network is built using the TensorFlow machine learning Python Library. TensorFlow is a tool for large-scale machine learning that can function in a number of contexts. TensorFlow supports a wide variety of applications, with an emphasis on the training and inference of deep neural networks. We predict the POD modes using two convolutional layers, two fully connected layers, and three transpose convolutional layers. After each operation, the batch normalisation method is applied to the weights and biases of the operation's output. After initially emulating Guo et al.'s [22] model architecture to determine if it also worked for this application, this model architecture was determined. We discovered that adding an additional dense layer between the encoder and decoder and increasing the depth of each feature map reduced the model's error. We do not use any biases for the convolutional and transpose convolutional layers.

We use Keras Tuner's [51] implementation of Bayesian optimisation to determine the approximate hyperparameters that the model works with. This model is a separate encoder-separate decoder model, which predicts each individual POD mode. We find that there is minimal difference between a shared-encoder separate-decoder model and a separate-encoder separate-decoder model. We choose to use the separate-encoder separate-decoder model as training these models requires fewer amounts of RAM overall, although it increases the computational demand required to train these models.

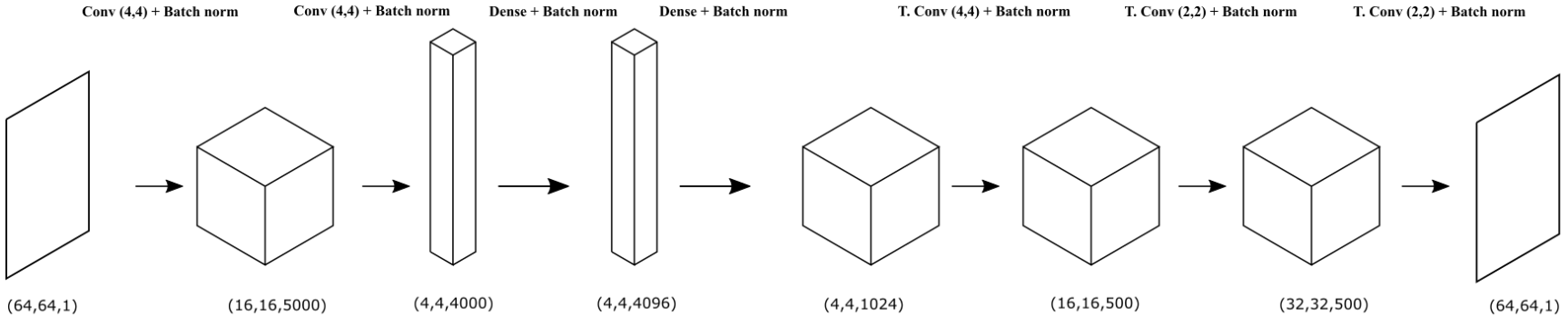

Fig. 5.10.: The model architecture used for this project. On the top of the figure, we denote the operation between each feature map, and on the bottom, the dimensions of the feature map are labelled. We abbreviate the operations used, in this case, 'Conv' means convolution, 'T.Conv' means transpose convolution and 'Batch norm' means batch normalisation. The numbers in brackets next to 'Conv' and 'T.Conv' represents the filter size used. As for the dimensions, the values represent the height, width and depth of the feature map, respectively.

5.3.4 Signed Distance Function

We use the signed distance function to represent the geometry we feed into the neural network. This model's input is an array representing the array's geometric information. The geometric data is given in binary, where 1's denotes the presence of the obstacle in the fluid domain and 0's represents the absence of the obstacle in the fluid domain. Although the neural network model should theoretically be able to form a link from the geometric data to the POD data, in practice, it is difficult for the model to train on input with zeroes. This is because the weights of the first layer are effectively reduced to zero, reducing the layer's effectiveness and the effectiveness of back-propagation as only the bias is being acted upon. See equation 2.4. A method to combat this is the signed distance function (SDF). The SDF is quite simple. It produces a field of values representing the shortest distance from the individual point to the surface of the nearest obstacle, as shown in equation 5.3. SDF defines the field outside of the obstacle as positive and inside as negative by convention. An example of this is shown in figures 5.11 and 5.12.

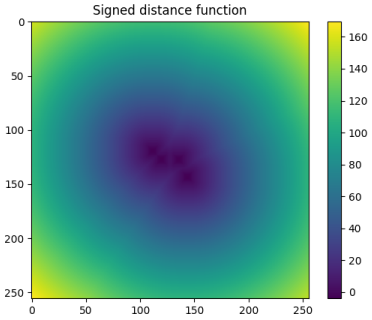

Fig. 5.11.: The SDF of an array with four obstacles.

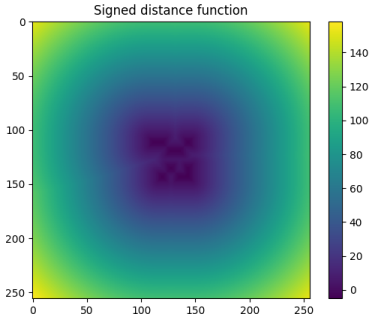

Fig. 5.12.: The SDF of an array with ten obstacles.

$$D(i,j) = min|(i,j) - (i',j')|sign(f(i,j)) \tag{5.3}$$

SDF is a method of neural implicit shape representation that aids in inferring the finer aspects of geometry. This method is low cost in terms of computational demand, eliminates noise, doesn't require any additional parameters to train, and gives a continuous representation of the geometry [57] [48]. To apply the signed distance function to the geometric data, we utilise the SciPy function "distance transform edt" as shown below.

```
1 def get_distance(f):
2     # Signed distance transform
3     dist_func = ndimage.distance_transform_edt
```

```
    x_1 = -dist_func(f)
    y_1 = dist_func(-(f-1))
    return np.where(f==0, y_1, x_1)
```

5.3.5 Pre-processing and Post-processing

We apply various pre-processing steps to the dataset before we use the dataset to train our model. Our SDF and POD datasets represent the input and outputs of our convolutional neural network, respectively. Depending on the nature of the datasets, it requires a different preprocessing method to enhance the ease of training for the convolutional neural network. We experimented with standardising and normalising the datasets and discovered that standardising the dataset improves the training process for these datasets.

As a reminder, standardisation is a method used to scale the dataset such that the mean is zero and the standard deviation is equal to 1, as shown in equation 5.4.

$$z = \frac{x - \mu}{\sigma} \tag{5.4}$$

where z are the scaled dataset values, x are the unscaled dataset values, μ is the mean and σ is the standard deviation of the unscaled dataset. Then, we apply downsampling to both datasets to reduce the dimensional size from $(303, 303)$ to $(64, 64)$. Reducing the dimension size reduces the computational demand and enables more complex and expansive model architectures. The model will then predict the downsampled POD mode prediction, which was upsampled to the original dimension for comparison later. The bilinear interpolation technique is utilised to downsample and upsample the datasets. This approach was taken as this reduces the size of the model which allowed training to happen at a faster rate. This does however lose coherent structures present in the flow due to the interpolation method.

5.3.6 Training

In this subsection, we describe the method used to train the model. First, we initialise the model parameters with a random normal distribution having a mean of zero and a standard deviation of 0.05. The random normal distribution was chosen after we determined that it performed more effectively than the He and Xavier distributions. All model training was conducted on the tier-2 HPC JADE2,

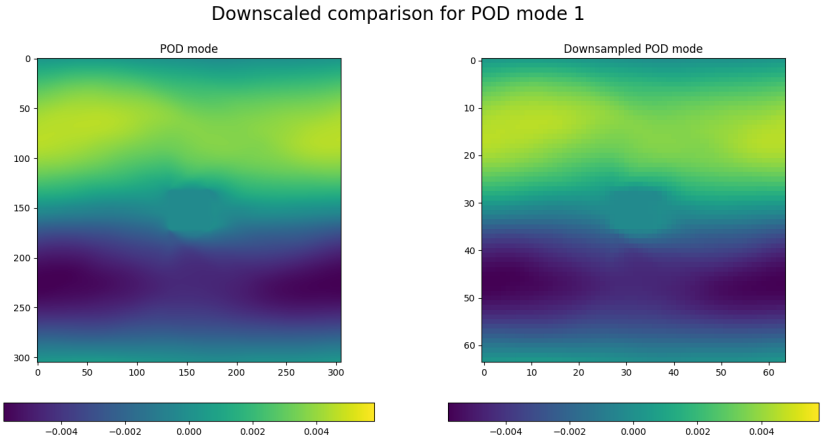

Fig. 5.13.: Two figures comparing the affect of downsampling from $(303, 303)$ to $(64, 64)$. This downsample reduces the number of pixels in the image by 95% but still retains the essential information of the image.

where the model is trained using eight Tesla V100s on NVIDIA's MAXQ Deep Learning System and connected via high-speed NVlink interconnect.

The generated dataset is post-processed and downsampled from a dimension of $(303, 303)$ to $(64, 64)$. As mentioned previously, the dimension reduction is due to the computational demand of training the model that can output larger predictions. The reduction contributes to a 95% decrease in pixels but still retains the essential information of the data as shown in figure 5.13.

We choose to use the MSE loss function as it appears to be the most stable loss function that allows the model to make reasonable predictions. We also attempted to use the MAE loss function, but it did not train the model well. And we used a batch size of 300 for training due to the RAM limitations of the GPUs used to train the model.

The dataset is randomly shuffled and split into training and validation sets to a ratio of 9:1. We then train the model with two different learning algorithms, Adam and SGD. We initially train the model for 500 epochs using the Adam loss optimiser and then train it for 5000 epochs using SGD. It appears that training the model with the Adam loss optimiser helps shift the model parameters to a state where SGD can then be utilised sensibly. We find via extensive experimentation that if the SGD loss optimizer were utilised exclusively, the training loss of the model would converge and fluctuate around a higher training loss. The initial use of the Adam loss optimiser appears to aid in training the model to a lower minima, and the use of the SGD loss optimiser afterwards further reduces it. We use an early stopping

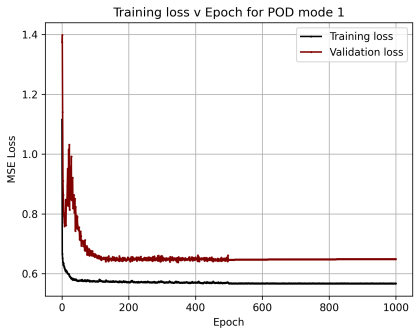

Fig. 5.14.: Epoch loss graph for POD mode 1

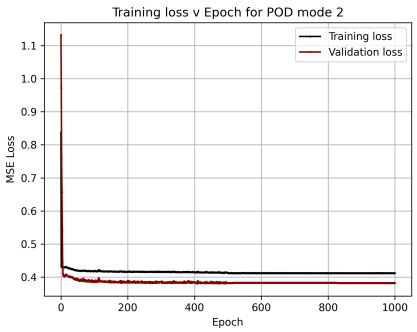

Fig. 5.15.: Epoch loss graph for POD mode 2

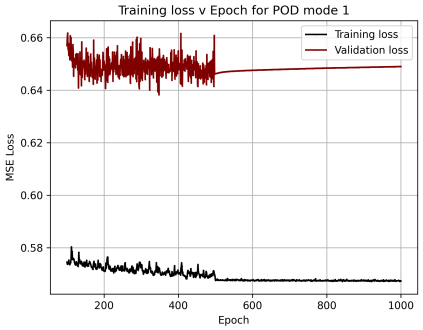

Fig. 5.16.: Cropped epoch loss graph for POD mode 1

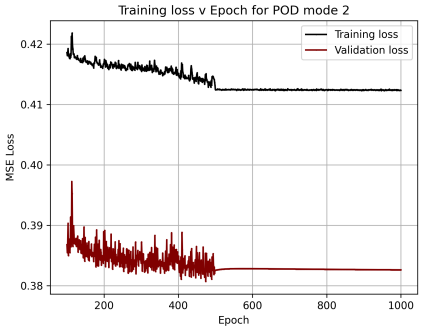

Fig. 5.17.: Cropped epoch loss graph for POD mode 2

method to halt the training process once the validation loss stops improving after 500 epochs to reduce computational demand.

We attach the epoch training graphs from figures 5.14 to A.16. These generally show that there is a big decrease in loss after a few epochs, which suggests that the Adam loss optimiser is highly effective in reducing loss. Due to the sudden decrease in training and validation loss in the first few epochs, we attach additional figures showing the training loss from epoch number 100 to the last epoch, in figures 5.16 to A.24.

Across all training graphs, the training loss decreases rapidly in the first few epochs as the model quickly adjusts away from the initialised parameters. For the first 500 epochs, we see a lot of fluctuation in the training loss and a general downward trend. After the first 500 epochs, the SGD loss optimiser is used, which smooths out the training and validation loss. In some cases, the SGD loss optimiser does not reduce the validation loss significantly, as the training process ends after 1000 epochs. However, in cases like POD modes 4 and 6, figures A.10 and A.12 respectively, the SGD loss optimiser does seem to do some work in reducing the training loss but not significantly.

5.4 Results

This section details the outcomes of our project. First, we evaluate the model's predictions and then evaluate the POD modes' accuracy in subsection 5.4.2. As the POD modes are a decomposition of the turbulent fluid flow, it is difficult to understand the significance of these errors intuitively when understanding how the errors in the POD modes affect the prediction of the velocity field. Therefore, in subsection 5.4.2, we reconstruct the turbulent fluid flow and analyse how the model's accuracy affects the results. We assume that the time coefficient data can be obtained with perfect accuracy and understand how the errors in the POD mode predictions alter the reconstructed turbulent flow.

5.4.1 Error evaluation

We use the mean relative absolute error (MRAE) to evaluate the performance of the model. This is defined as

$$\text{MRAE} = \frac{1}{T_{\text{max}} - T_{\text{min}}} \cdot \frac{1}{n \times m} \sum_{i=1}^{n} \sum_{j=1}^{m} |T_{ij} - P_{ij}|, \tag{5.5}$$

where T and P represent the original POD modes' true and predicted values and the ML-predicted POD modes. Finally, T_{max} and T_{min} are the highest and lowest elements in the matrix T, and n and m represents the domain size which is 303 and 303 respectively.

We select the MRAE as the error metric because it uniformly measures the overall error across the prediction image, and it is more intuitive to understand. The root mean relative squared error (RMRSE) is an alternative measure of error that could be used, but due to the squared nature of RMRSE, the error tends to represent the greatest element-wise difference between the true POD mode and the predicted POD mode.

We calculate the model's accuracy using the MRAE by comparing the model predictions to the downsampled training and validation datasets. We then report the MRAE of our model in figure 5.18 for each POD mode. Note that the MRAE values are computed using the model predictions before being upsampled, so the error values do not contain interpolation errors. Additionally, the standard deviation of the MRAE is calculated and shown in figure 5.19.

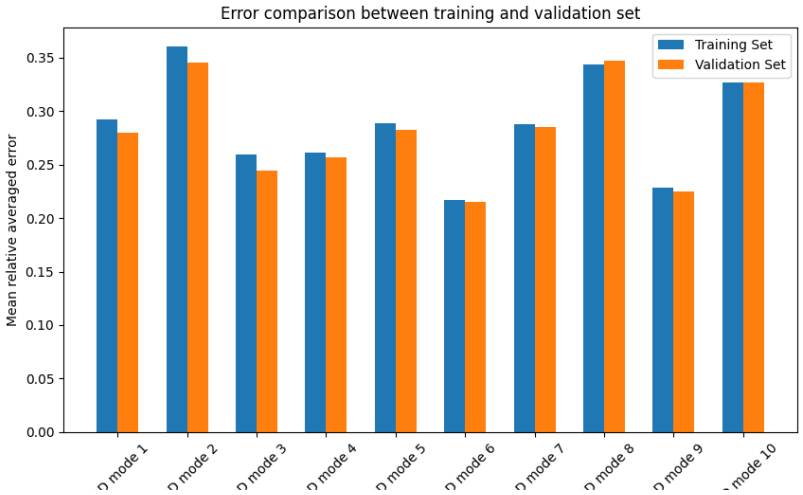

Fig. 5.18.: Mean relative average error comparison between training set and validation set.

Depending on the POD mode predicted, the model's average MRAE ranges from 0.21 to 0.36, and its standard deviation ranges from 0.045 to 0.15. We do not observe any overfitting in the model because the MRAE of the validation set is not consistently higher than that of the training set, which would indicate overfitting.

To further evaluate the model, we investigate the distribution of MRAE values across all POD modes and the training set and validation set to determine whether there are any discernible patterns in the accuracy of the predictions. The histogram figures show these distributions in figures 5.20 to A.32. Across all histograms, the error distribution tends to be positively skewed, with a long right-hand tail indicating a wide range of errors. Still, the majority of errors are typically relatively small.

We examine further more into the performance of the model by comparing the number of obstacles in the array to the averaged MRAE error as seen in figures 5.22 to A.40. This was done as we theorised that the dynamics of the flow through a few obstacles are drastically different from the flow around an array with a high number of obstacles. We find that the model does well at predicting the POD modes for arrays with higher obstacles but does poorly for arrays with smaller obstacles. To explain this, we examine the predictions made by the model case-by-case to understand the predictive power in subsection 5.4.2.

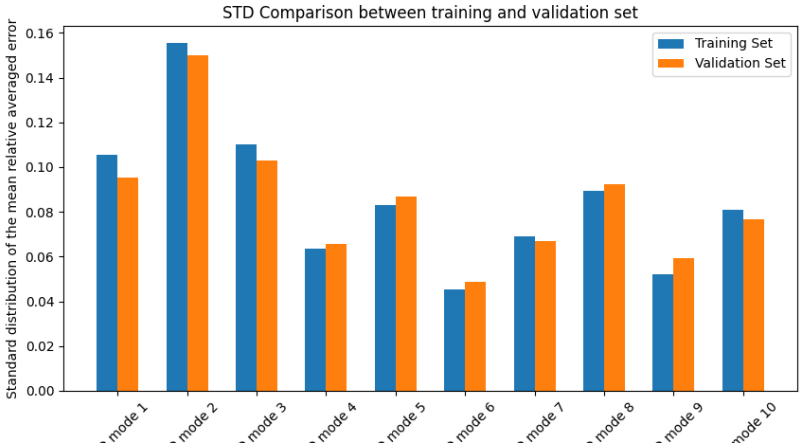

Fig. 5.19.: Standard deviation of MRAE comparison between the training set and validation set.

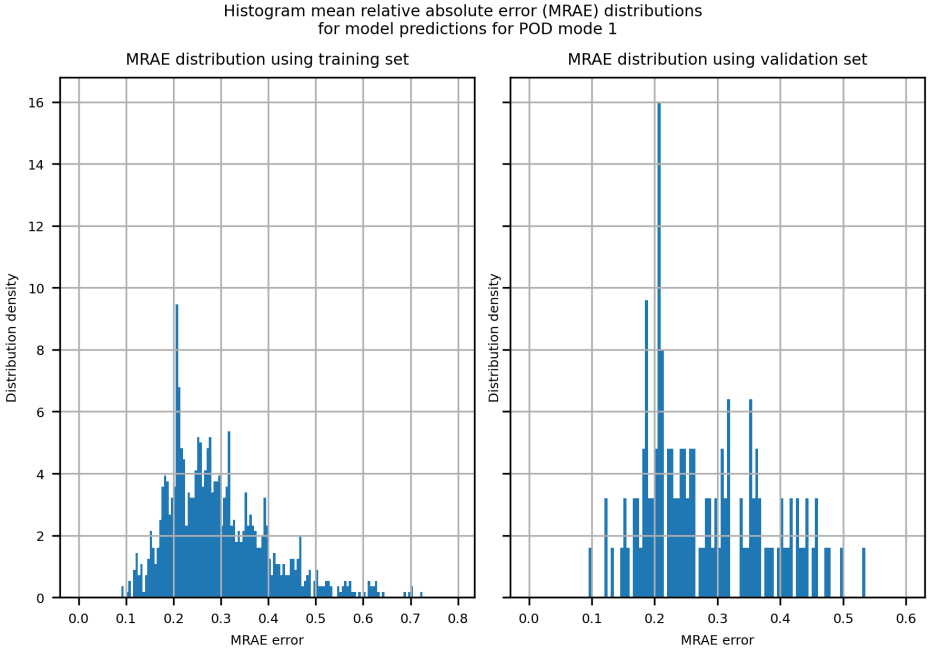

Fig. 5.20.: The normalised histogram showing the distribution of MRAE error for the first POD mode, separated between the training set and validation set.

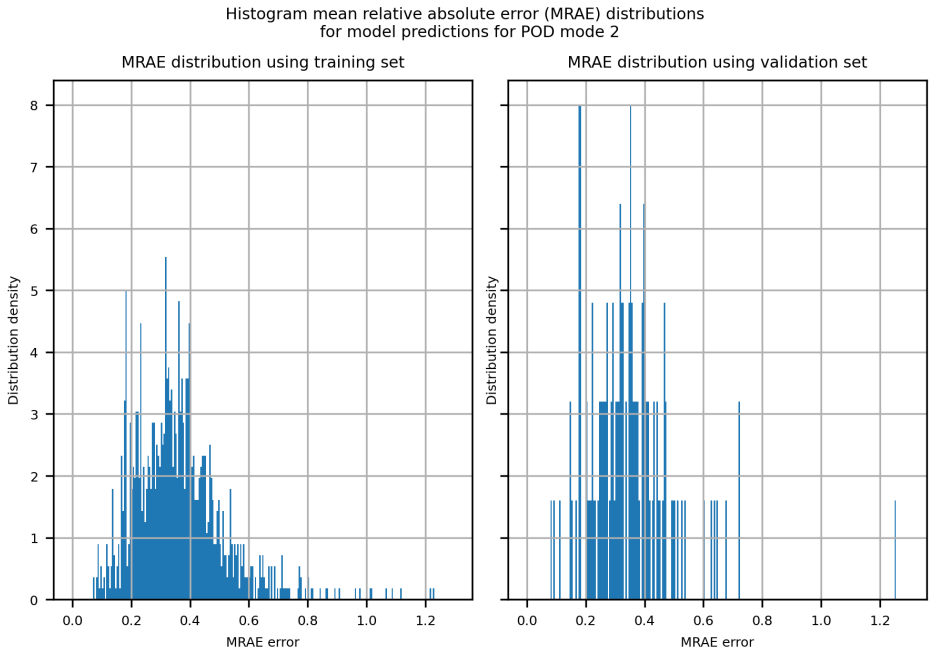

Fig. 5.21.: The normalised histogram showing the distribution of MRAE error for the second POD mode, separated between the training set and validation set.

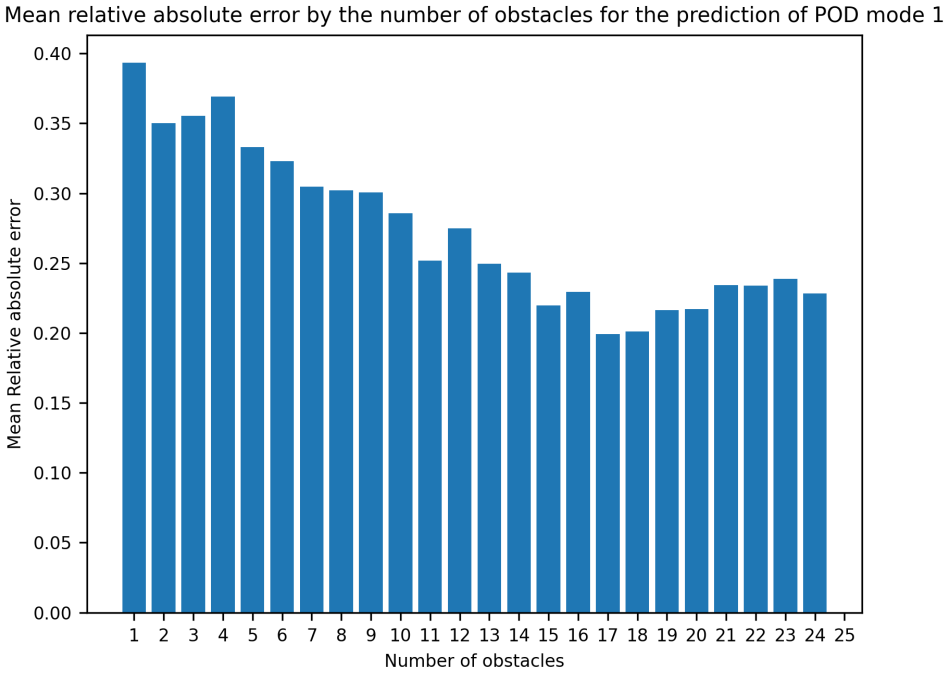

Fig. 5.22.: A histogram showing the error distribution against the number of obstacles in the array for the first POD mode.

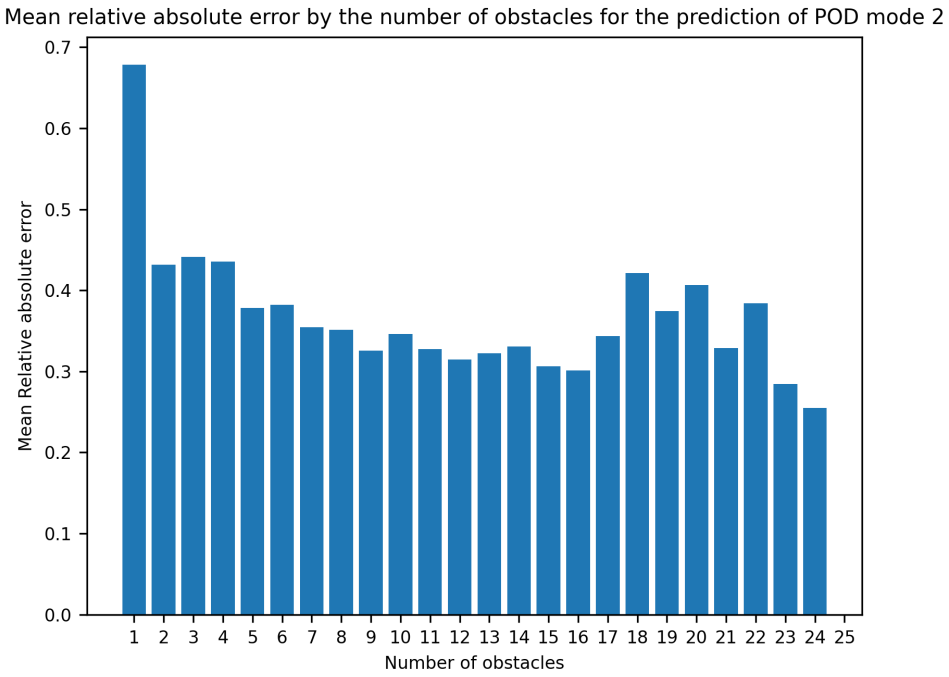

Fig. 5.23.: A histogram showing the error distribution against the number of obstacles in the array for the second POD mode.

5.4.2 Predictions and reconstruction

In this subsection, we present the model predictions. We first show the best POD mode predictions and then evaluate the model predictions for two different cases. For these two cases, one had the lowest error across all the POD modes, and the other had the highest error across all POD modes. We will then reconstruct the turbulent fluid flow using the predicted POD modes from the two cases and compare it to the simulated data to determine the model's efficacy.

We attach the best ML-predicted POD modes in figures 5.24 to 5.25 and figures A.41 to A.48 in the appendix. We find that the model can accurately predict POD modes 1, 2, 4, and 5 for certain simulations. This demonstrates that the machine-learning architecture is capable of predicting the correct POD modes. For the remaining POD modes, the predicted POD modes appear incapable of predicting a POD mode similar to the actual POD modes. This becomes a common theme through the predictions, that the model appears capable of reproducing images that resembles the POD modes but cannot reliably predict the POD modes.

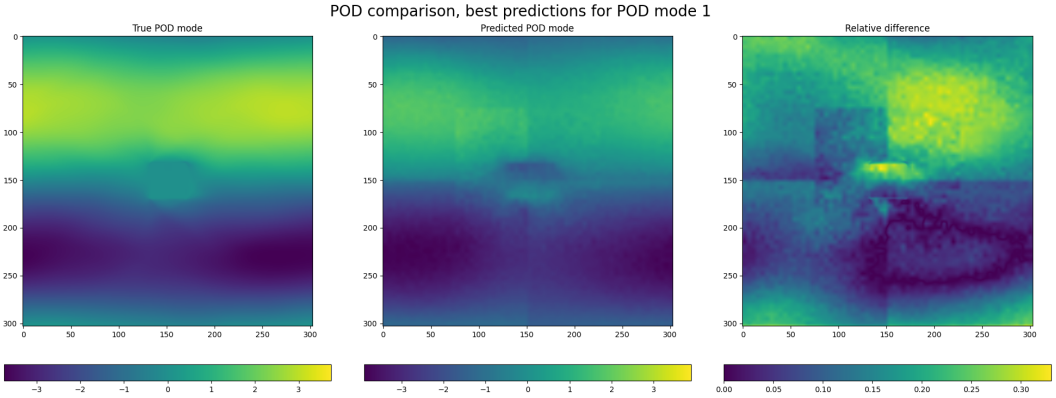

Fig. 5.24.: Best ML-predicted POD mode for POD mode 1.

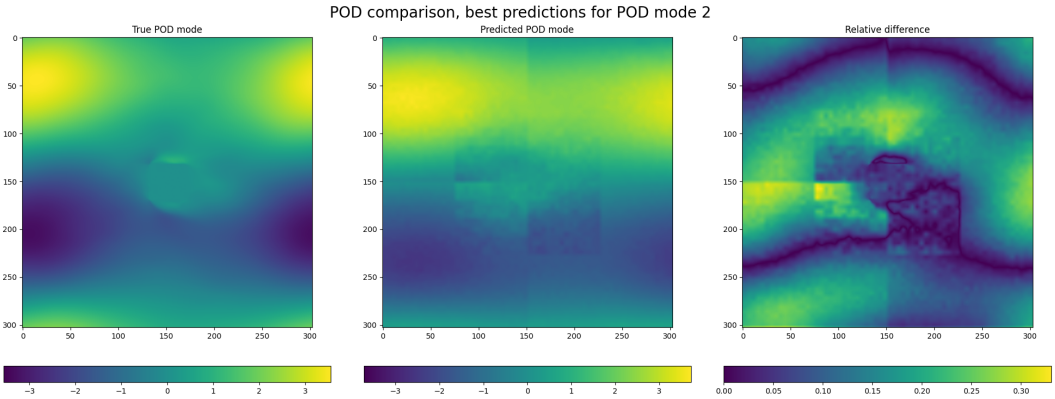

Fig. 5.25.: Best ML-predicted POD mode for POD mode 2.

We now present two different cases, array 332 and 33 after the index used to categorise the simulations. Figures 5.26 and 5.27 depict these two distinct cases. These

obstacle arrays were selected because they represent the case with the lowest and highest MRAE errors across all POD modes, respectively. In the provided figures, the dark shaded area indicates the presence of an obstacle, while the white area indicates the absence of a physical obstruction in the array.

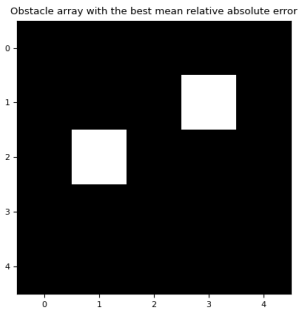

Fig. 5.26.: Simulation 332

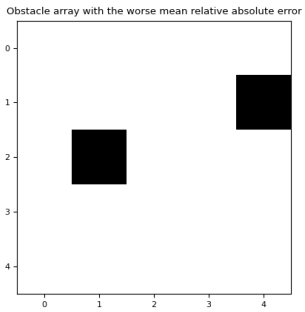

Fig. 5.27.: Simulation 33

Simulation 332 represents the array with the lowest total MRAE errors across all POD modes and simulation 33 is the array with the highest total MRAE errors across all POD modes. The black block represents where the obstacle resides, and white is the empty space.

For arrays 332 and 33, we attach figures comparing the predicted and true POD modes in figures 5.28 to 5.31 for the first two POD modes, and we leave figures A.49 to A.64 representing the last eight POD modes in the appendix. Note that we call the POD modes calculated from the simulation, the 'true POD mode' and for the ML predicted POD modes, we call those POD modes 'the predicted POD mode'.

In evaluating case 332, as depicted in figures 5.28 and 5.29, and A.49 to A.56 we observe that the model can predict, in general, the characteristics of the POD modes. In the first POD mode, shown in figure 5.28, the mode accurately predicts the two distinct positive and negative regions, as well as the beginning and ending edge of these regions.

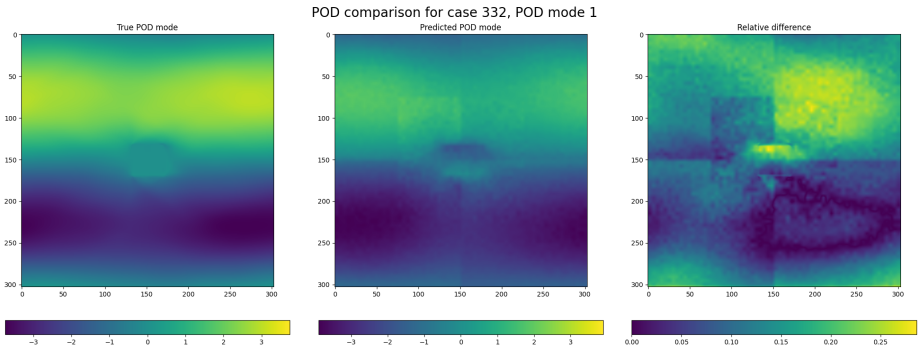

Fig. 5.28.: POD mode 1 comparison for array 332.

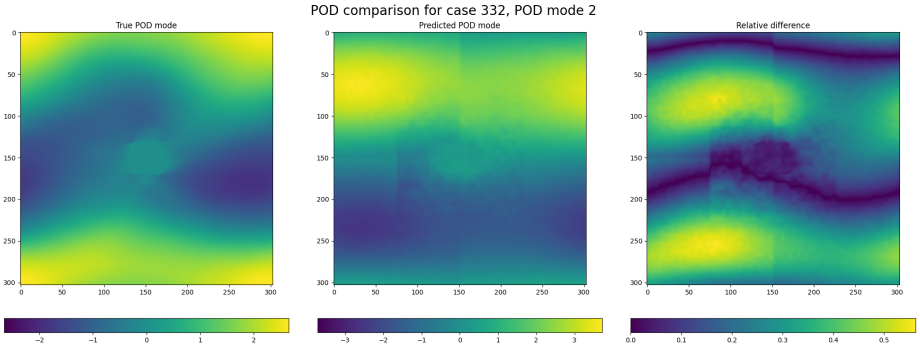

Fig. 5.29.: POD mode 2 comparison for array 332.

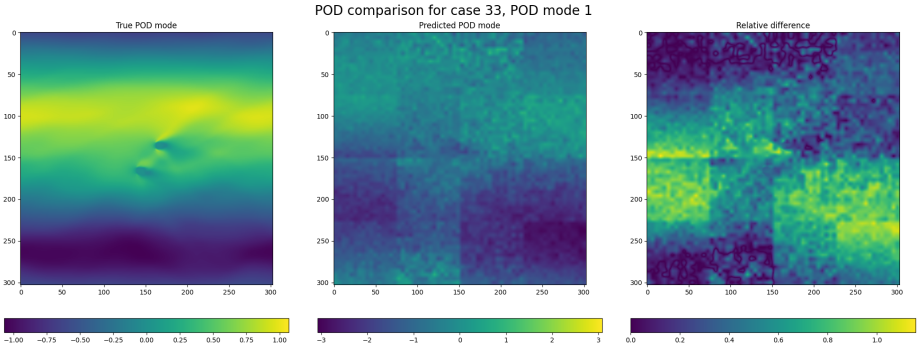

Fig. 5.30.: POD mode 1 comparison for array 33.

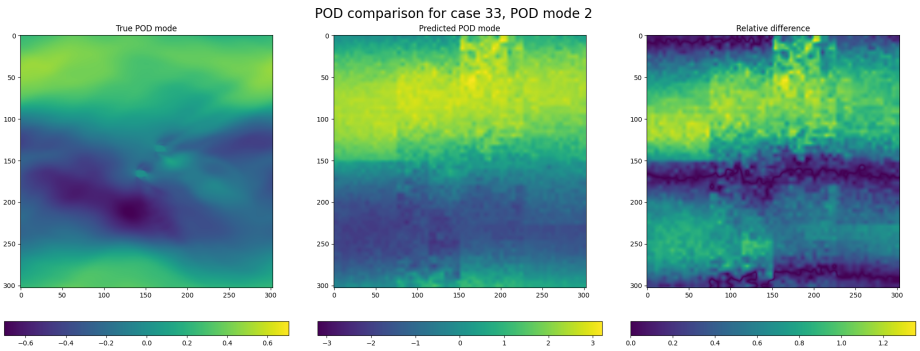

Fig. 5.31.: POD mode 2 comparison for array 33.

For the remaining POD modes, the model seems to predict characteristics that resemble those of the true POD mode, although the prediction is not similar. It seems that the model is capable of learning these patterns but are unable to use them in the correct context so far. To demonstrate this, for the second POD mode represented in figure 5.29, the model makes a prediction comparable to the first POD mode despite lacking access to the corresponding dataset. The prediction differs in the location of the regions of positive and negative energies, as we can see that the predicted POD modes would be similar to the true POD modes if the positive and negative features were translated slightly upwards. This suggests that for the majority of second POD modes, the model considers the second POD mode's dataset to be extremely similar to the first POD mode's dataset. This is likely the result of the force acting on the fluid. As the fluid is accelerated in a streamwise direction, its direction is predominantly linear, and we would expect to see this in the initial few POD modes. This is also seen in other types of flow, using the example from section 4, figure 4.6 is highly similar to the averaged flow seen in figure 4.4.

For the third, fourth, and fifth POD modes, as depicted in figures A.49, A.50, and A.51, the model predicts more interesting features. In the true POD modes for these three POD modes, there are four distinct regions containing two positive and two negative areas that alternate along the streamwise and transverse directions from positive to negative and from negative to positive. The model is unable to accurately predict the regions for the third POD mode, but is able to do so for the fourth and fifth POD modes. This demonstrates that the model can learn more complex features.

The model appears to struggle with certain POD modes, such as the sixth POD mode shown in figure A.52. Here, the model appears incapable of predicting similar characteristics to the true POD mode. Although this is likely due to the highly complex features displayed in the true POD mode, as this POD mode does not contain distinguishable symmetrical features unlike the previous POD modes. Moreover, it appears that regions with positive energies encompass the majority of the true POD mode, whereas the model appears to attempt to impose some sort of symmetry on the output.

As represented in figures A.53 to A.56, the model predictions appear to capture the essence of the remaining POD modes, despite not predicting the features in the correct locations. For instance, in the ninth POD mode depicted in figure A.55, the model appears to predict the 'spotty' nature of the true POD, and in the tenth POD mode shown in figure A.56, the model appears to capture the 'strippy' nature of the POD modes although the model is unable to predict where these features should be located.

We observe artefacts in the prediction in all POD modes, particularly near the obstacle's centre. This is most evident in A.54, where there is a notable artefact in the predicted POD mode's upper left corner. Notable mention is also made of the model's poor ability to predict the correct range of values. Considering the color-map in figures 5.28 through A.56, the model is only capable of accurately predicting the values for the first POD mode. For the remaining POD modes, the range of predicted POD mode values is either half or double the range of true POD mode values. This is an area where the model's predictions can be improved.

For array 33, the model performs poorly. For the first POD mode, as depicted in figure 5.30, the model appears to understand that there are two regions of interest, but it is unable to place these regions in the correct area or generate a similar range of values as the true POD mode. This trend also extends to the second POD mode, seen in figure 5.31. Here, the model seems to over-predict the correct range of values sixfold and is not able to produce similar features seen in the true POD mode.

Looking back at the second POD modes for array 33 and array 332, figures 5.31 and 5.29 respectively, the model seems to think that the ideal prediction is something similar to the first POD mode. As previously stated, it appears to have arrived at this prediction after training on a dataset of second POD modes without access to the dataset of first POD modes. This could suggest that the training dataset does not contain sufficient amount of samples for the model to train on as the 'averaged' image of these second POD modes could be similar to the first POD mode. Another possibility is that the model is too rigid to comprehend these distinctive characteristics and attempts to predict a "safer" option and implies that the model is underfitting for these predictions.

For the remaining POD modes of array 33, seen in figures A.57 to A.64, the model predictions more closely resemble noise than the actual POD modes and contain numerous artificial artefacts and, therefore, unsuited for the purpose of prediction.

To provide a more intuitive examination of the model's prediction performance, we reconstruct the turbulent fluid flow for the two cases listed above. In addition, we reconstruct the case with the smallest error while scaling the POD modes to match the actual POD mode. This is done to see visually how the errors in predicting the POD modes can alter the fluid flow if it is used to reconstruct the velocity fields.

We assume that the time coefficient components of the POD modes can be accurately reproduced, and we determine the time coefficients used to reconstruct the turbulent flow by utilising the simulation data and the true POD modes. Using

the same time coefficients and the ML-predicted POD modes, we reconstruct the turbulent fluid flow. This video of the reconstructed turbulent fluid flow is uploaded to YouTube for the reader's pleasure. The link for the case with the lowest MRAE is given in the footnotes[2]. Likewise, the link for the highest MRAE is given in the footnotes[3]. Finally, the link for the reconstructed flow using the case with the lowest MRAE and rescaled POD modes is given in the footnotes[4].

We also attach snapshots of the reconstructed flow field for offline readers in section A.7 in the appendix. Figures A.73 to A.80 show the snapshots of the reconstructed flow for array 33. Figures A.65 to A.72 show the snapshots for array 332, and figures A.81 to A.88 show the snapshots for array 332 with rescaled POD modes.

Each figure contains four images, the first of which depicts the actual turbulent fluid flow that the LBM simulation was used to simulate. The second is the turbulent fluid flow that was reconstructed by employing every true POD mode of the turbulent flow. The third image shows the reconstructed fluid flow using the first 10 POD modes, which we call as 'POD-10 flow' for brevity. Finally, the last image represents the reconstructed turbulent flow using the ML-predicted POD modes, which we call 'ML-predicted flow'. We show the first image as a reference, and the second image as a check to ensure the reconstruction method is correct. As we only take the first 10 POD modes, the third image is the ideal reconstructed fluid flow and the last image is the actual flow the model predicts would happen using the ML-predicted POD modes.

We observe some encouraging developments for the ML-predicted flow for array 332. When we look at the POD-10 flow, we can observe that there are no apparent coherent patterns and that the turbulent flow dynamics are random and chaotic. Since each of the 1215 smaller POD modes contributes to a smaller scale structure, it is not surprising that the POD-10 flow does not capture the small scale coherent structures observed in the simulation, but it does show the large scale features of the flow well. The ML-predicted flow aims to predict the POD-10 flow, and it appears that the large scale features can be reasonably predicted by the ML-predicted flow. The regions where we anticipate a positive streamwise velocity and a negative streamwise velocity are correctly highlighted by the ML-predicted flow, and the ML-predicted flow can transition in time with the POD-10 flow. However, it appears that the ML-predicted flow is unable to capture the correct range of velocities as seen in the POD-10 flow and it is consistently off by a factor of two.

[2] Link for the reconstructed turbulent flow using the ML-predicted POD modes for the array with lowest MRAE.
[3] Link for the reconstructed turbulent flow using the ML-predicted POD modes for the array with the highest MRAE.
[4] Link for the reconstructed turbulent flow using the rescaled ML-predicted POD modes for the array with lowest MRAE.

These encourage developments, however, are not observed for the ML-predicted flow in array 33. Up until the transition appears to stop occurring, the Ml-predicted flow appears to oscillate between positive and negative velocities. Additionally, we observe significant noise in the prediction as well as enduring artefacts within the domain with distinct artificial edges. This is also visible in array 332, though less prominently than in the ML-predicted flow displayed here. This simulation demonstrates that the ML-predicted flow is also inconsistently off by a factor of two and does not capture the correct range of velocities.

Finally we analyse the reconstructed flow for the rescaled POD modes for array 332. By rescaling the ML-predicted POD modes to match the actual POD modes, we try to address the scaling issue and then use the rescaled POD modes to reconstruct the turbulent flow. Rescaling the POD modes seem to have helped the ML-predicted flow match the correct range of velocities as that of the POD-10 flow. Additionally, this flow highlights some smaller features more clearly, improving on the previous ML-predicted flow.

This subsection will be concluded by examining the effectiveness of the ML-predicted POD modes using the reconstruction of the flow. We use the ML-predicted flow to understand how significant the prediction errors are, and we find that for the array with the smallest amount of error, it is able to reasonably be used to reconstruct the turbulent fluid flow, assuming the time coefficients can be perfectly obtained. Although in practise, predicting the time coefficients is another challenge.

5.4.3 Prediction discussion

This subsection discusses the CNN model's predictions. We note above that the histograms containing the MRAE for arrays with varying numbers of obstacles, given in figures 5.22 and 5.23 reveals that the model predicts the flow of arrays with fewer obstacles poorly.Examining the first POD mode, we observe that this trend begins with the first obstacle and the MRAE decreases gradually until the number of obstacles in an array reaches approximately 16, giving a void fraction of 64%. This accounts for approximately 983 samples out of a total of 1590, or 62% of the dataset. For the purpose of reconstructing turbulent flow, it is ideal to have a smaller amount of error for the first few modes, as these modes tend to represent the majority of the large-scale features of the flow; therefore, the inability to predict these POD modes suggests that the reconstruction of the flow using these modes will result in an inaccurate reconstruction. This implies that for the majority of cases, we expect to see a poor to average quality reconstruction.

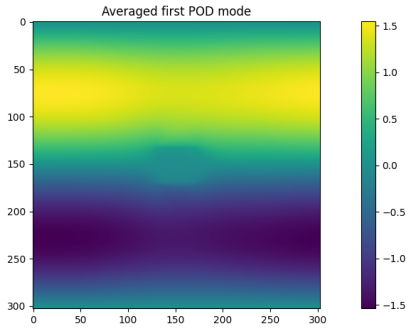

Fig. 5.32.: The averaged POD mode across the first POD mode.

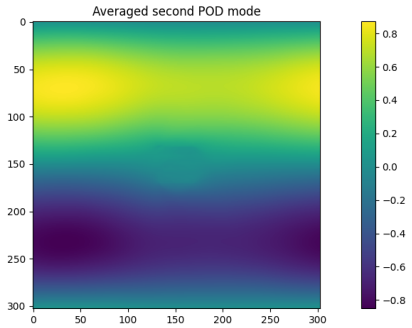

Fig. 5.33.: The averaged POD mode across the second POD mode.

For the remaining 38%, we intuitively visualise the flow around these arrays. We anticipate that as the number of obstacles increases, the flow will tend toward the flow around a single large bluff body. Consequently, this result may suggest that 38% of the remaining samples are comparable to this flow. To see if this is true, we average all the POD modes obtained from the simulations, which we show in figures 5.32 to 5.33 and figures A.89 to A.96 in section A.8 in the appendix. We find that the averaged POD modes are very similar to those seen in array 332, in figures 5.28 to 5.29 and figures A.49 to A.56, which suggests the model has a preference for predicting the flow for these arrays.

There are several potential causes to explain this. One reason is that the remaining 38% of simulations are the largest group with similar POD modes, while arrays with fewer obstacles have more diverse POD modes that the model cannot learn and predict. Another reason is that the geometric SDF data could provide too few useful data for the model to predict a wider range of POD modes, and instead is limited to the POD mode close to the average POD mode. Lastly, the dataset may be too small and unable to provide the model with sufficient data to understand the complex flows for arrays with fewer obstacles.

5.5 Future work with SSIM

Concerning the model accuracy, there appear to be two matters of interest in predicting fluid flow with the CNN model. The first is that the model has to predict the correct likeness of the POD modes, and the other obstacle is predicting the correct range of velocity values. As a potential future approach for this method, we propose using the structural similarity index measure (SSIM) to predict the likeness

of the POD modes. Proposed by Wang et al. [62], the SSIM can measure the likeness of two images. The SSIM is defined as

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{5.6}$$

where x and y represent the two reference and sample images, μ_x and μ_y represents the pixel sample mean of the image, σ^2 represent the variance of the images, and C_1 and C_2 are two variables to improve the numerical stability of the calculation. C_1 and C_2 are calculated by (k_1L^2) and (k_2L^2) where L is the dynamic range of the pixels and k_1 and k_2 are constants with the values of 0.01 and 0.03 respectively.

We present the figure 5.34 image, which illustrates the application of the SSIM measure and compares it to the MSE measure. The image in the upper left corner is the original, error-free data. The following five images are distorted such that the mean squared error (MSE) remains constant at 144. Simultaneously, the SSIM measure is used to analyse the similarity of the original data, and as the images progress, the SSIM measure decreases steadily. We observe that the images in the upper middle and upper right are subjectively very similar to the original data and therefore have a high SSIM score. Unlike the last two images, the distortions in the bottom three images are significant and alter the likeness of the image, which is indicated well by the decrease in the SSIM score.

Using the MSE metric to evaluate the similarity of the images would have yielded identical values, even though the similarity between the images is vastly different. There are two significant reasons why the SSIM measure is superior to the MSE for quantifying a superior measure of similarity. First, the majority of error metrics, such as the MSE, measure the difference in pixel values between a reference image and a sample image. However, the human visual perceptual system does not do this and can distinguish structural differences between a reference image and an example image. The SSIM measure defines these structural differences as the images' luminosity, contrast, and structural similarity, and these measures are defined to be the statistical moments of the reference and sample images.

The luminosity of the images is measured by comparing the mean of the images, defined as,

$$l(x, y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}. \tag{5.7}$$

The contrast is quantified as the standard deviation in the image, and we compare the standard deviation of the images using,

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}. \tag{5.8}$$

Finally, the structural similarity of the images are calculated by the covariance and variance of the images, represented as,

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}. \tag{5.9}$$

The SSIM measure combines these measures into one singular measure,

$$\text{SSIM} = l(x,y) \cdot c(x,y) \cdot s(x,y), \tag{5.10}$$

which is simplified into equation 5.6.

As discussed in subsection 5.4.2, the ML-predicted POD modes are occasionally unable to predict the general similarity of the true POD modes, and we hypothesise that the SSIM measure can be used to compensate for this shortcoming. We hope this serves as a convincing argument for using this metric to measure the similarity between POD modes compared to the MSE measure.

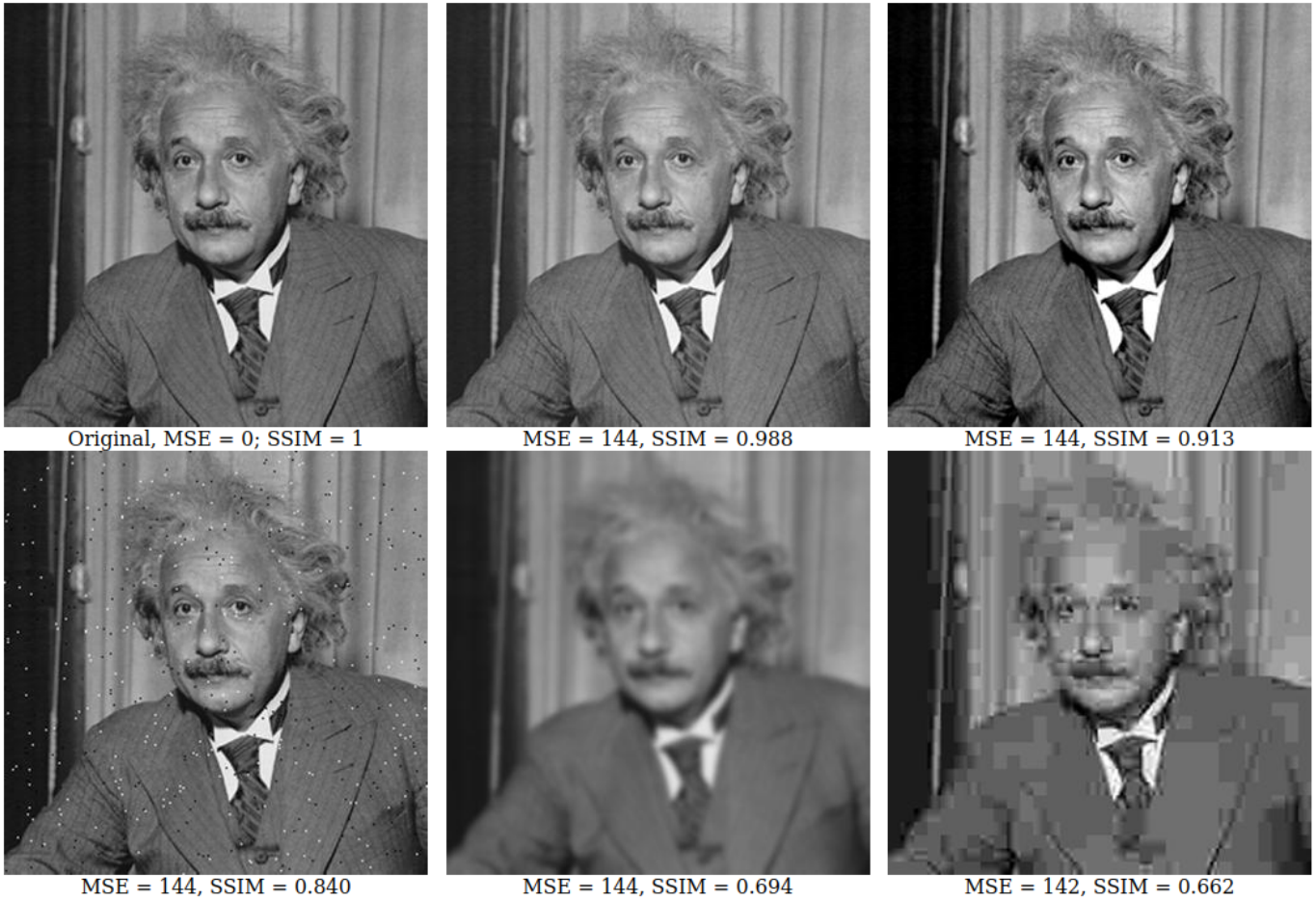

Fig. 5.34.: The SSIM measure, which is used to evaluate the likeness of the image and is compared to the MSE measure. Used with permission from Professor Wang, at the University of Waterloo.

Below, we train a CNN model with the same architecture used to predict the POD modes but using the SSIM as a loss function and present the results below. These results are preliminary as we have not altered the model architecture or training method to improve these results.

Figures 5.35 to 5.36 and figures A.97 to A.104 in the appendix, shows the comparison between these model using the MSE loss function and the SSIM loss function. For brevity, we call the POD modes predicted by the SSIM loss function the 'SSIM pod modes' and the POD modes predicted by the MSE loss function 'the MSE pod modes'.

Generally, the SSIM pod modes look more similar to the true POD modes than the MSE pod modes for the first two POD modes. Here, the SSIM pod modes seem to be more smooth and more accurately capture the similarity of the true POD mode. However, for the remaining 8 POD modes, the MSE POD modes seem more similar to the true POD modes.

Comparing the accuracy of the SSIM pod modes to the MSE pod modes, we attach figures 5.37 to 5.38 and figures A.105 to A.112. Here, we find that the SSIM pod modes are generally less accurate than the MSE pod modes. The SSIM pod modes have a higher median error and a higher variance, with a long tail extending outwards, suggesting that certain predictions are very poor. Despite that, this method could be potentially useful if the model architecture and hyperparameters are tuned properly. The idea behind using the SSIM loss function is that we aim to predict the correct likeness and then compensate by separately predicting the eigenvalues of the POD modes. Although it is a more complicated approach overall, it splits the problem down into two aspects which can be tackled individually,

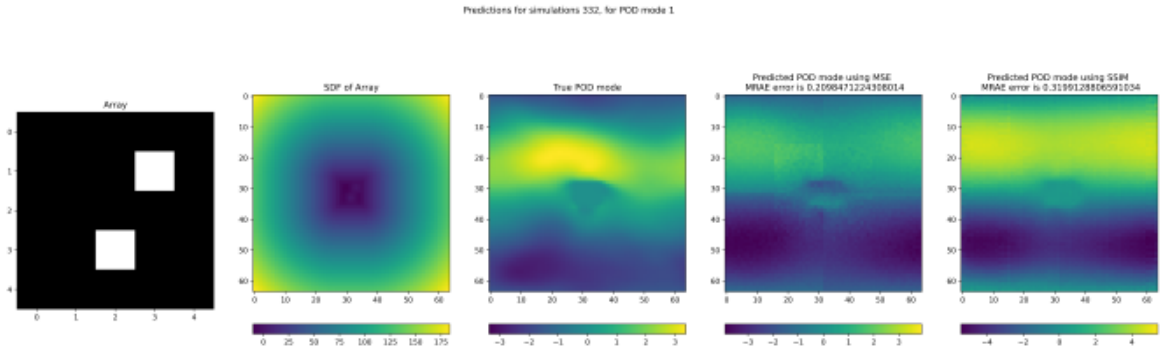

Fig. 5.35.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the first POD mode.

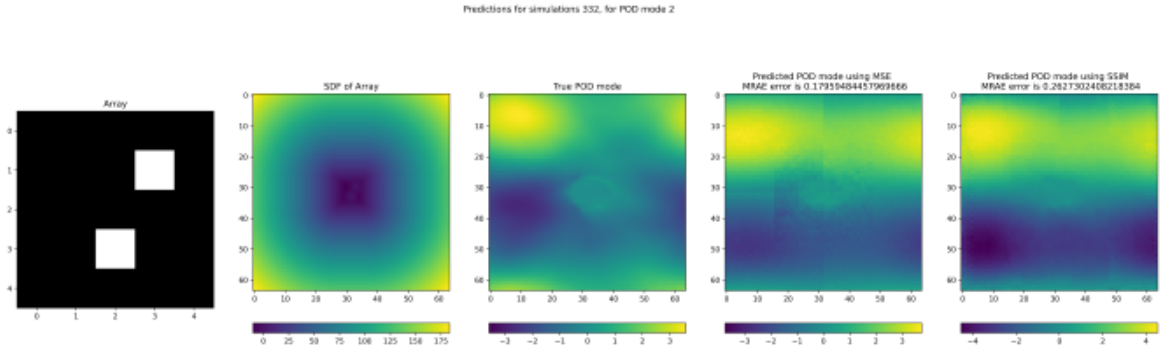

Fig. 5.36.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the second POD mode.

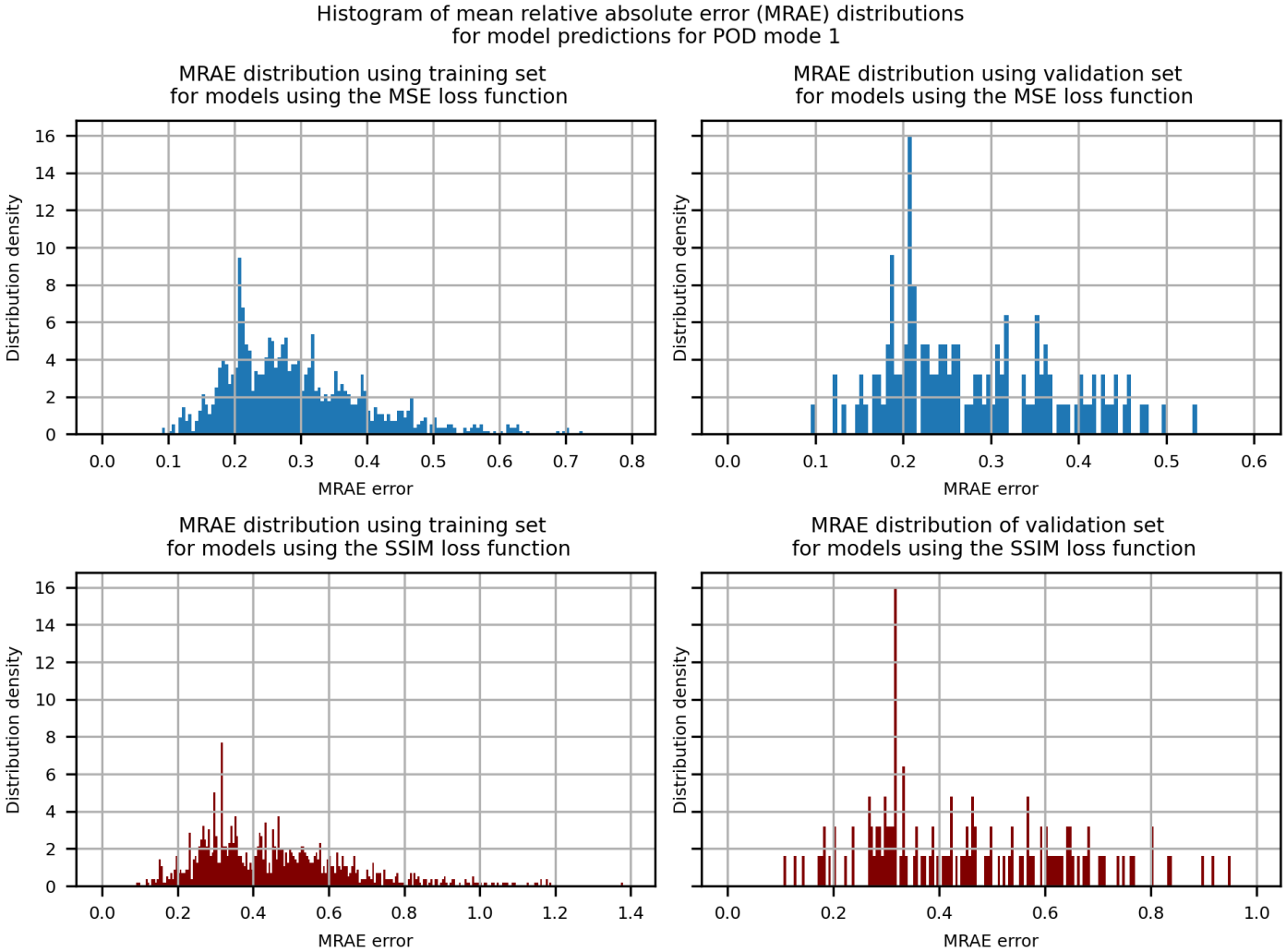

Fig. 5.37.: The normalised histogram for the first POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

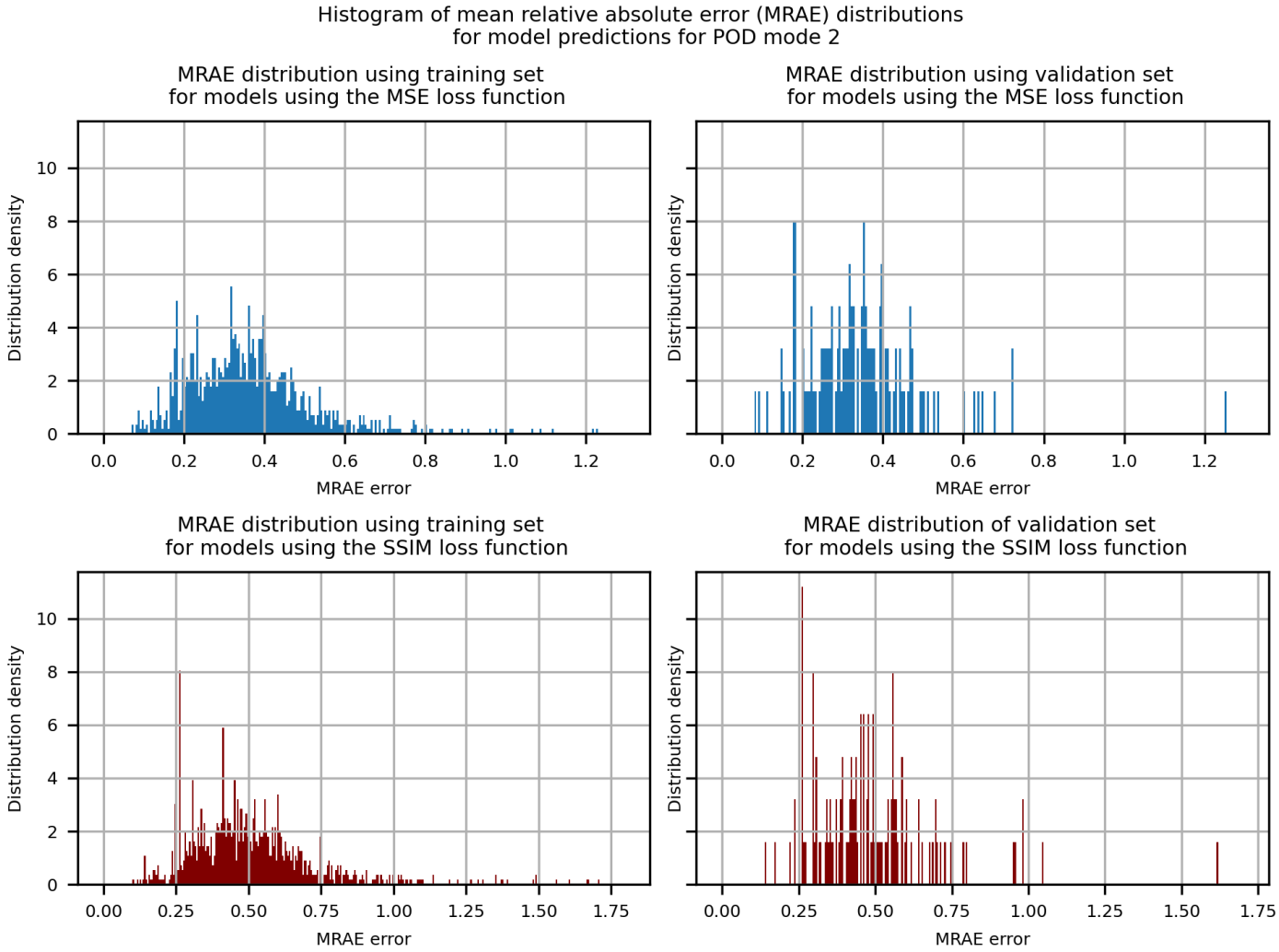

Fig. 5.38.: The normalised histogram for the second POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

6 Conclusion

> *We are just an advanced breed of monkeys on a minor planet of a very average star. But we can understand the Universe. That makes us something very special*
>
> **— Stephen Hawking**

We conclude this thesis by reviewing the initial research questions that we posed. We wished to understand the nature of turbulent flows though complex obstacles, and wanted to understand the relationship between the geometric information of the obstacles, and the turbulent flow going through it. The flow past a bluff body is a well-studied problem, whereas the flow through or past multiple obstacles remains a vastly more complex problem due to the number of possible geometric configurations that an array of obstacles may assume. The size, shape, and placement of each obstacle within an obstacle can vary, which influences the flow around it.

This naturally leads to the use of simulations to evaluate how different geometric configurations affect flow, but we would require many simulations with many different arrays to begin to build a picture of this occurs. Due to the magnitude of processing the data generated by these simulations, it is natural to employ 'big data' techniques such as machine learning to process these data.

We therefore devise a method for predicting turbulent flow solely based on the array's geometric data, while keeping boundary conditions constant. Due to the transient nature of turbulent flow, we had two distinct methods for predicting turbulent flow, the first of which was to use a recurrent neural network (RNNs) to predict the flow. Training RNNs is extremely challenging and computationally expensive. Instead, we were inspired by the method employed by Guo et al. [22], who predicted the steady flow around bluff bodies using convolutional neural networks (CNNs), and modified their method such that CNNs will predict a reduced order form, the POD modes, of the turbulent fluid flow. The goal is for the predicted POD modes to reveal information about the nature of turbulent flows, so that we can investigate how these arrays affect the flow.

The POD method decomposes the flow into two distinct terms, the spatial term and the temporal term. We intend to first predict the spatial term and disregard the temporal term for the time being. This is because it could be argued that the geometric information from the arrays could have a distinct relationship with the spatial term from the POD, such that the CNN model could derive a link between them, whereas this argument is weaker for the temporal term and the geometric information from the arrays. We hope that the temporal term could be predicted by other teams, and therefore could provide the last link towards reconstructing turbulent flows.

Using the LBM, we simulate thousands of 2D flows past random arrays within a five-by-five square grid to generate our dataset. The POD modes are then calculated from the dataset and used as the training and validation set for the CNN model, along with the geometric information of the arrays. After trial and error, we settle on a model architecture based on the mean squared error loss function, which shows some promising predictions but is unable to generalise across all the array types used, particularly arrays with fewer obstacles.

We do find that the predicted POD modes are accurate enough to be used to reconstruct turbulent flows that captures the essence of the large-scale features of the flow, assuming that the time coefficients used are ideal. This represents a first step towards using deep learning methods to understand how the geometric arrangements of obstacles alter turbulent flows. Although the model is not good enough to generalise across all the different arrays, there are many different reasons for this which could be tackled to improve the generalisation of the model.

One potential method is the use of the structural similarity index measure (SSIM) as a loss function for the model. The concept presented here is that there are two aspects to predicting the POD modes: the first is the prediction of the image's 'similarity' and the second is accurately determining the correct range of values that these predictions should encompass. The use of the SSIM loss function appears promising as it is designed to be used to predict the similarity of different images, and therefore could be used to predict the similarity of the POD modes. Using the fact that the L2 norm of the POD modes is the eigenvalue of the POD mode, a second model to predict the eigenvalues of the POD modes would be more effective than a model that attempts to predict both indirectly.

This approach is highly dependent on the determination of the time coefficients for the reconstruction of turbulent flow, a problem that has not been addressed in this work and must be addressed if we wish to reconstruct turbulent flow using the POD method.

To conclude this work, we hope that a first step has been taken toward comprehending the nature of turbulence and, hopefully, toward creating methods of predicting turbulent flows rapidly. There are numerous important applications that necessitate the evaluation of computationally intensive simulations, and we hope that a simple tool could be created that can be used to solve many of these problems quickly and effectively.

Additional figures

A

> *An image is worth a thousand words.*
>
> —

A.1 Calculated POD mode example, for POD modes from three to ten

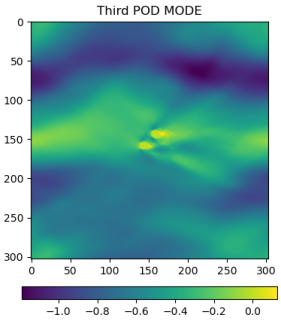

Fig. A.1.: Third POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

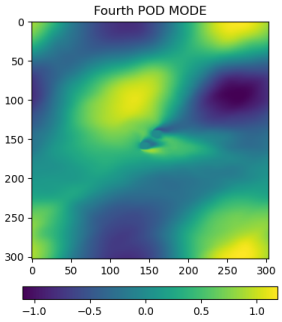

Fig. A.2.: Fourth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

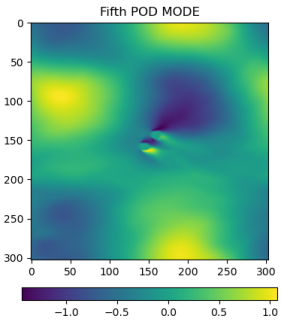

Fig. A.3.: Fifth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

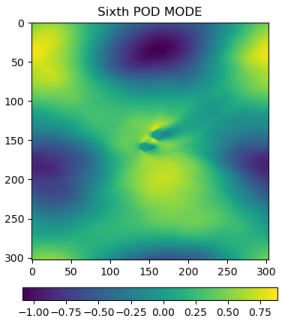

Fig. A.4.: Sixth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

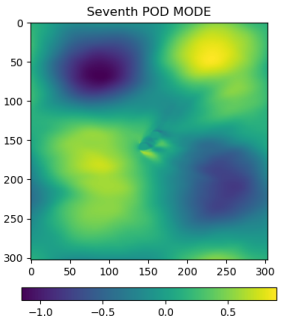

Fig. A.5.: Seventh POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

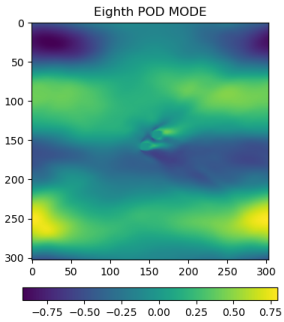

Fig. A.6.: Eighth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

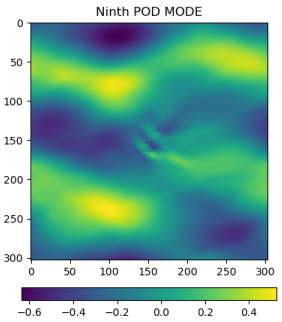

Fig. A.7.: Ninth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

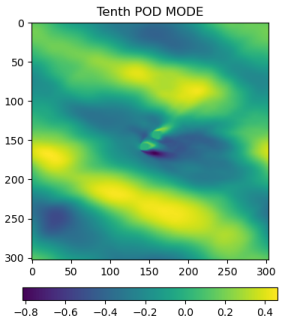

Fig. A.8.: Tenth POD mode of the simulated flow through an array of obstacles seen in figure 5.7.

A.2 Epoch Training graphs for POD modes 3 to 10

A.2.1 Full Epoch training graphs

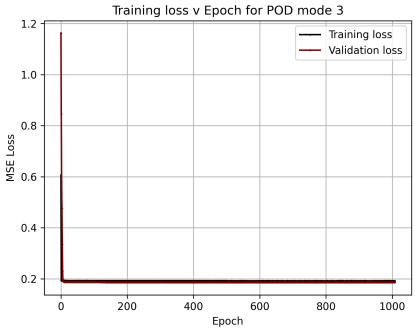

Fig. A.9.: Epoch loss graph for POD mode 3

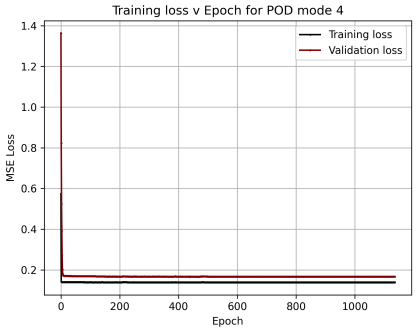

Fig. A.10.: Epoch loss graph for POD mode 4

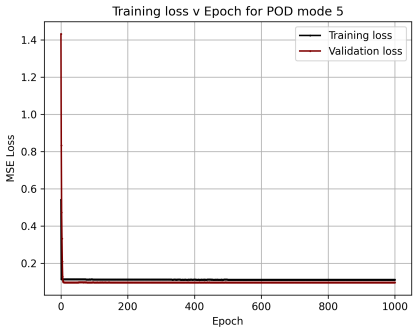

Fig. A.11.: Epoch loss graph for POD mode 5

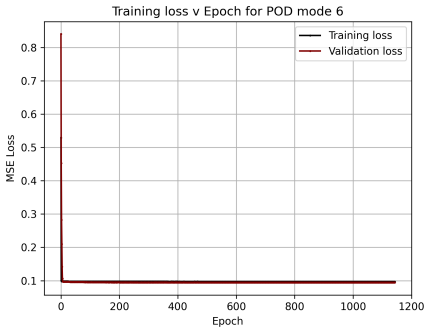

Fig. A.12.: Epoch loss graph for POD mode 6

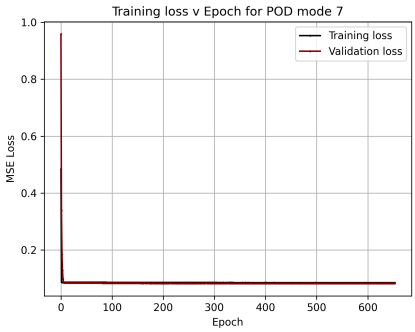

Fig. A.13.: Epoch loss graph for POD mode 7

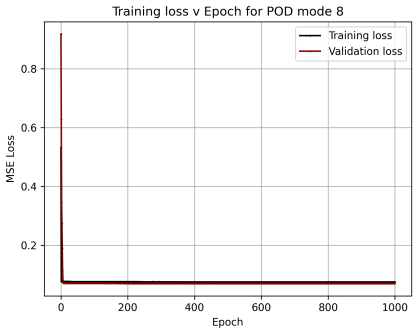

Fig. A.14.: Epoch loss graph for POD mode 8

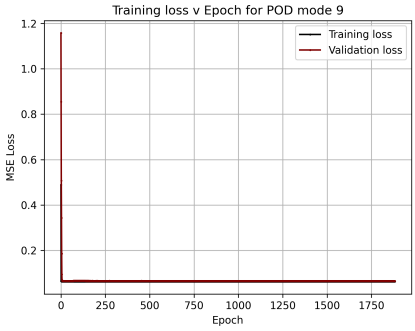

Fig. A.15.: Epoch loss graph for POD mode 9

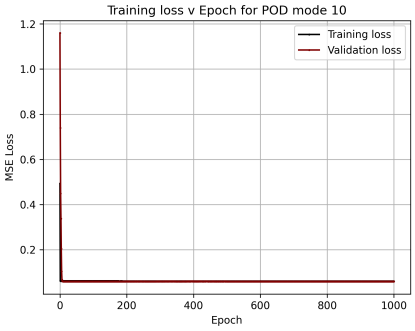

Fig. A.16.: Epoch loss graph for POD mode 10

A.2.2 Cropped Epoch training graphs

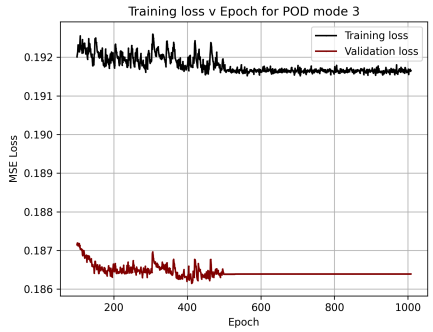

Fig. A.17.: Cropped epoch loss graph for POD mode 3

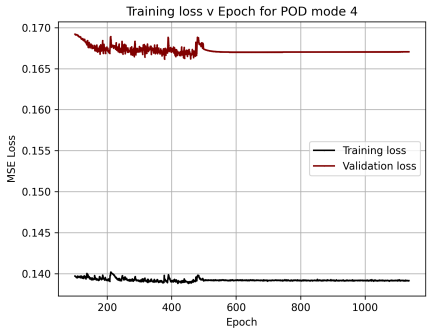

Fig. A.18.: Cropped epoch loss graph for POD mode 4

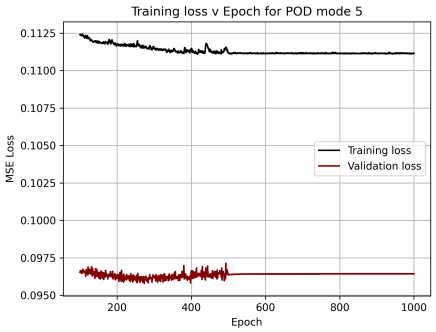

Fig. A.19.: Cropped epoch loss graph for POD mode 5

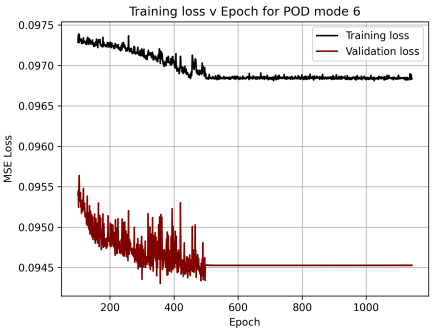

Fig. A.20.: Cropped epoch loss graph for POD mode 6

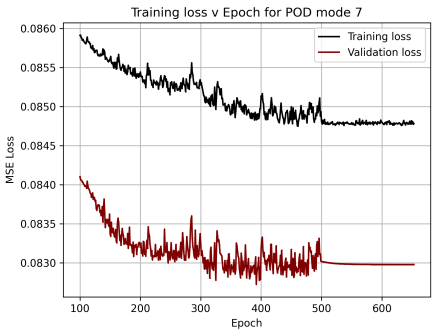

Fig. A.21.: Cropped epoch loss graph for POD mode 7

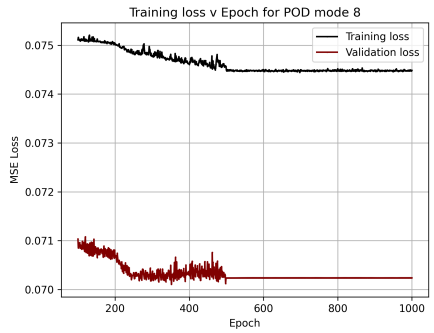

Fig. A.22.: Cropped epoch loss graph for POD mode 8

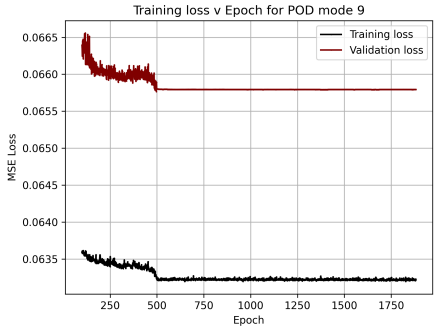

Fig. A.23.: Cropped epoch loss graph for POD mode 9

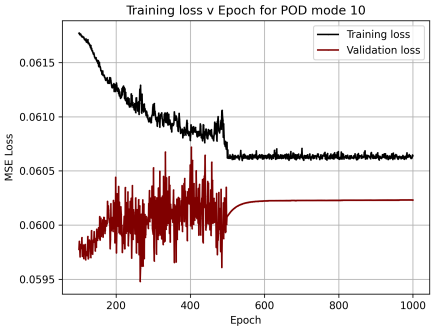

Fig. A.24.: Cropped epoch loss graph for POD mode 10

A.3 Model error histograms for POD modes 3 to 10

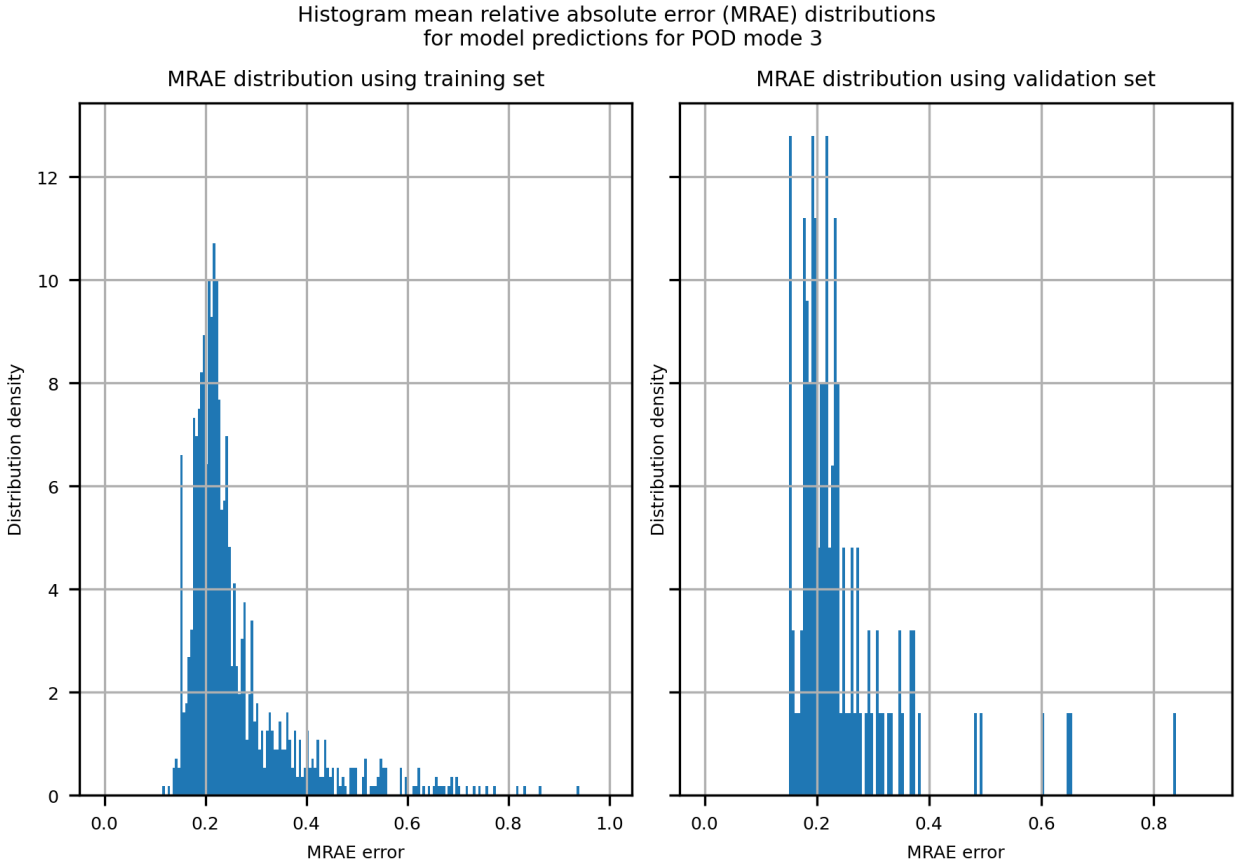

Fig. A.25.: The normalised histogram showing the distribution of MRAE error for the third POD mode, separated between the training set and validation set.

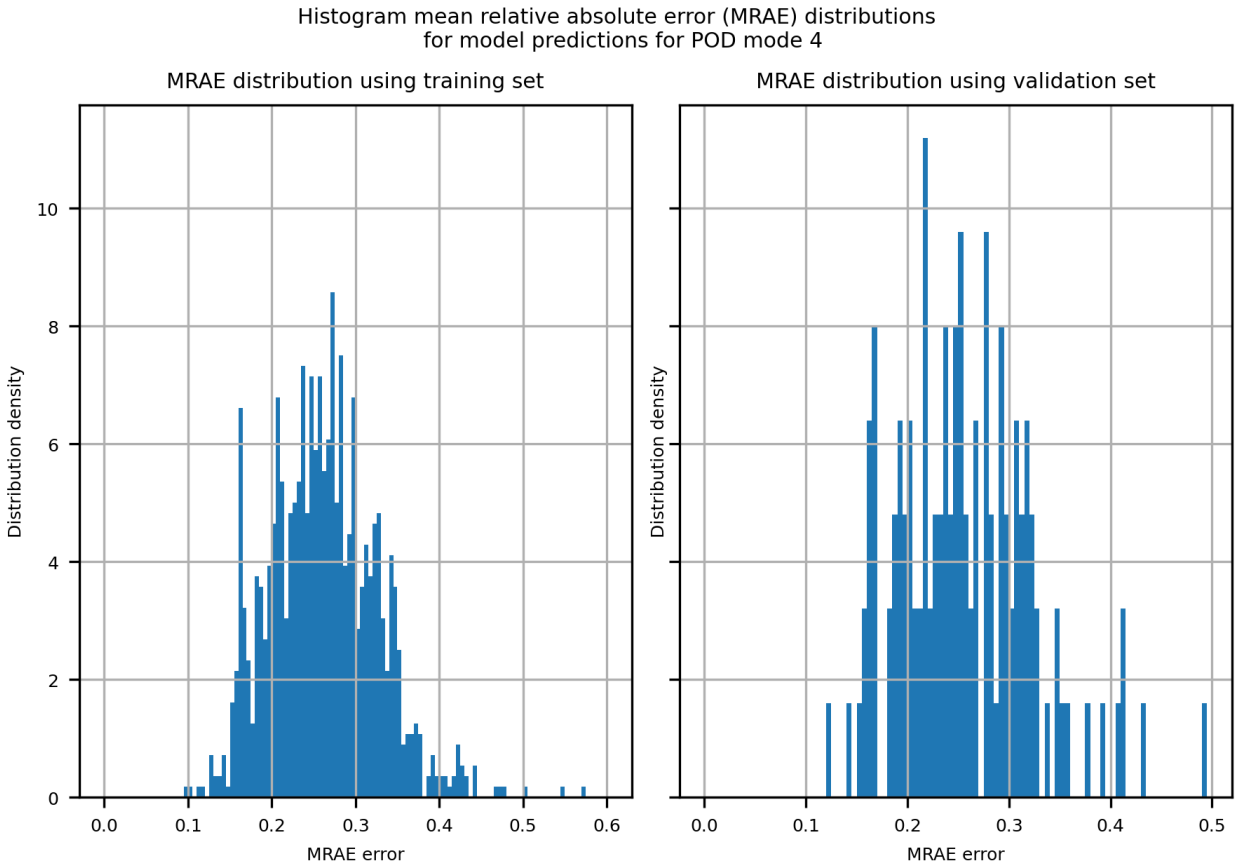

Fig. A.26.: The normalised histogram showing the distribution of MRAE error for fourth first POD mode, separated between the training set and validation set.

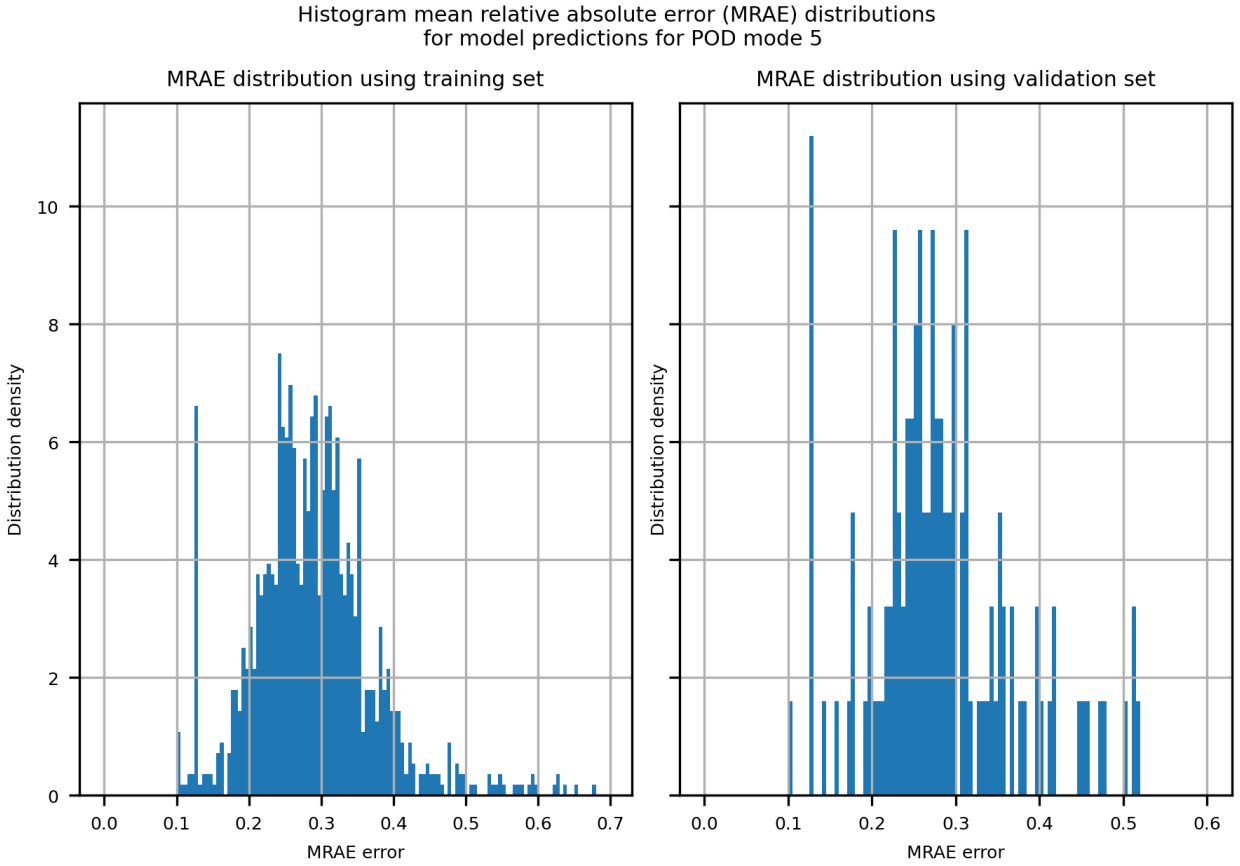

Fig. A.27.: The normalised histogram showing the distribution of MRAE error for the fifth POD mode, separated between the training set and validation set.

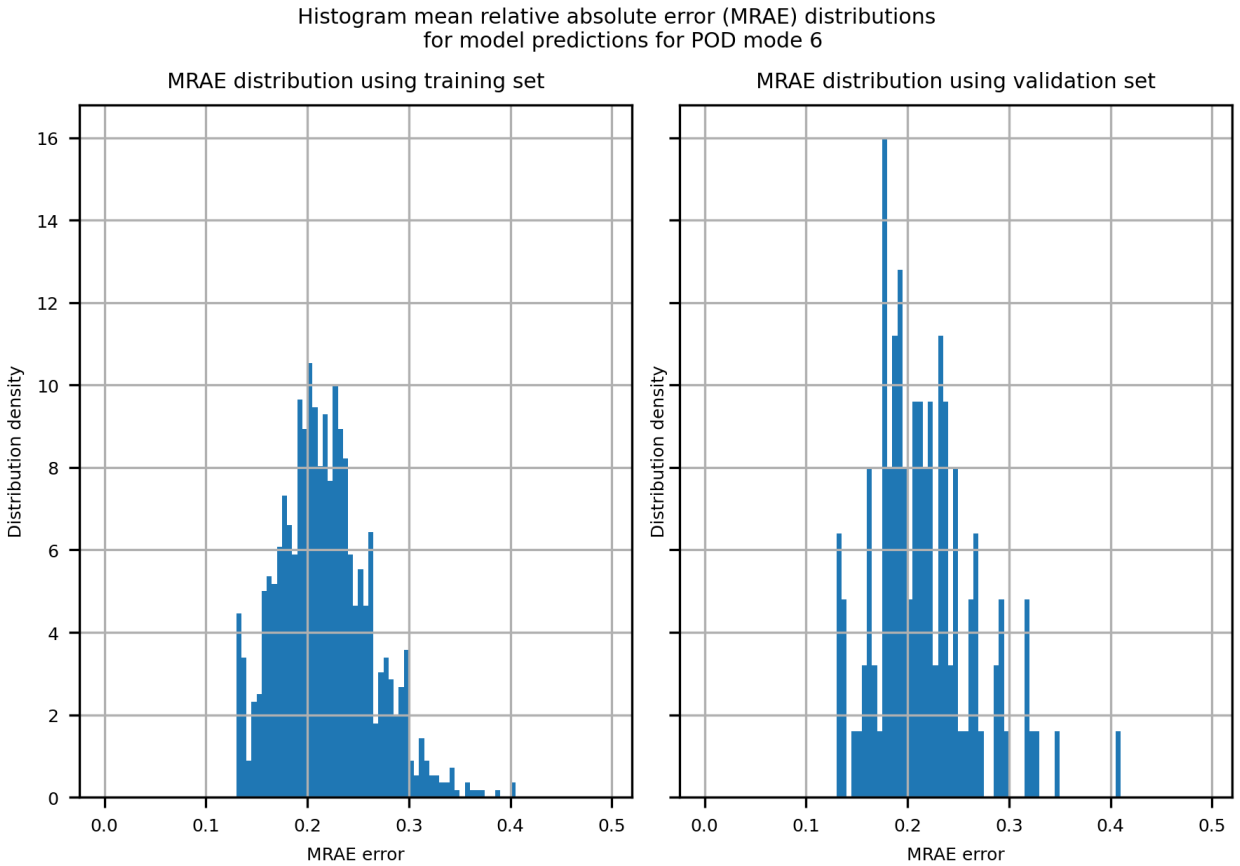

Fig. A.28.: The normalised histogram showing the distribution of MRAE error for the sixth POD mode, separated between the training set and validation set.

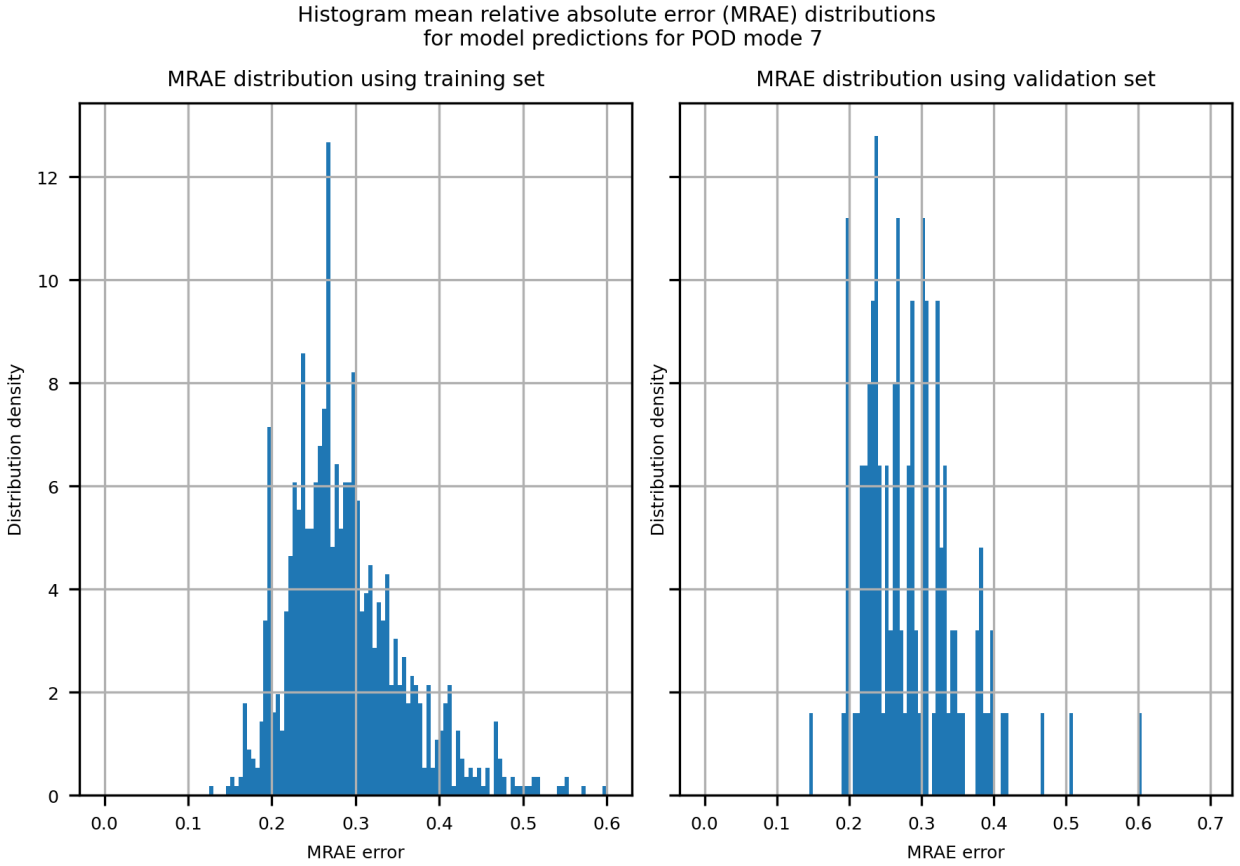

Fig. A.29.: The normalised histogram showing the distribution of MRAE error for the seventh POD mode, separated between the training set and validation set.

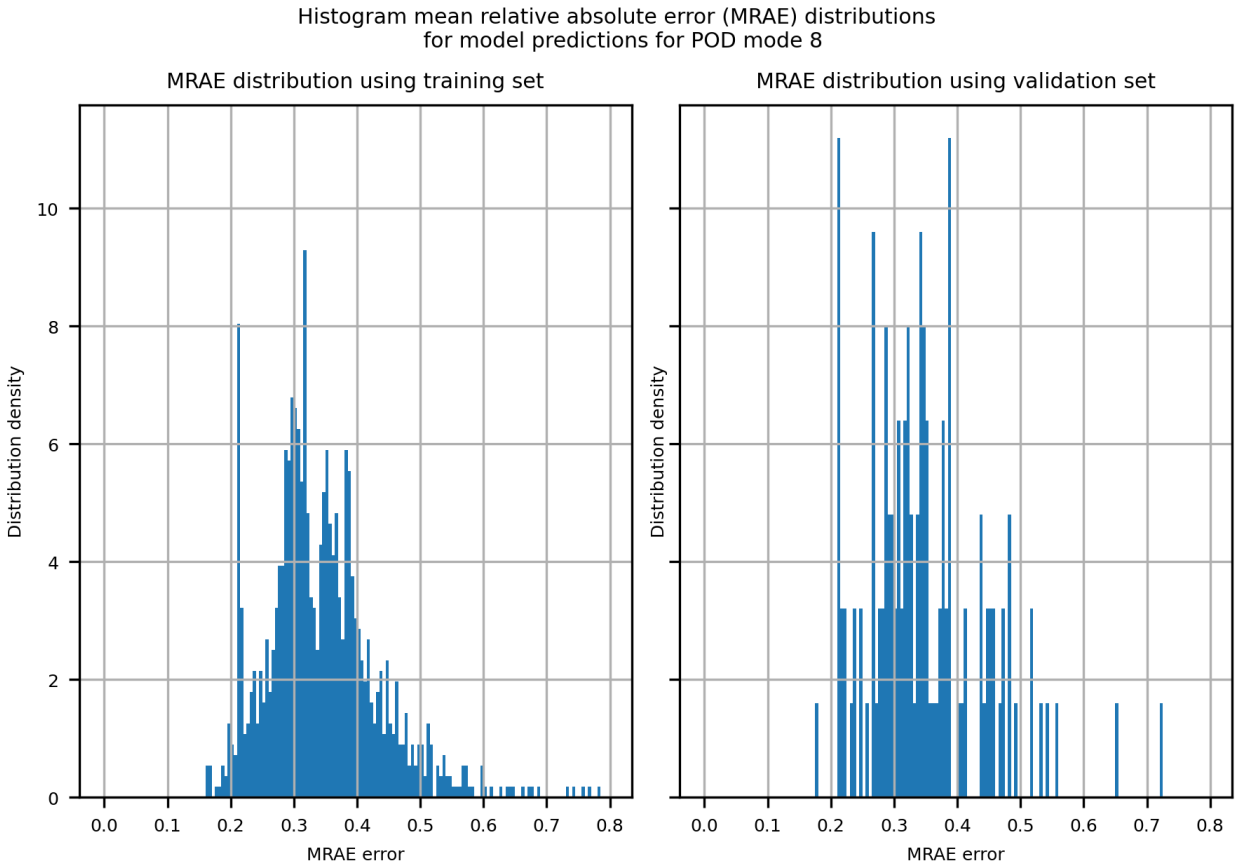

Fig. A.30.: The normalised histogram showing the distribution of MRAE error for the eighth POD mode, separated between the training set and validation set.

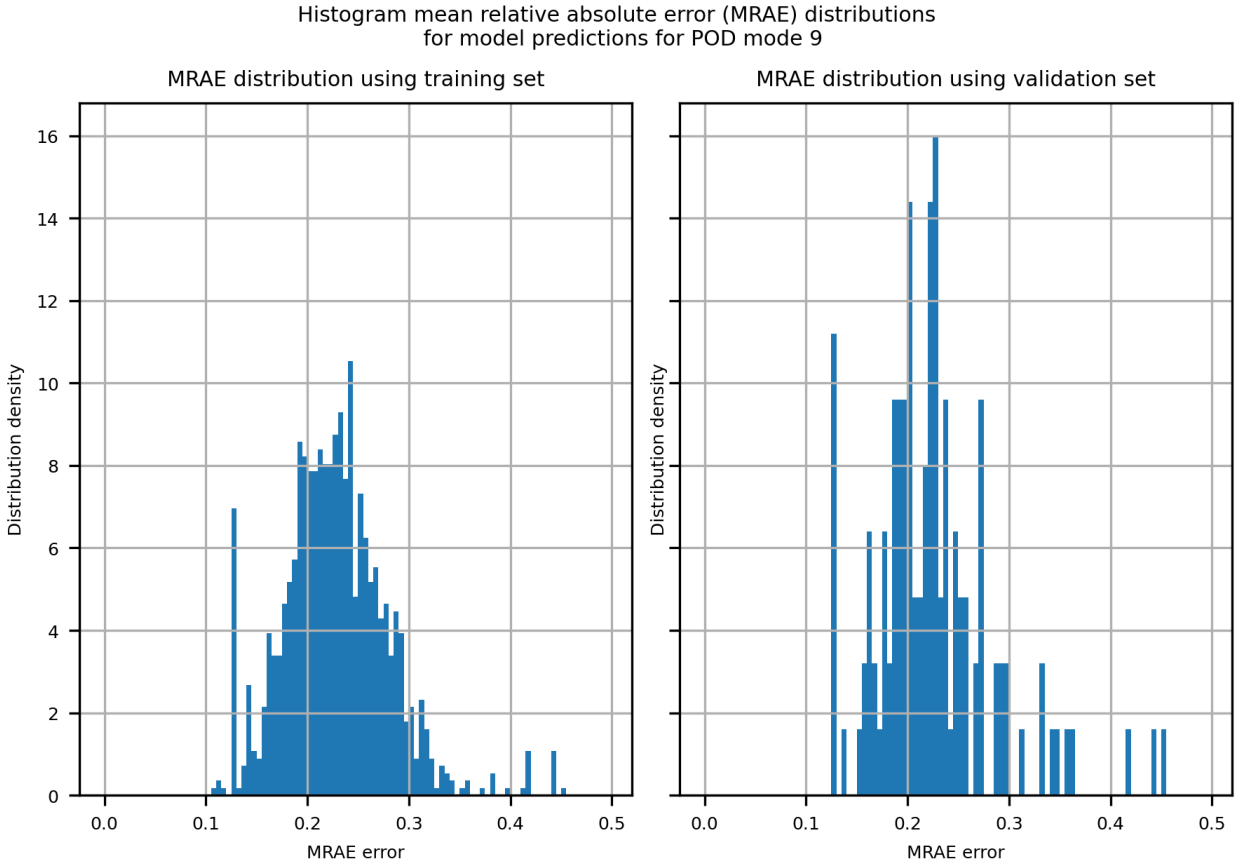

Fig. A.31.: The normalised histogram showing the distribution of MRAE error for the ninth POD mode, separated between the training set and validation set.

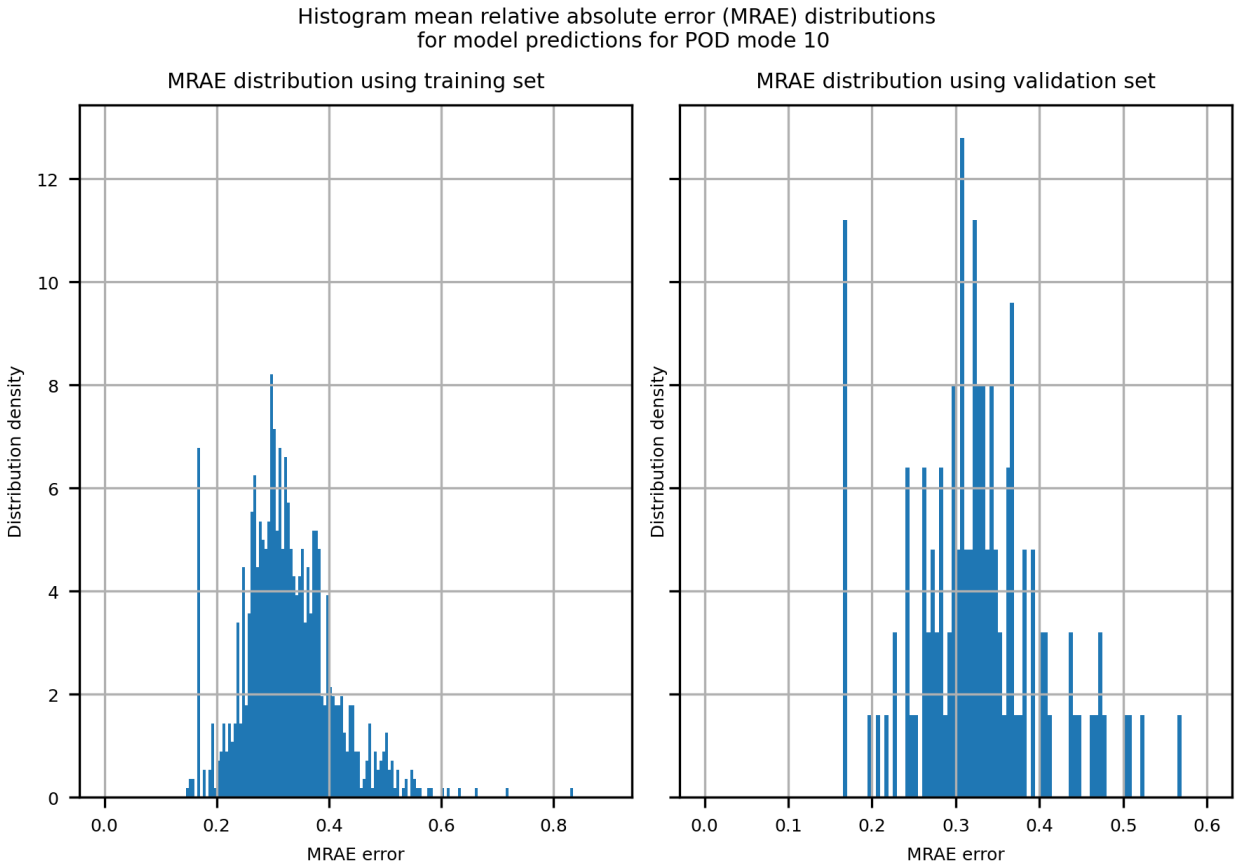

Fig. A.32.: The normalised histogram showing the distribution of MRAE error for the tenth POD mode, separated between the training set and validation set.

A.4 Error distribution compared against obstacle number plots for POD modes 3 to 10

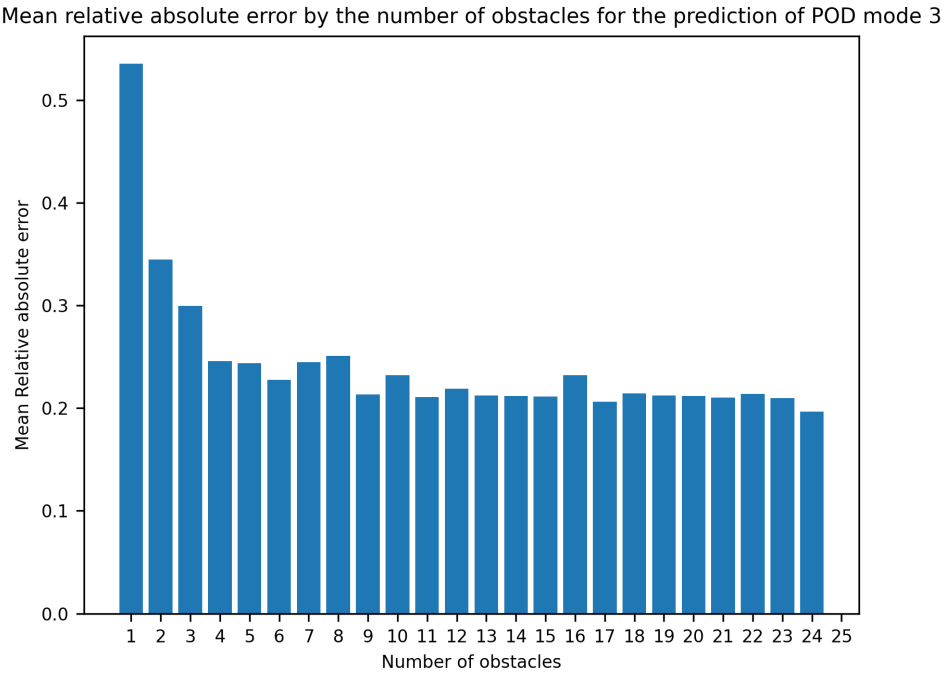

Fig. A.33.: A histogram showing the error distribution against the number of obstacles in the array for the third POD mode.

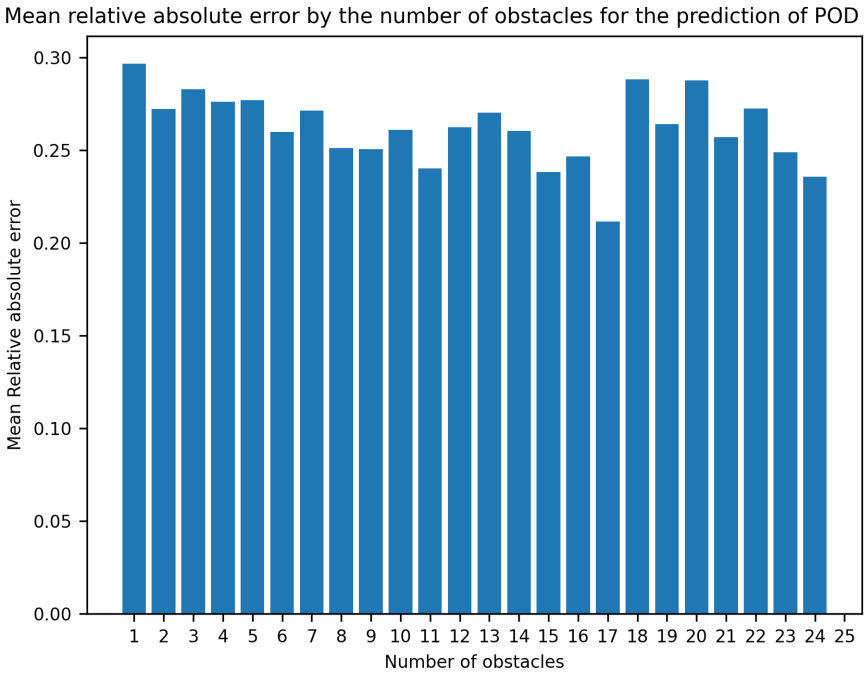

Fig. A.34.: A histogram showing the error distribution against the number of obstacles in the array for the fourth POD mode.

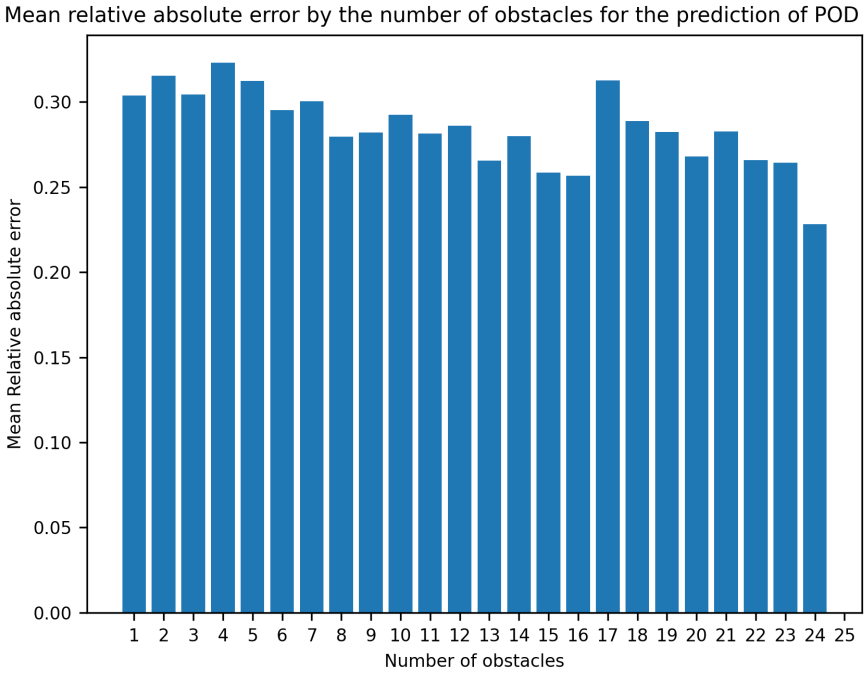

Fig. A.35.: A histogram showing the error distribution against the number of obstacles in the array for the fifth POD mode.

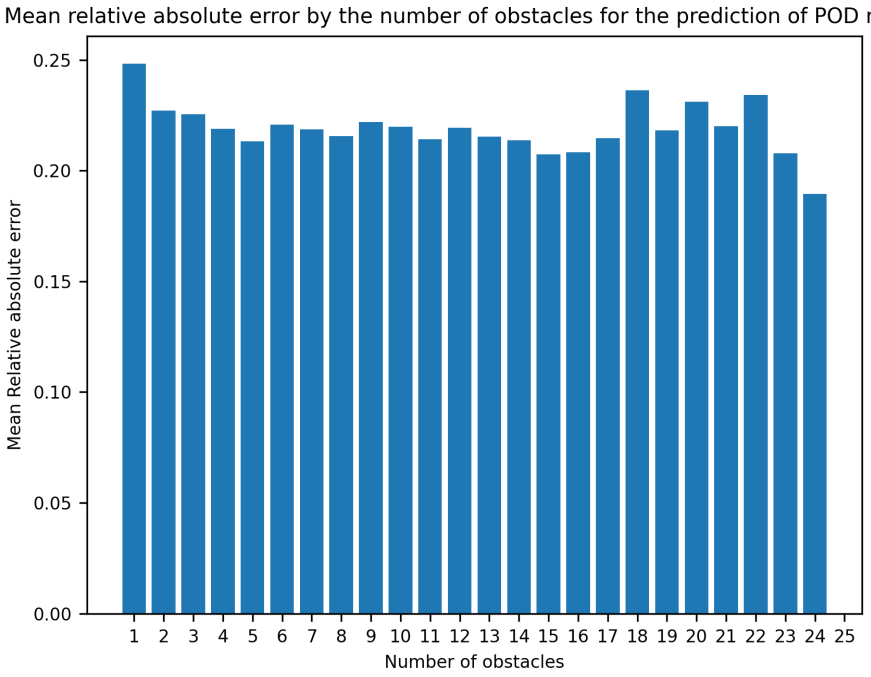

Fig. A.36.: A histogram showing the distribution of error against the number of obstacles in the array for the sixth POD mode.

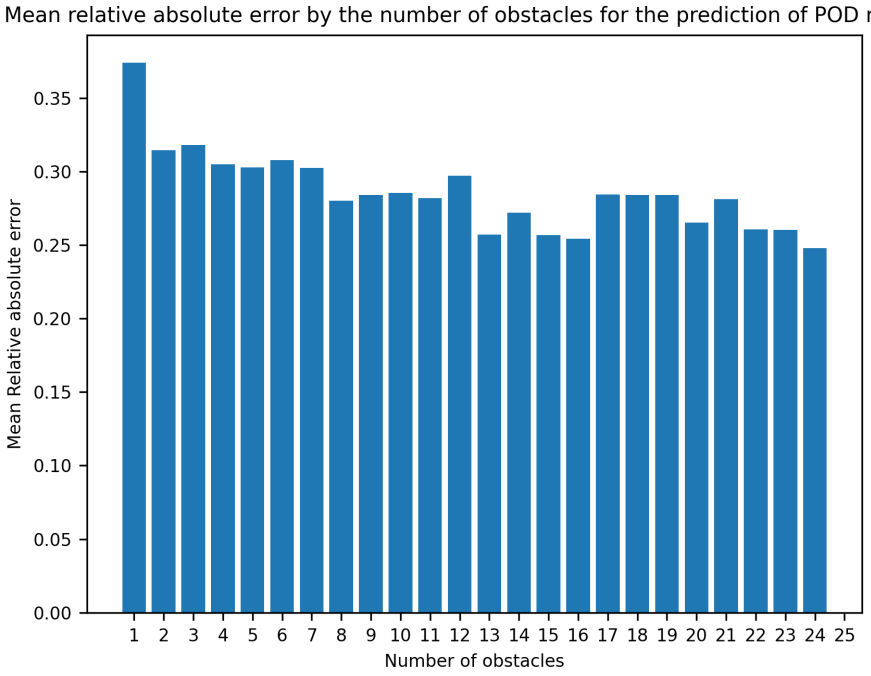

Fig. A.37.: A histogram showing the distribution of error against the number of obstacles in the array for the seventh POD mode.

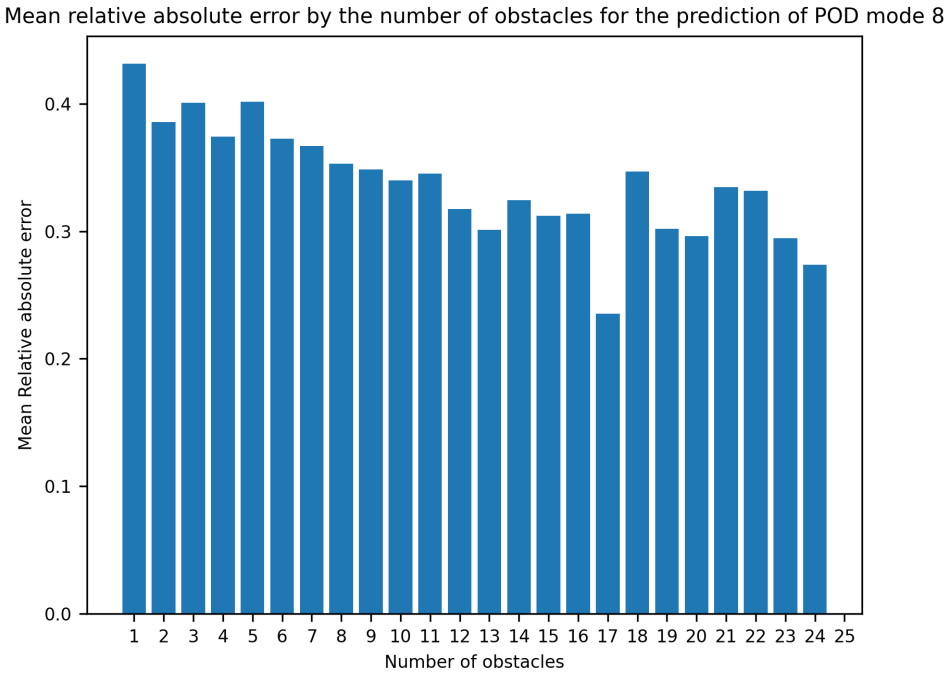

Fig. A.38.: A histogram showing the distribution of error against the number of obstacles in the array for the eighth POD mode.

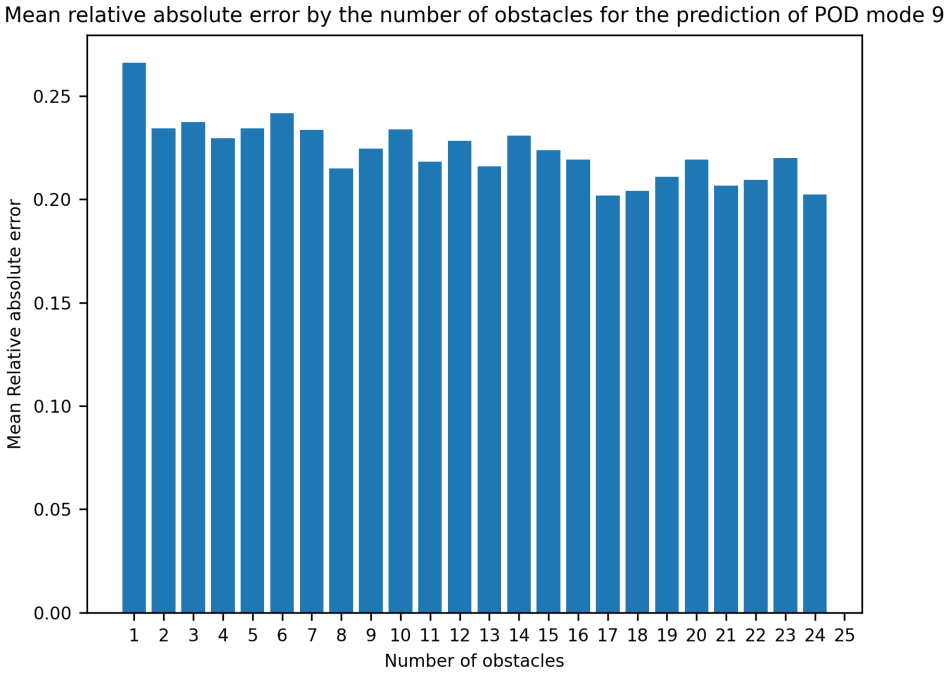

Fig. A.39.: A histogram showing the distribution of error against the number of obstacles in the array for the ninth POD mode.

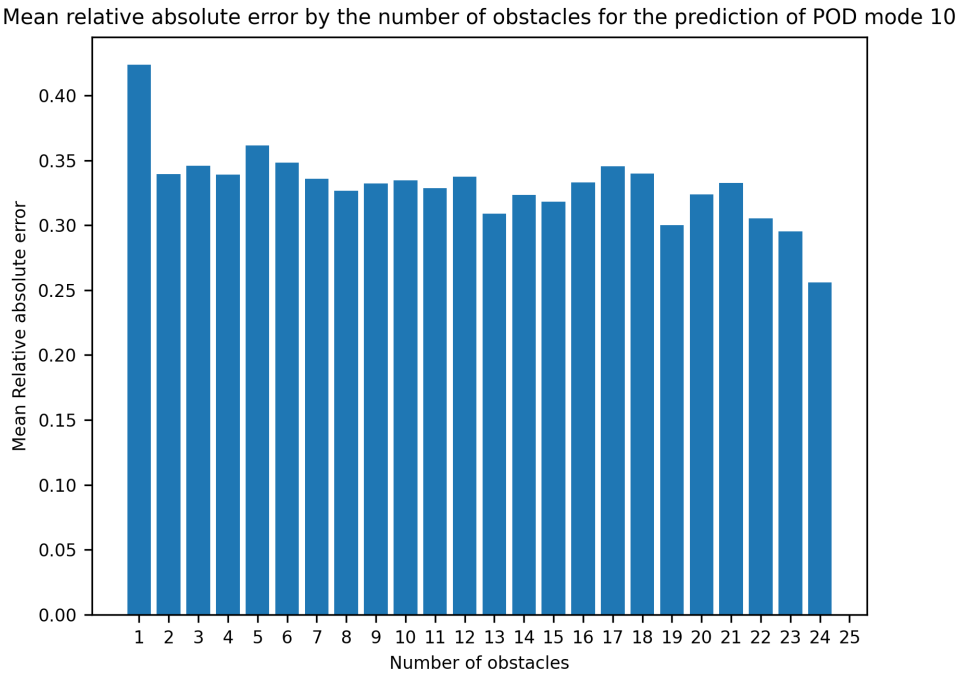

Fig. A.40.: A histogram showing the distribution of error against the number of obstacles in the array for the tenth POD mode.

A.5 Best POD mode predictions

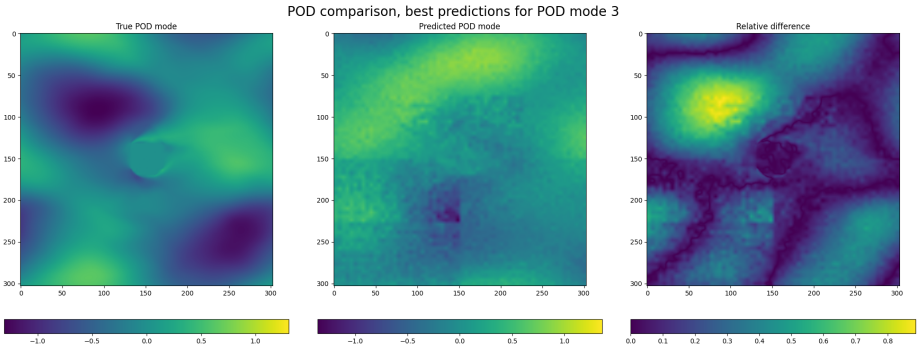

Fig. A.41.: Best ML-predicted POD mode for POD mode 3.

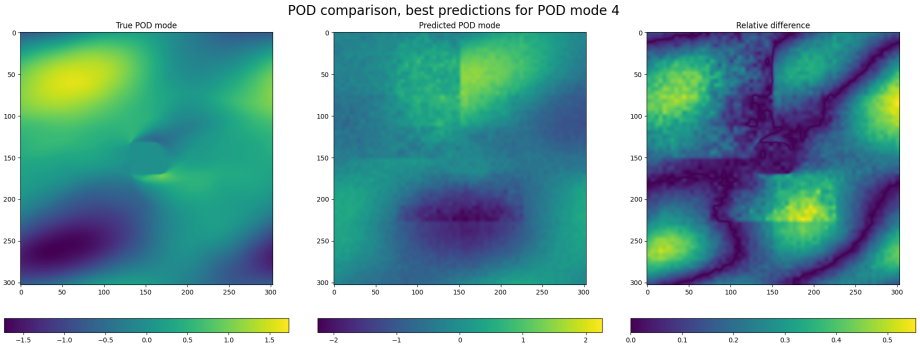

Fig. A.42.: Best ML-predicted POD mode for POD mode 4.

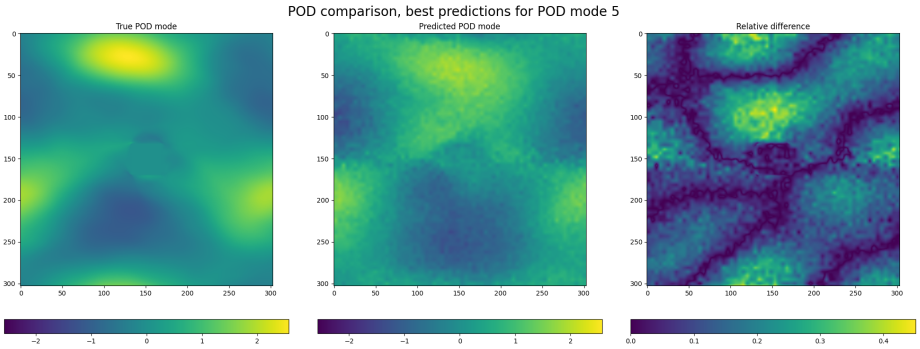

Fig. A.43.: Best ML-predicted POD mode for POD mode 5.

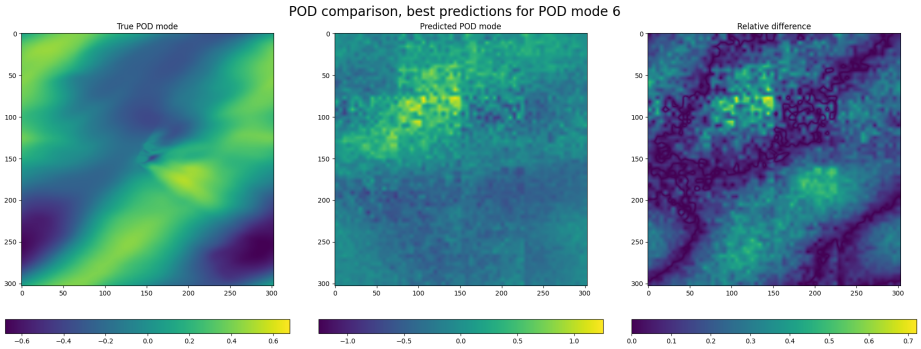

Fig. A.44.: Best ML-predicted POD mode for POD mode 6.

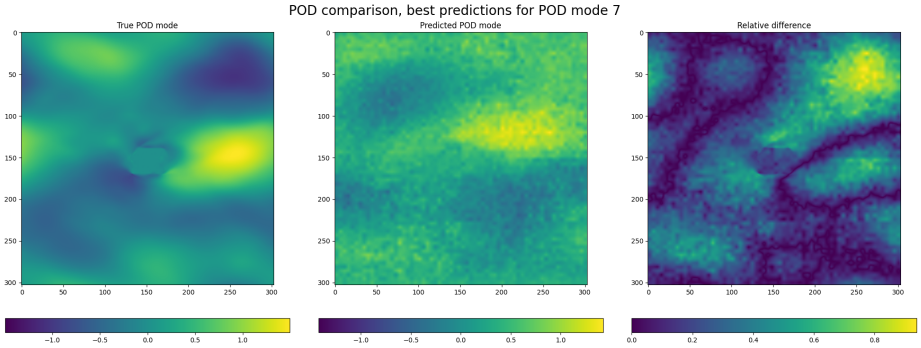

Fig. A.45.: Best ML-predicted POD mode for POD mode 7.

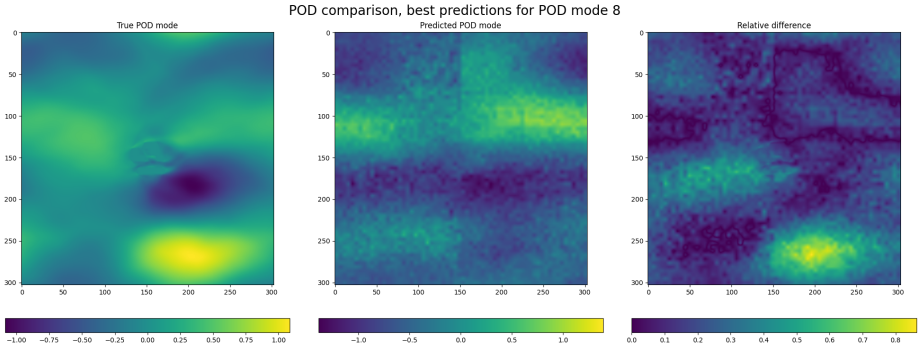

Fig. A.46.: Best ML-predicted POD mode for POD mode 8.

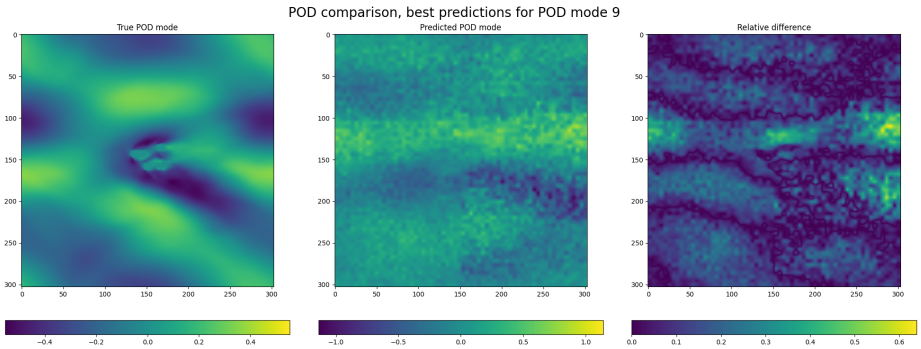

Fig. A.47.: Best ML-predicted POD mode for POD mode 9.

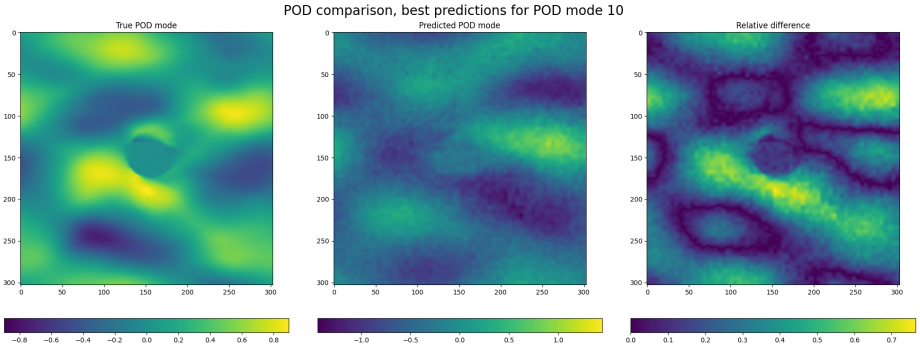

Fig. A.48.: Best ML-predicted POD mode for POD mode 10.

A.6 POD mode comparison

A.6.1 Case 332

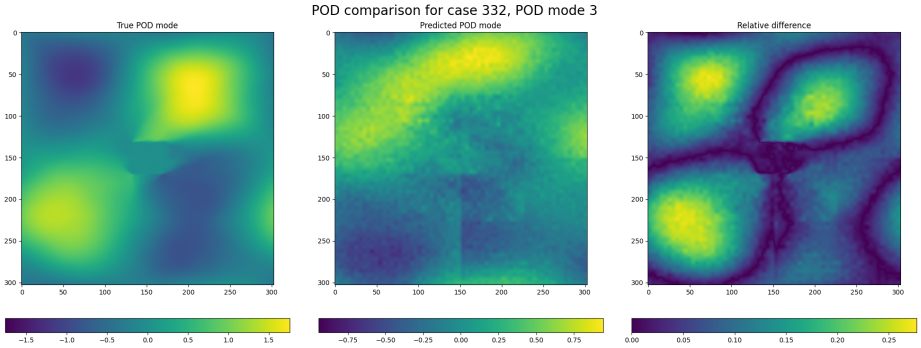

Fig. A.49.: POD mode 3 comparison for array 332.

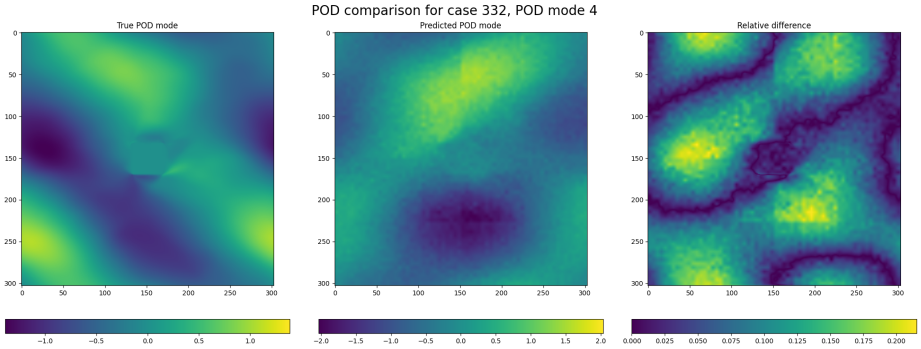

Fig. A.50.: POD mode 4 comparison for array 332.

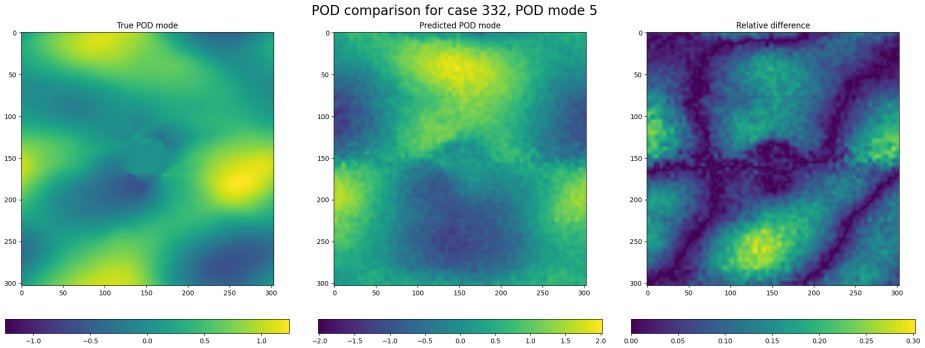

Fig. A.51.: POD mode 5 comparison for array 332.

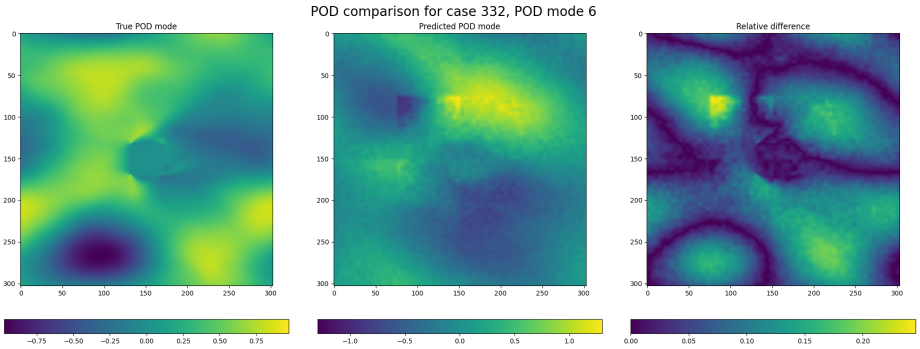

Fig. A.52.: POD mode 6 comparison for array 332.

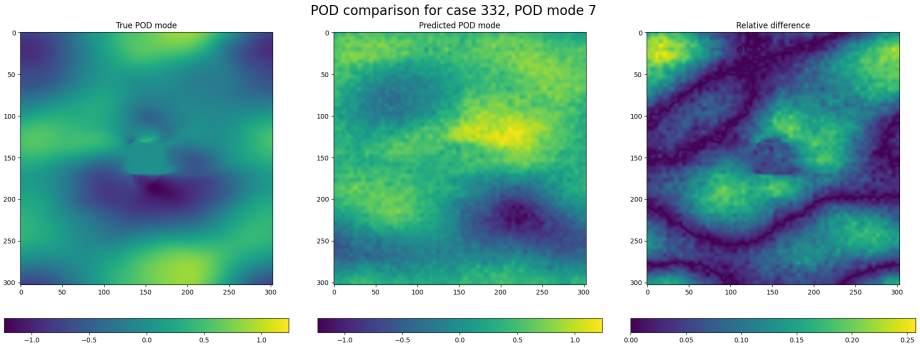

Fig. A.53.: POD mode 7 comparison for array 332.

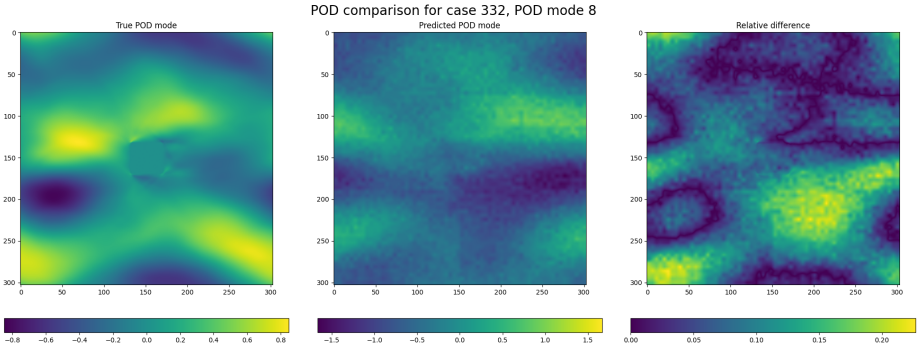

Fig. A.54.: POD mode 8 comparison for array 332.

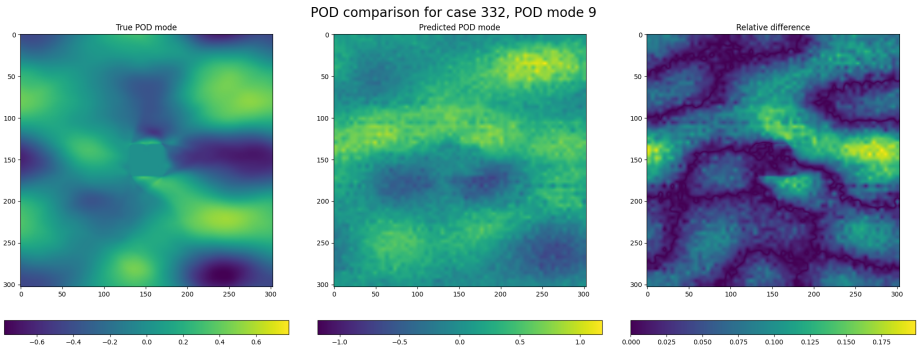

Fig. A.55.: POD mode 9 comparison for array 332.

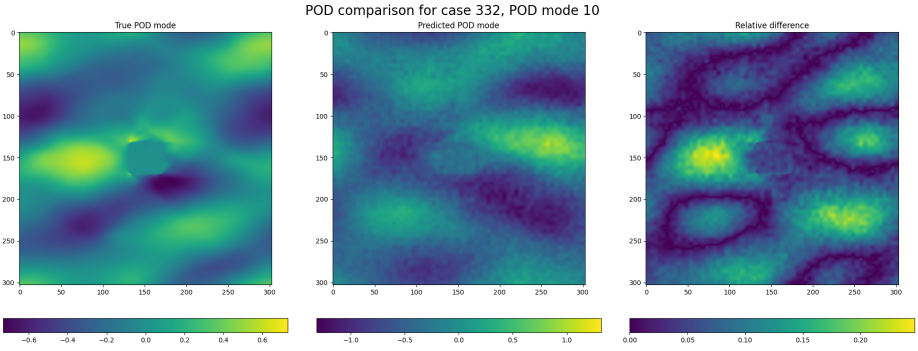

Fig. A.56.: POD mode 10 comparison for array 332.

A.6.2 Case 33

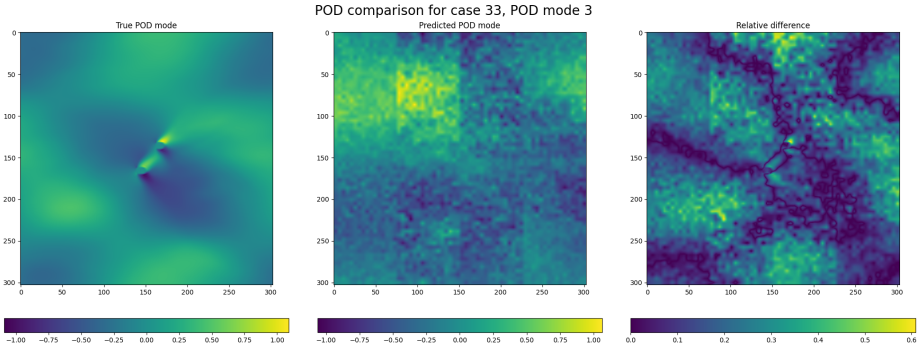

Fig. A.57.: POD mode 3 comparison for array 33.

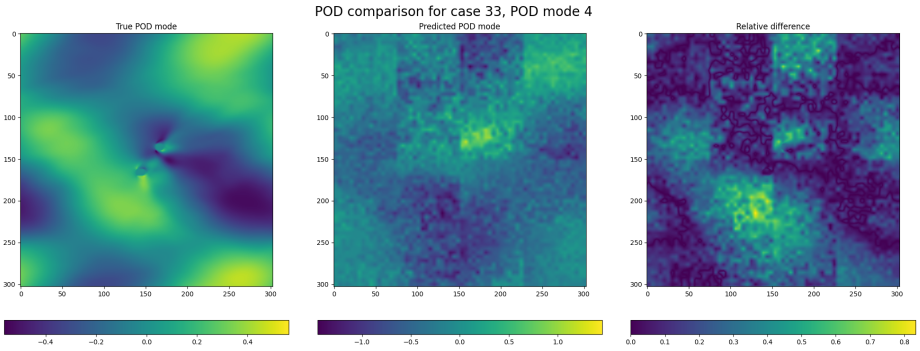

Fig. A.58.: POD mode 4 comparison for array 33.

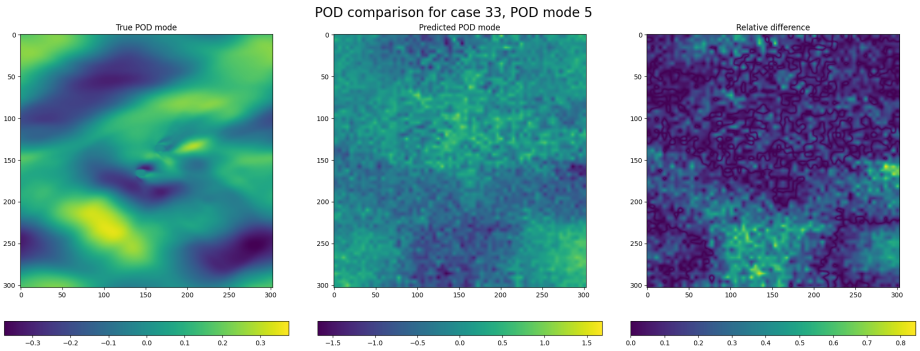

Fig. A.59.: POD mode 5 comparison for array 33.

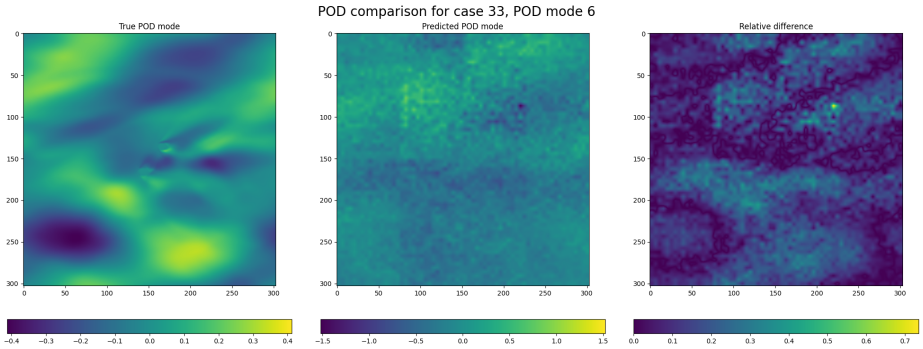

Fig. A.60.: POD mode 6 comparison for array 33.

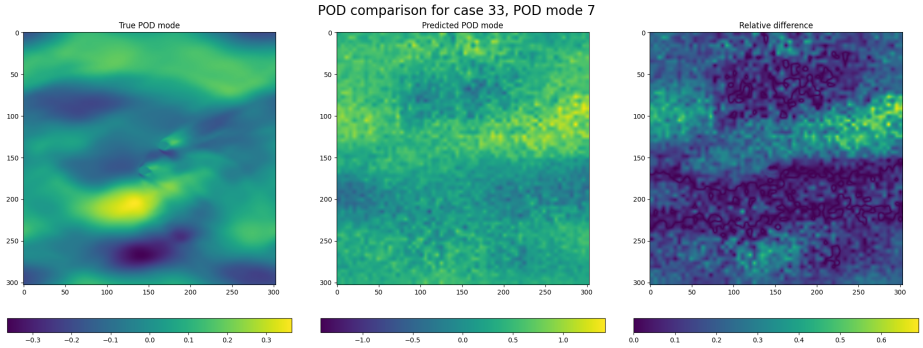

Fig. A.61.: POD mode 7 comparison for array 33.

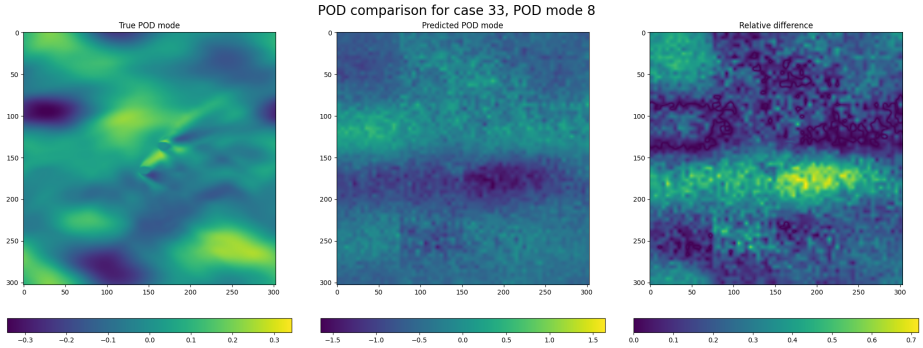

Fig. A.62.: POD mode 8 comparison for array 33.

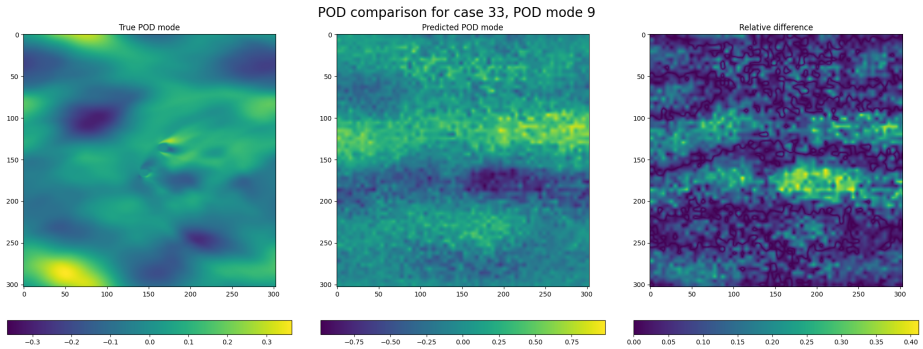

Fig. A.63.: POD mode 9 comparison for array 33.

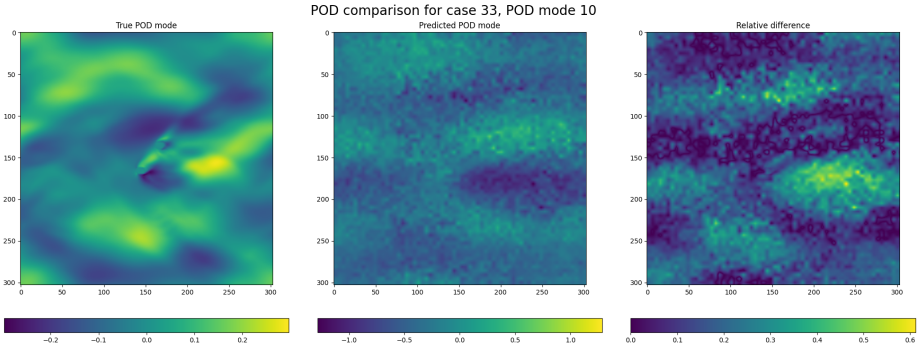

Fig. A.64.: POD mode 10 comparison for array 33.

A.7 Reconstruction snapshots

A.7.1 Case 332

A.7.2 Case 33

A.7.3 Case 332 with rescaled POD modes

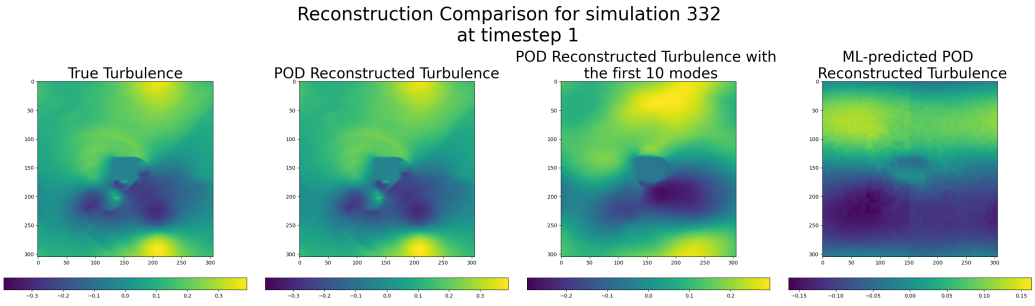

Fig. A.65.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 1 for case 332.

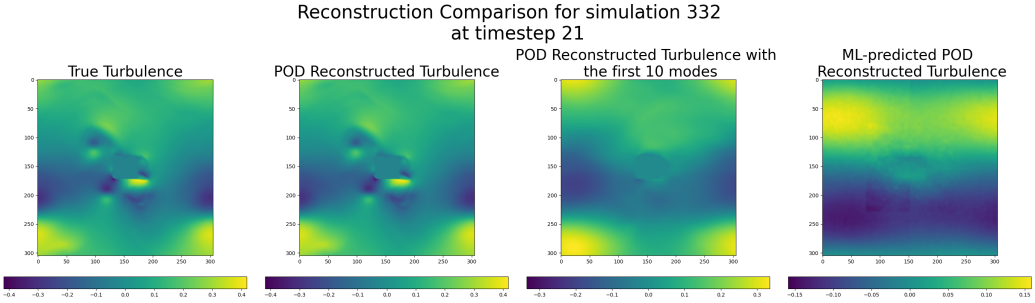

Fig. A.66.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21 for case 332.

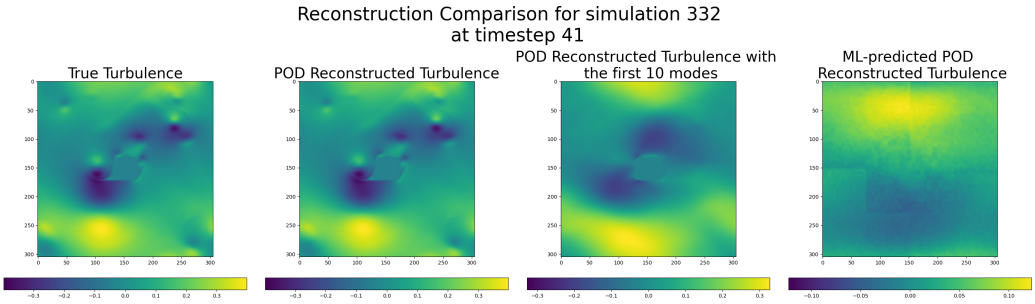

Fig. A.67.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41 for case 332.

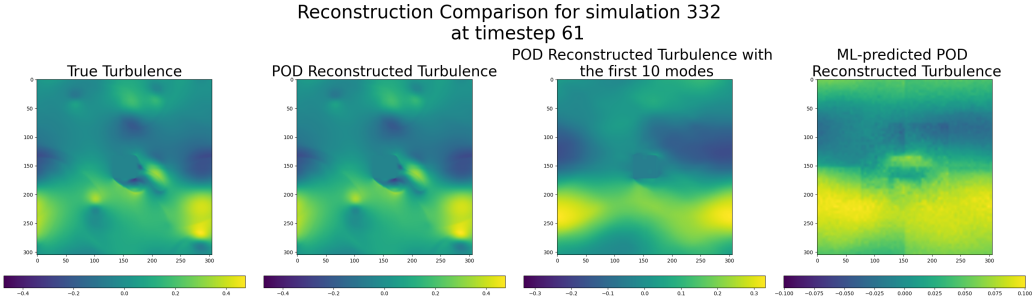

Fig. A.68.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61 for case 332.

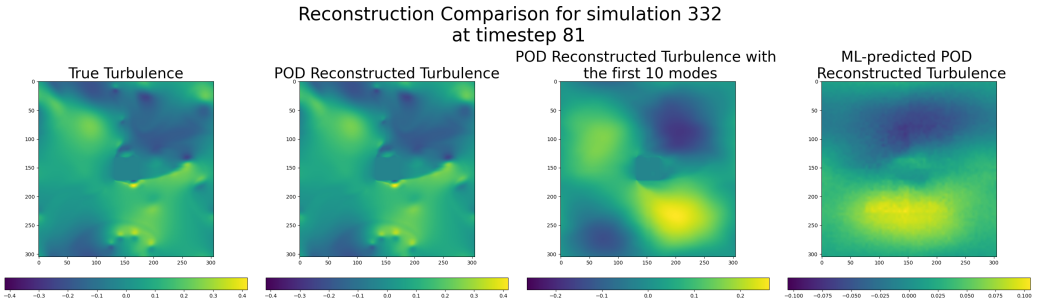

Fig. A.69.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81 for case 332.

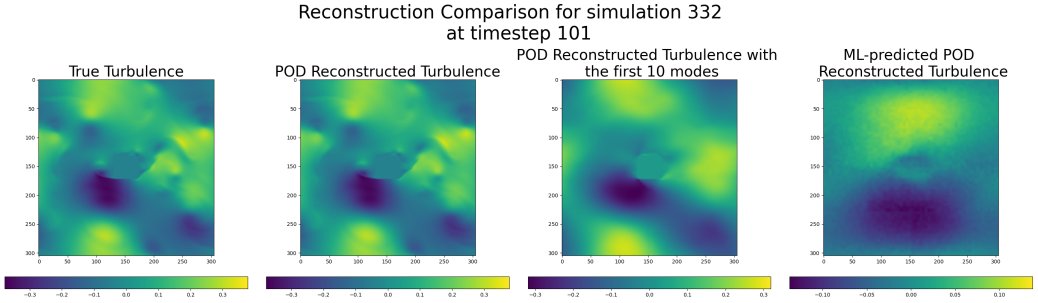

Fig. A.70.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101 for case 332.

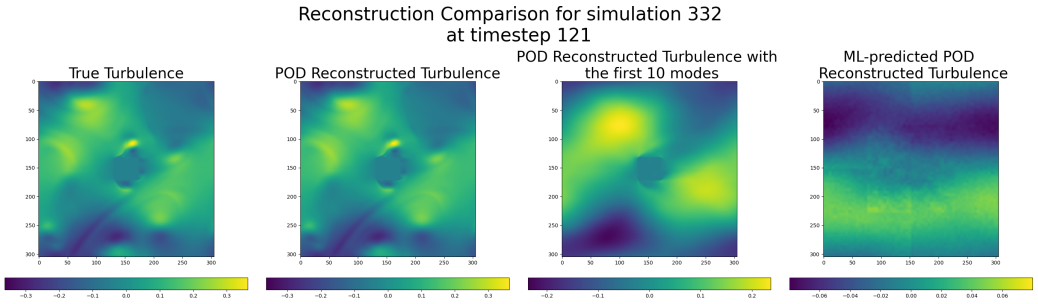

Fig. A.71.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121 for case 332.

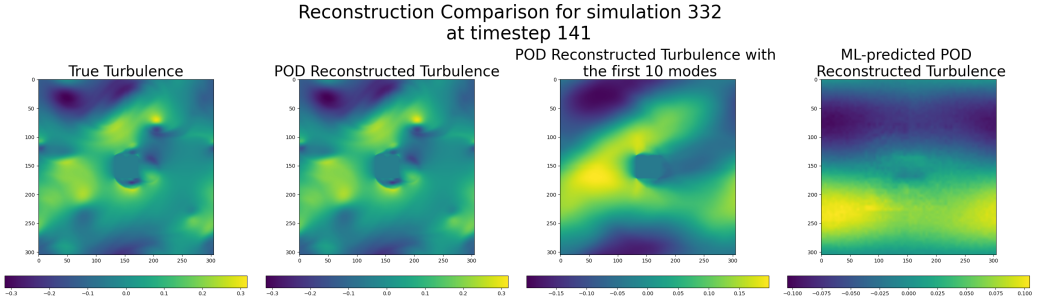

Fig. A.72.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141 for case 332.

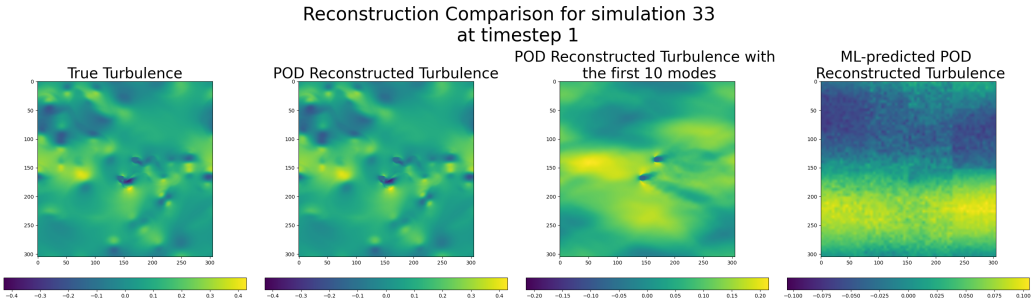

Fig. A.73.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 1 for case 33.

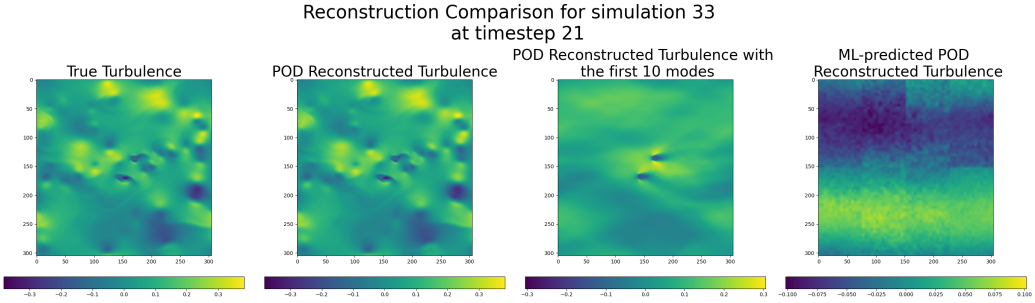

Fig. A.74.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21 for case 33.

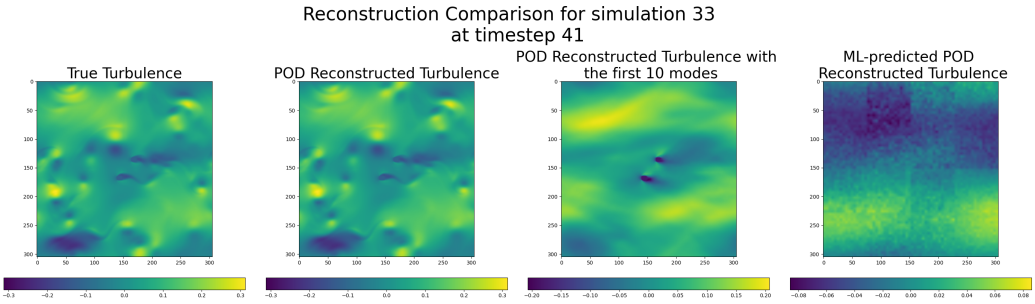

Fig. A.75.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41 for case 33.

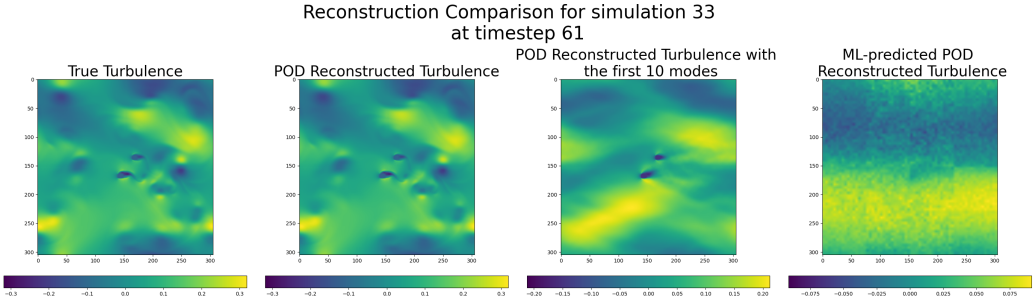

Fig. A.76.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61 for case 33.

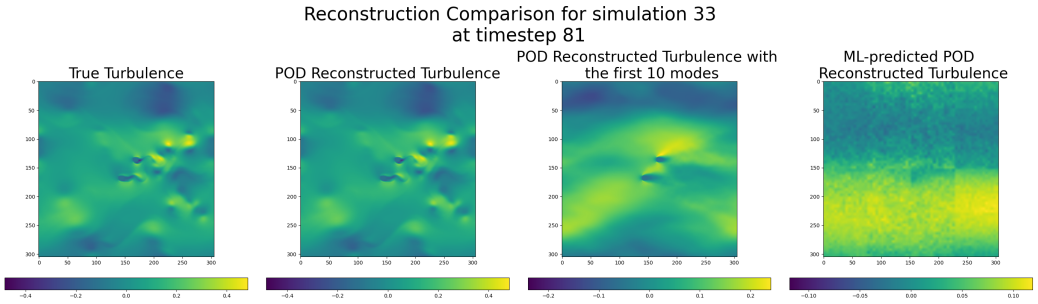

Fig. A.77.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81 for case 33.

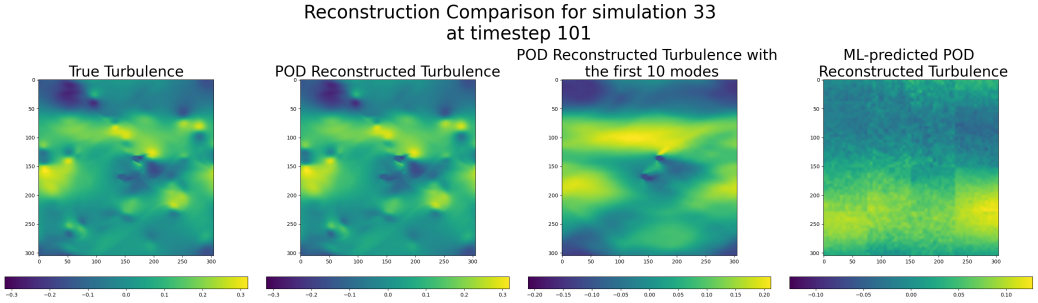

Fig. A.78.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101 for case 33.

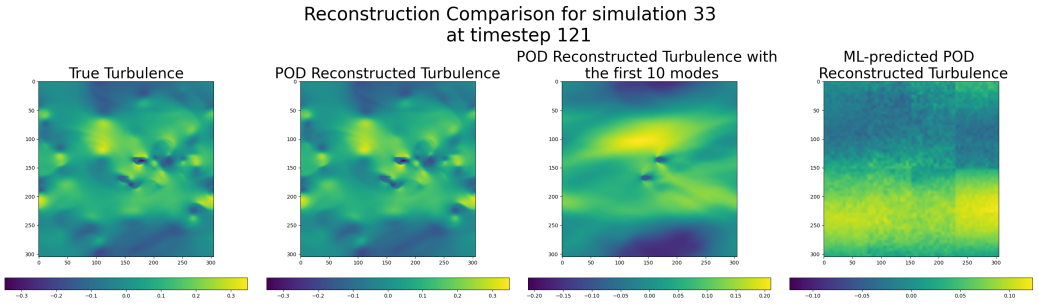

Fig. A.79.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121 for case 33.

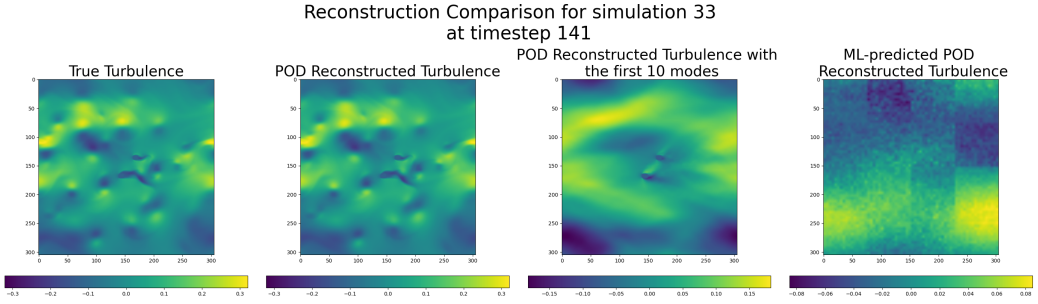

Fig. A.80.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141 for case 33.

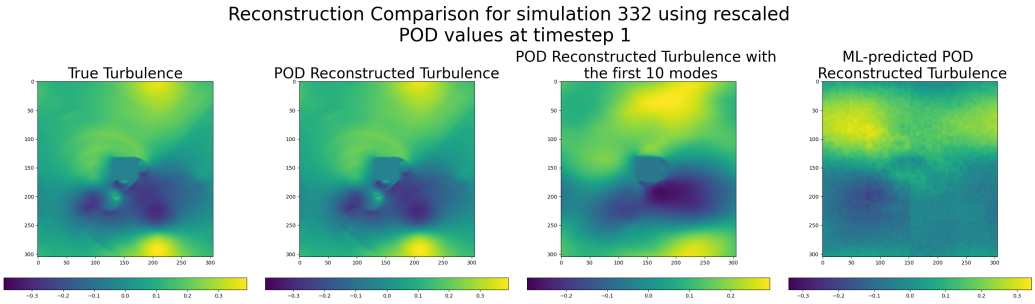

Fig. A.81.: An instantaneous snapshot of the turbulent fluid flow at timestep 1 for array 332. The ML-predicted POD modes are rescaled to match the true POD modes.

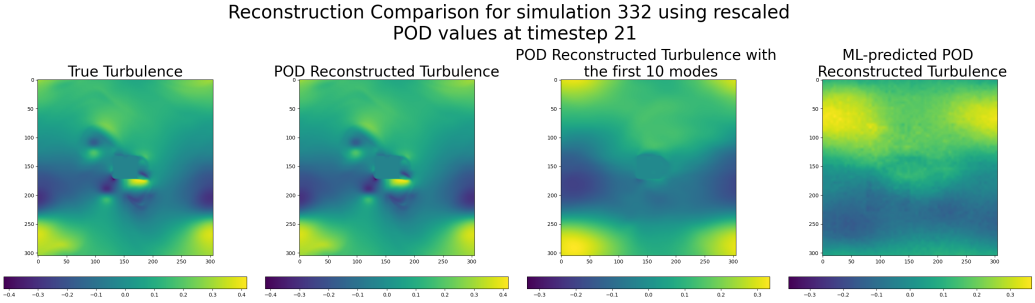

Fig. A.82.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 21.

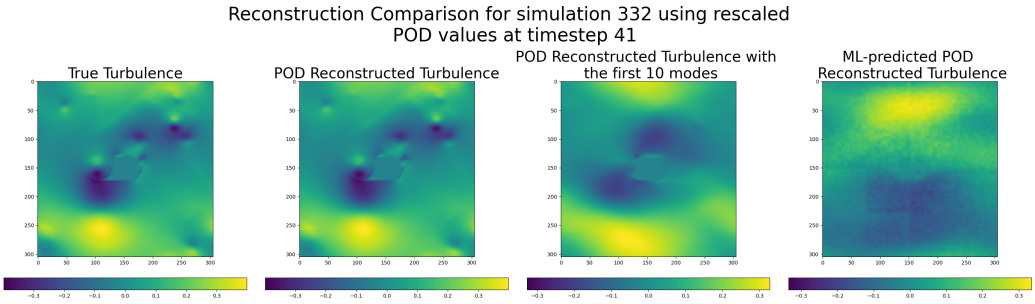

Fig. A.83.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 41.

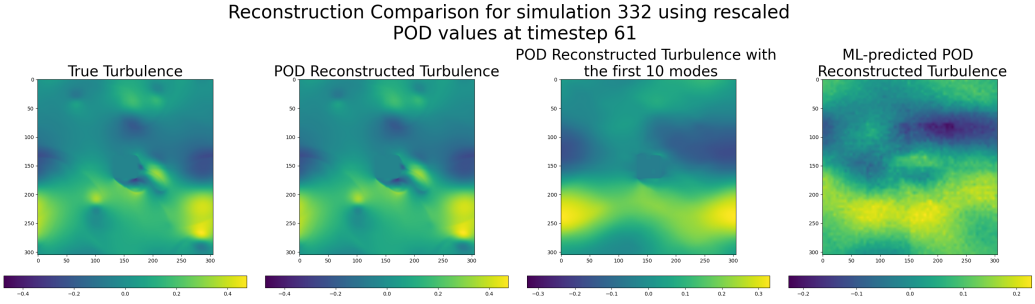

Fig. A.84.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 61.

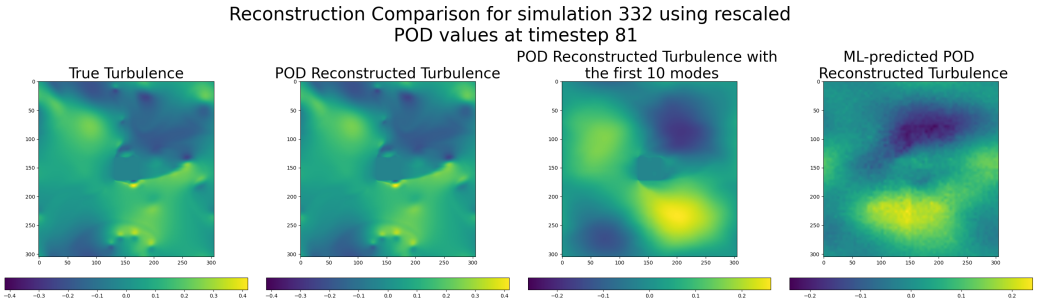

Fig. A.85.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 81.

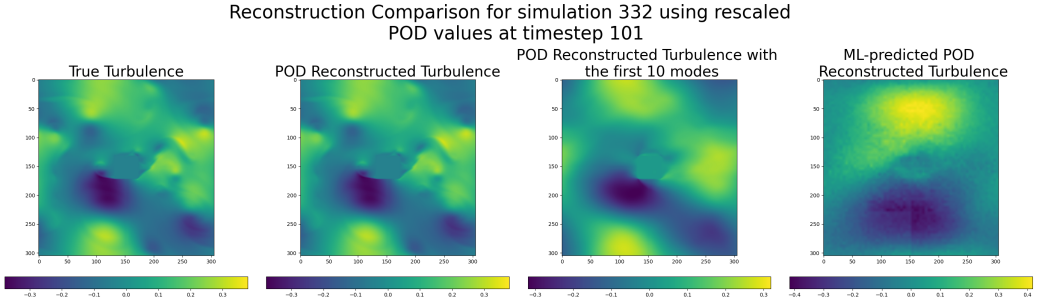

Fig. A.86.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 101.

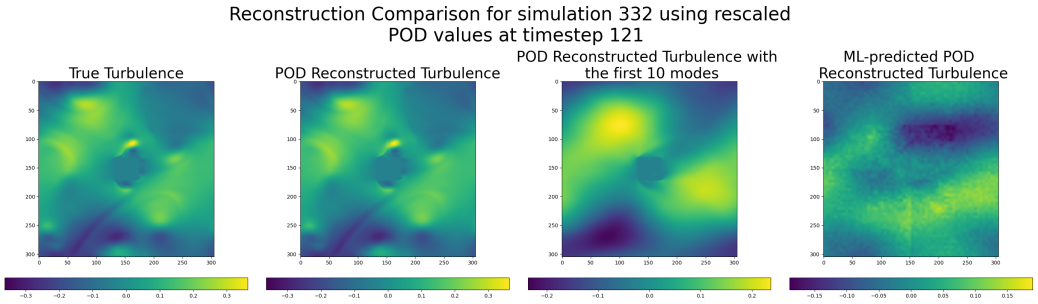

Fig. A.87.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 121.

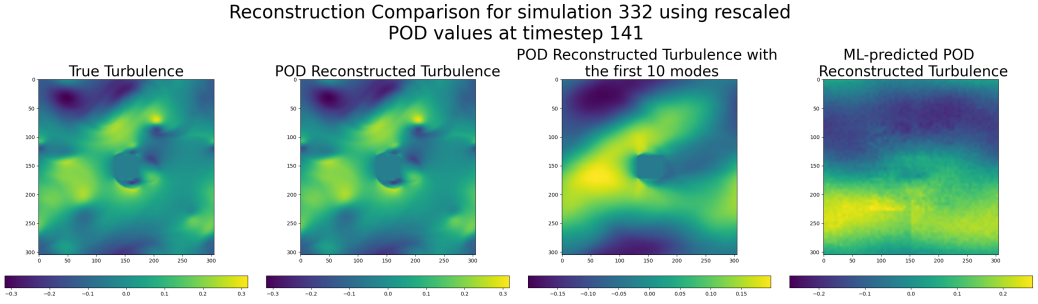

Fig. A.88.: An instantaneous snapshot of the reconstructed turbulent fluid flows at timestep 141.

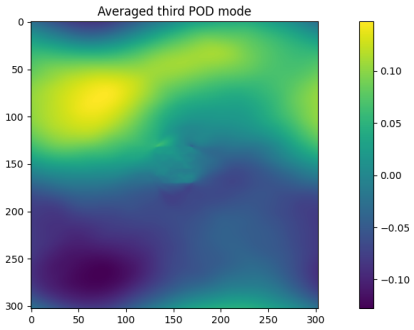

Fig. A.89.: The averaged POD mode across the third POD mode.

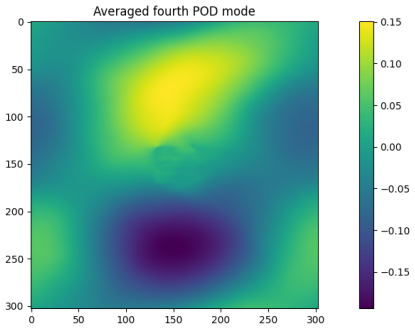

Fig. A.90.: The averaged POD mode across the fourth POD mode.

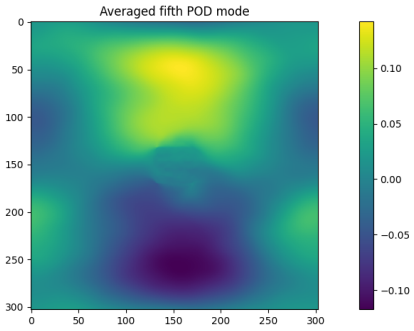

Fig. A.91.: The averaged POD mode across the fifth POD mode.

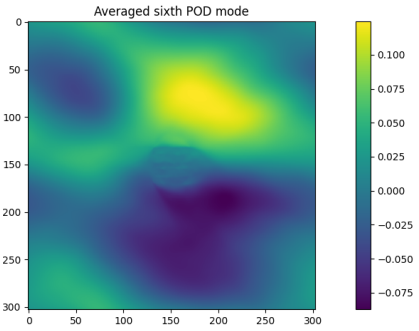

Fig. A.92.: The averaged POD mode across the sixth POD mode.

A.8 Averaged POD modes

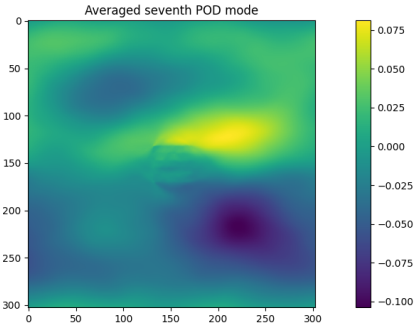

Fig. A.93.: The averaged POD mode across the seventh POD mode.

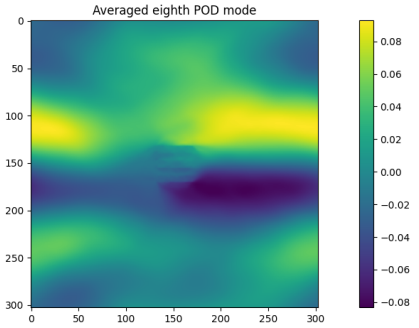

Fig. A.94.: The averaged POD mode across the eighth POD mode.

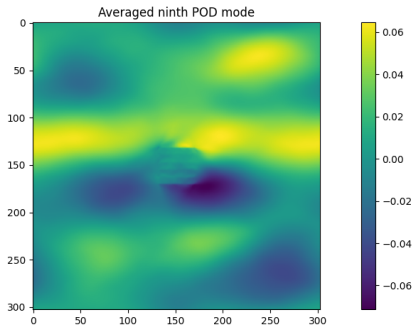

Fig. A.95.: The averaged POD mode across the ninth POD mode.

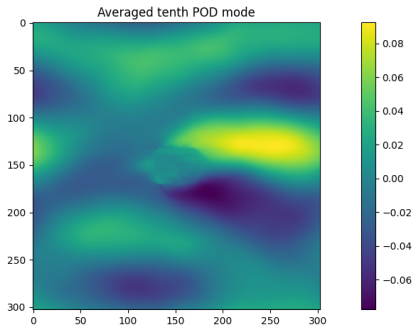

Fig. A.96.: The averaged POD mode across the tenth POD mode.

A.9 SSIM

A.9.1 SSIM comparison figures

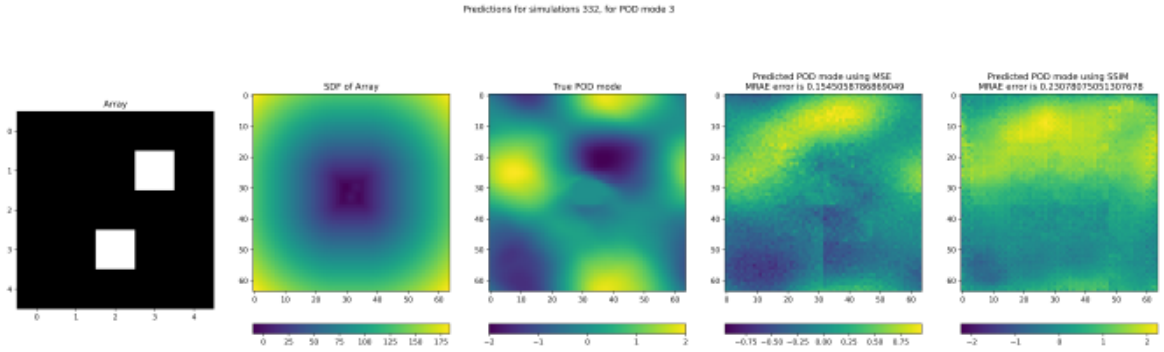

Fig. A.97.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the third POD mode.

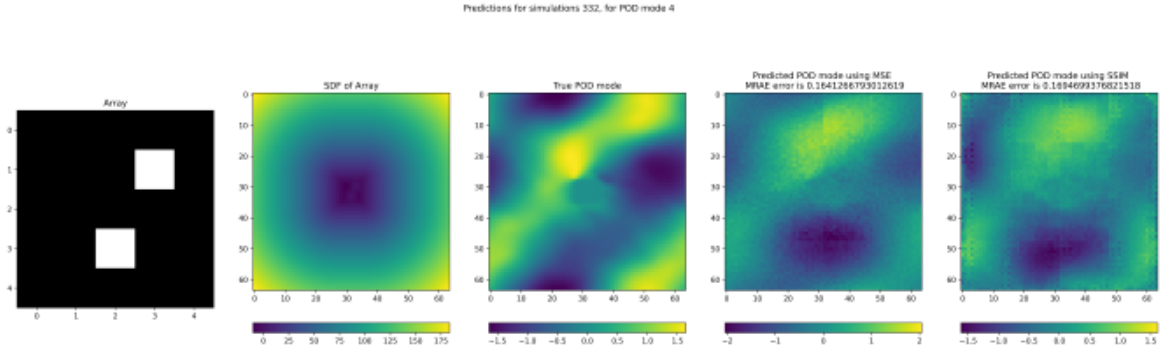

Fig. A.98.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the fourth POD mode.

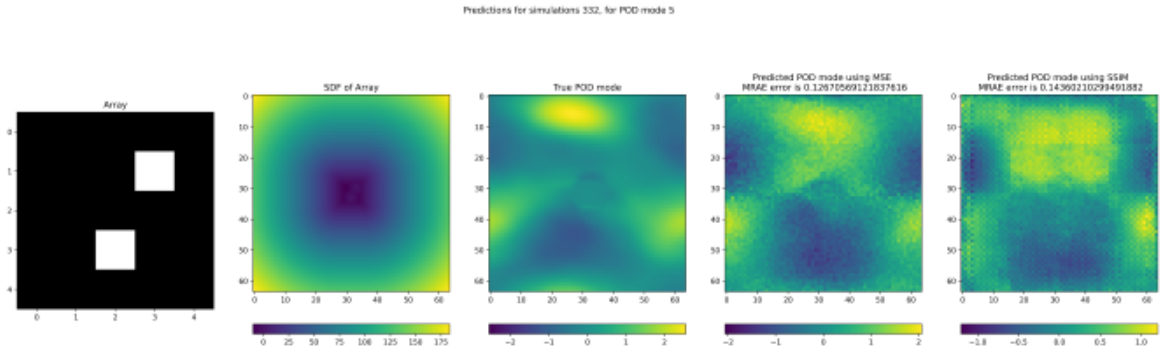

Fig. A.99.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the fifth POD mode.

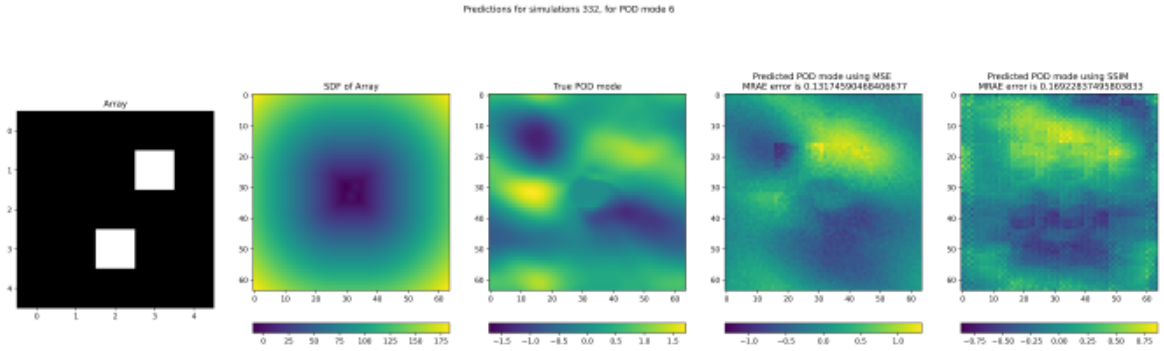

Fig. A.100.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the sixth POD mode.

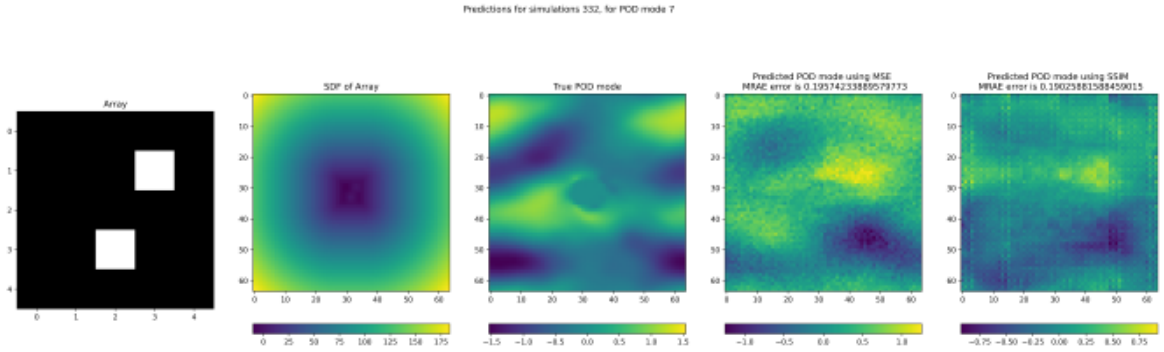

Fig. A.101.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the seventh POD mode.

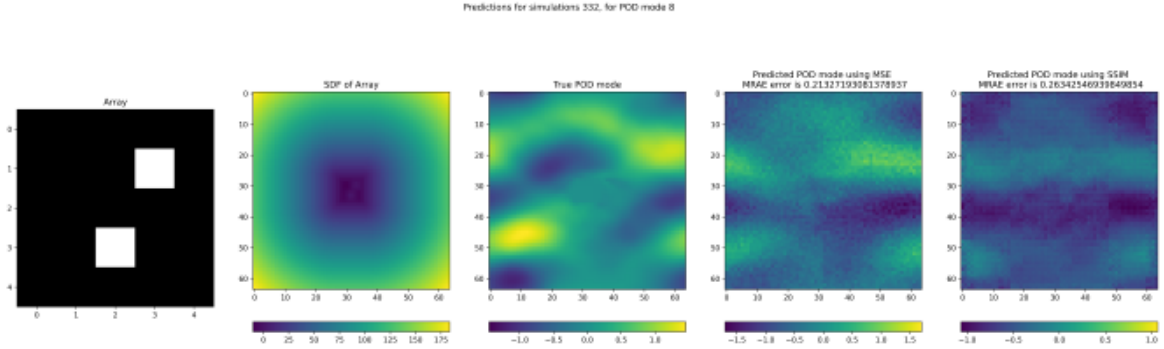

Fig. A.102.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the eighth POD mode.

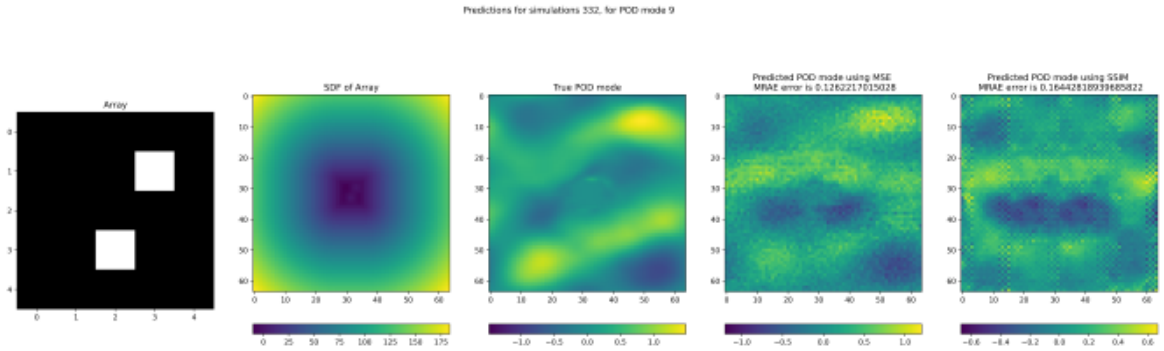

Fig. A.103.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the ninth POD mode.

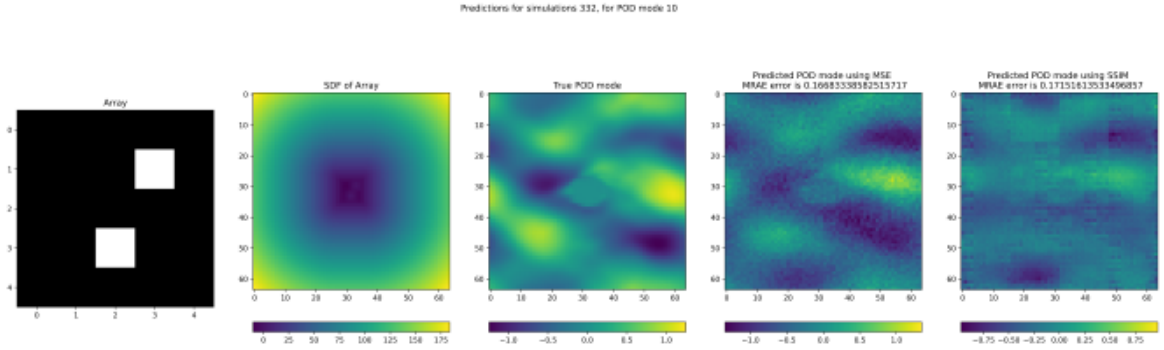

Fig. A.104.: This figure shows the comparison between the MSE loss function and the SSIM loss function for the tenth POD mode.

A.9.2 SSIM histogram error

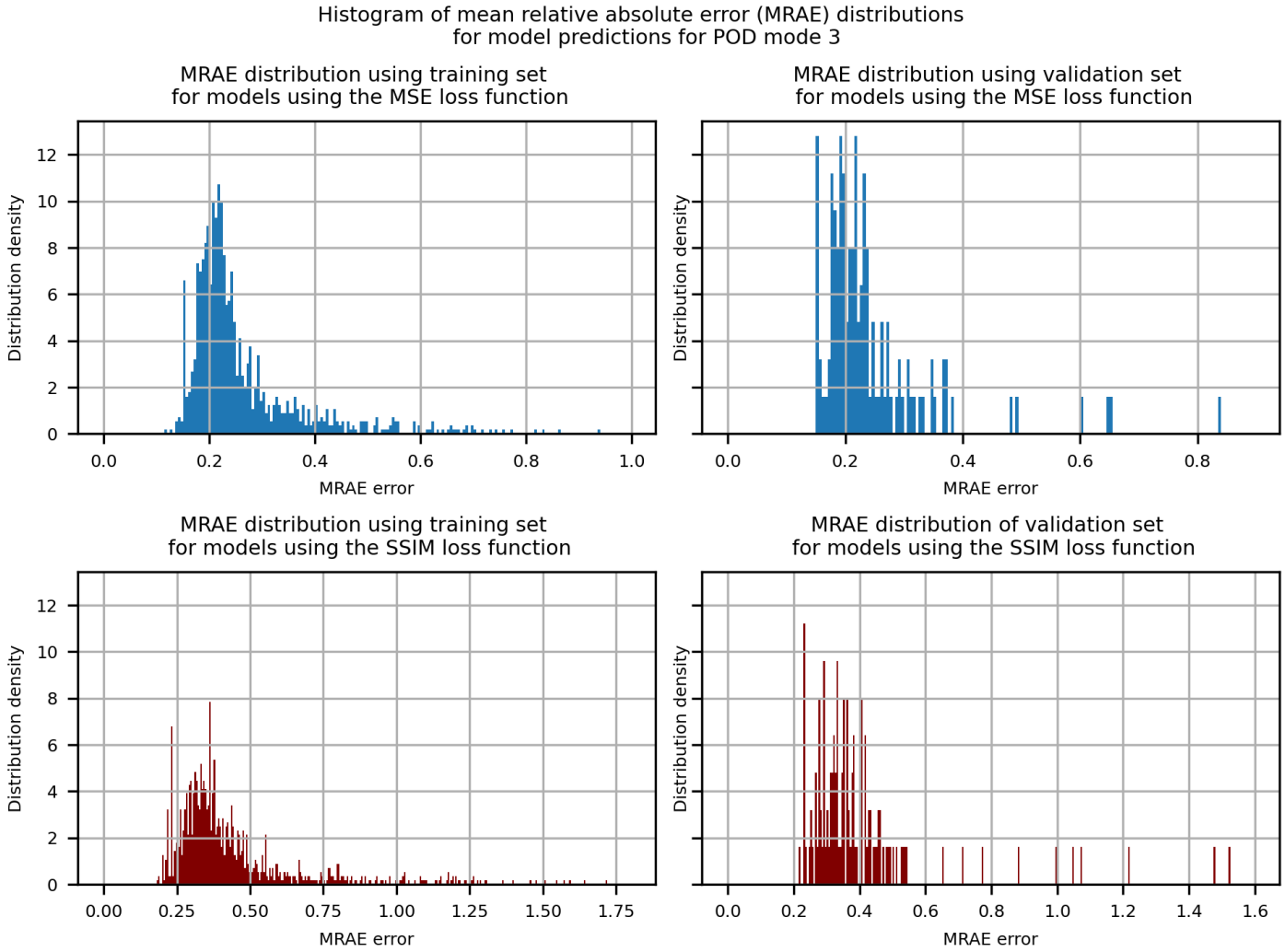

Fig. A.105.: The normalised histogram for the third POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

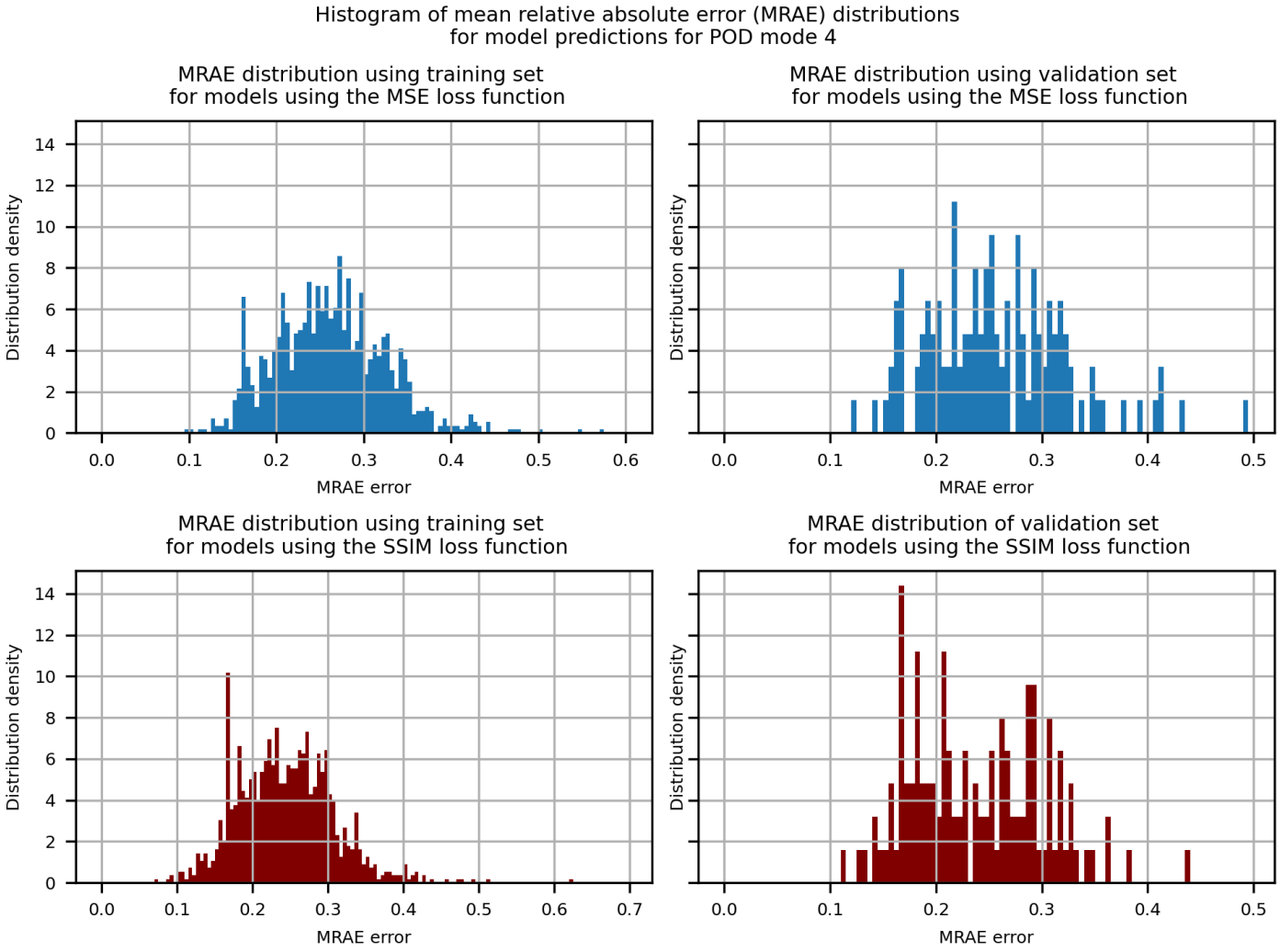

Fig. A.106.: The normalised histogram for the fourth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

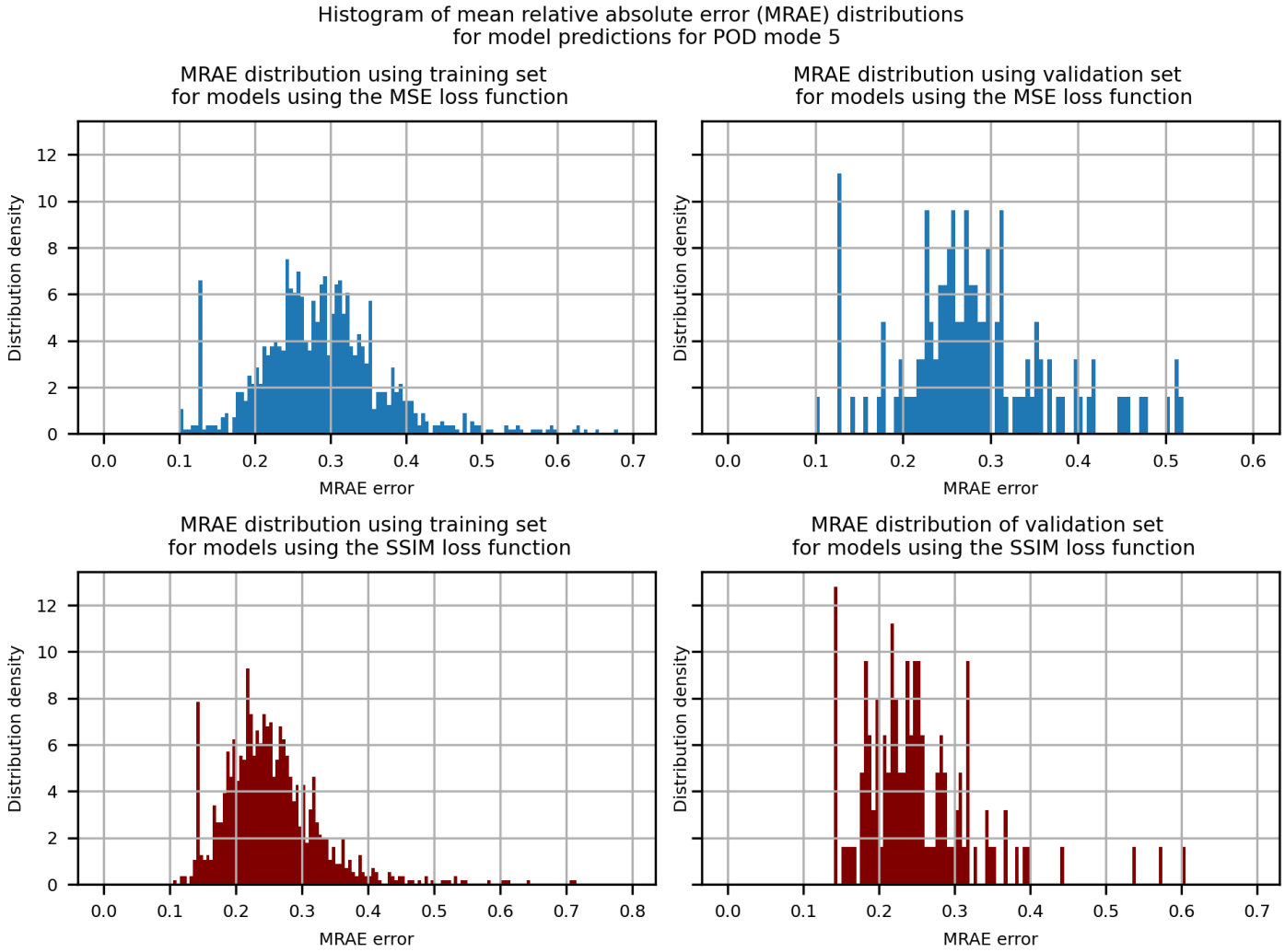

Fig. A.107.: The normalised histogram for the fifth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

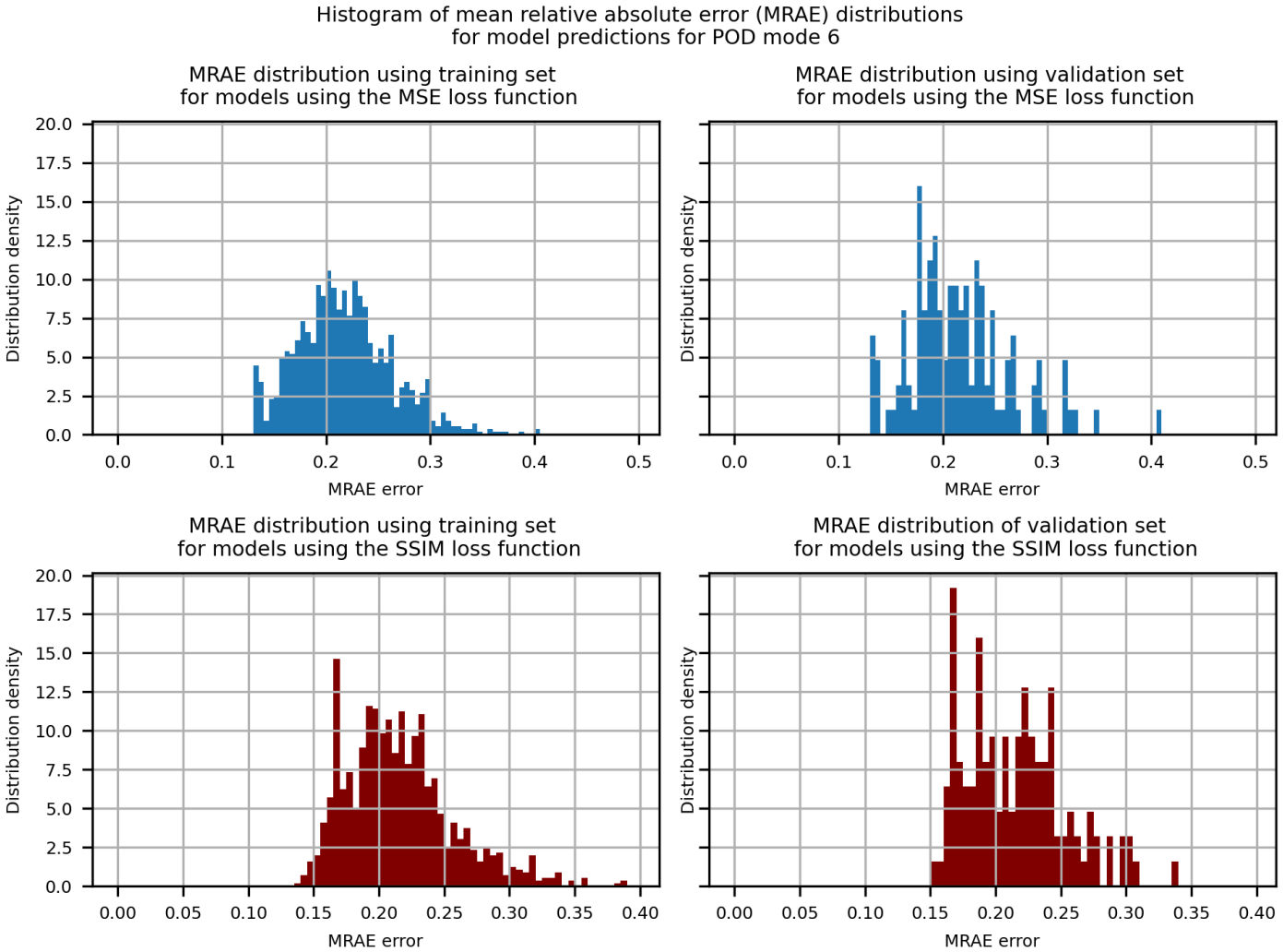

Fig. A.108.: The normalised histogram for the sixth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

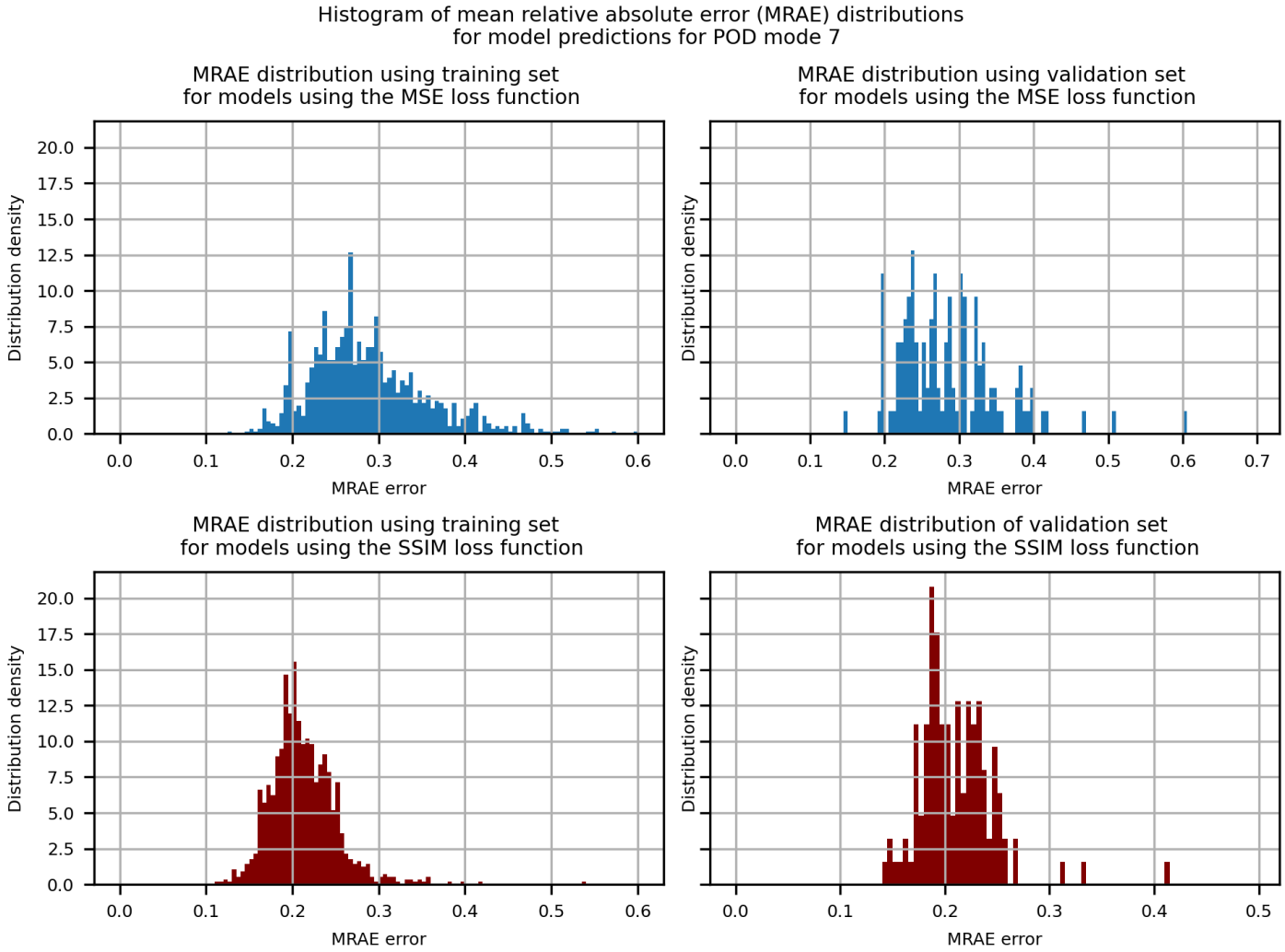

Fig. A.109.: The normalised histogram for the seventh POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

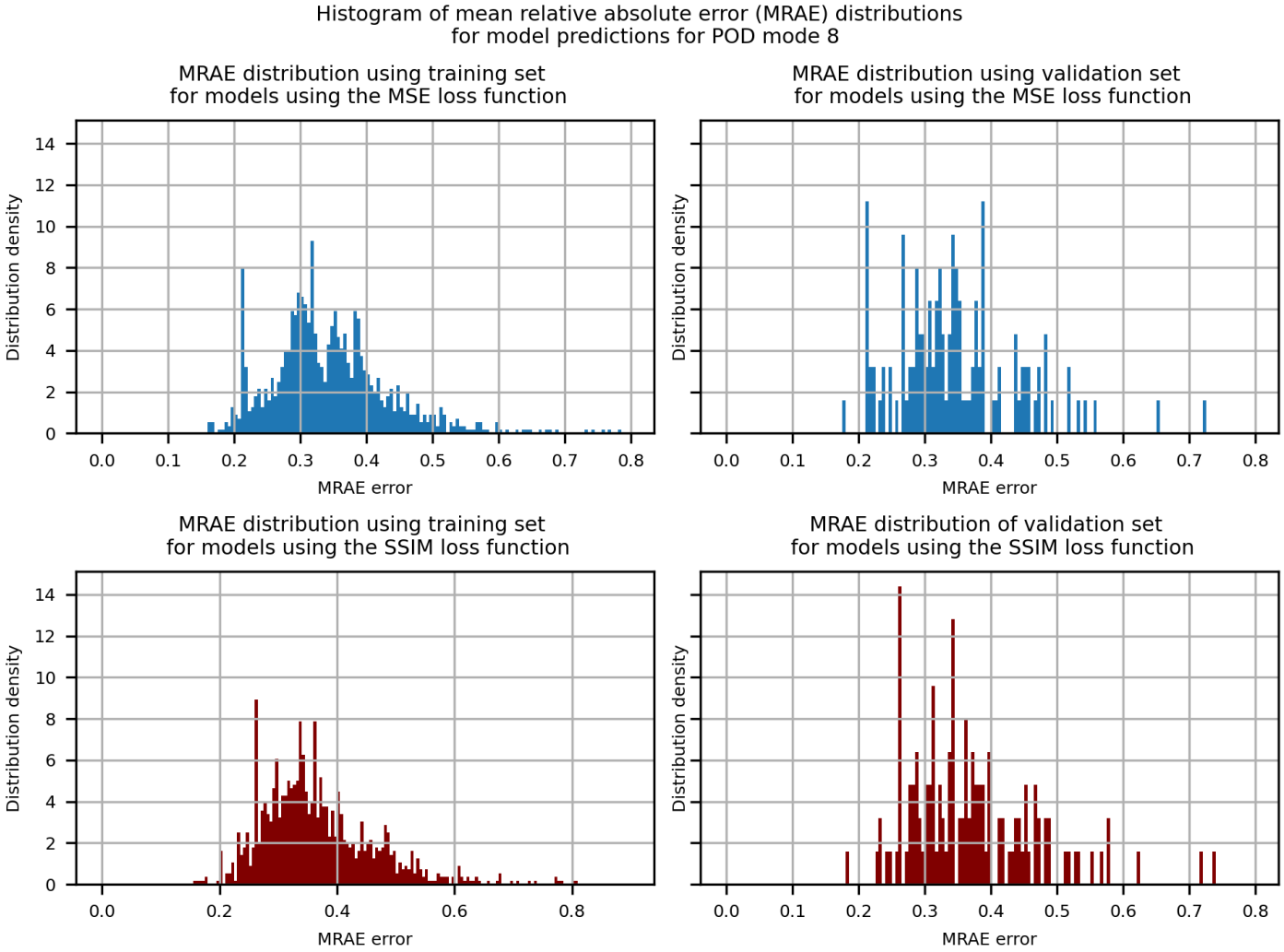

Fig. A.110.: The normalised histogram for the eighth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

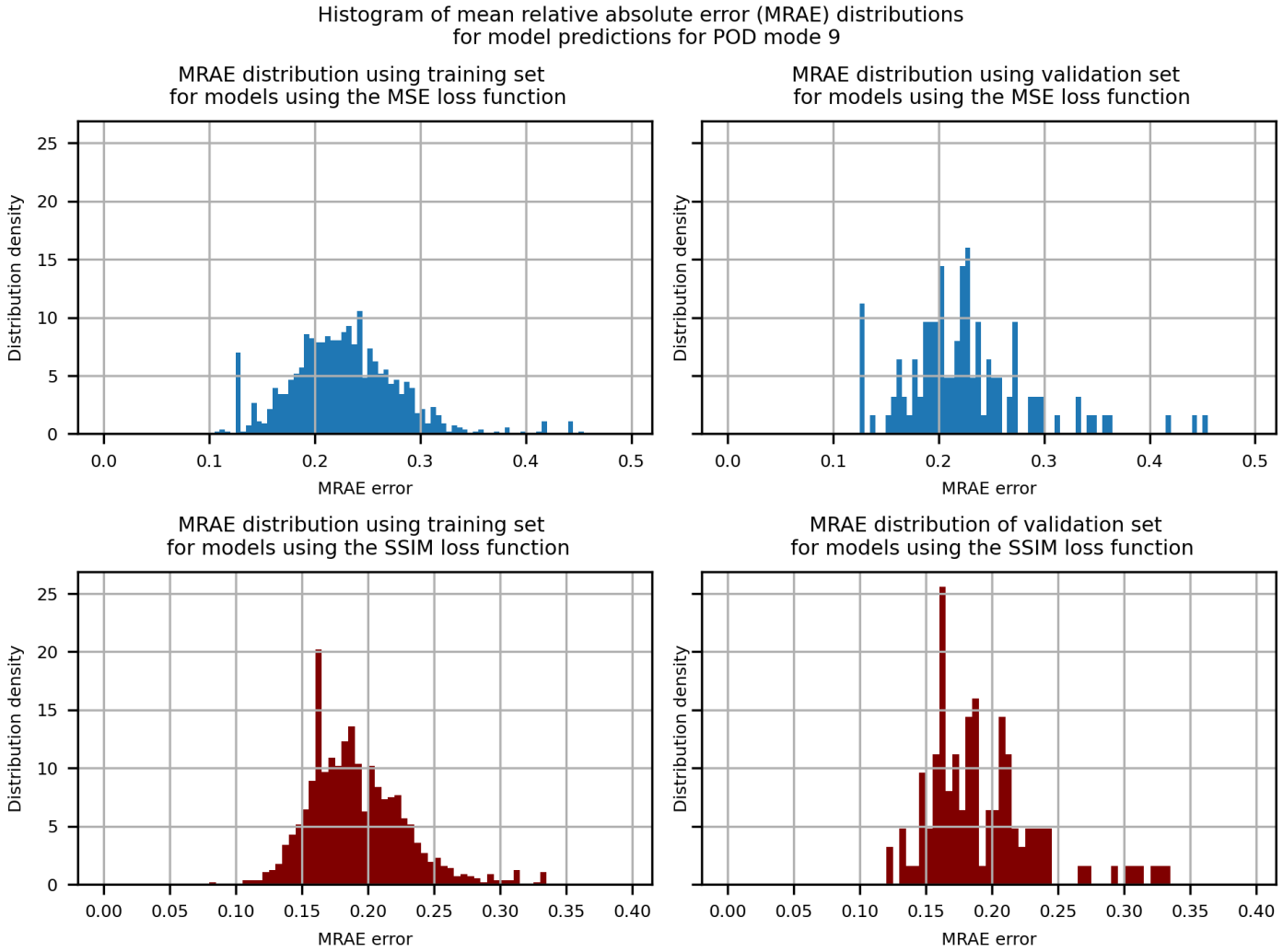

Fig. A.111.: The normalised histogram for the ninth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

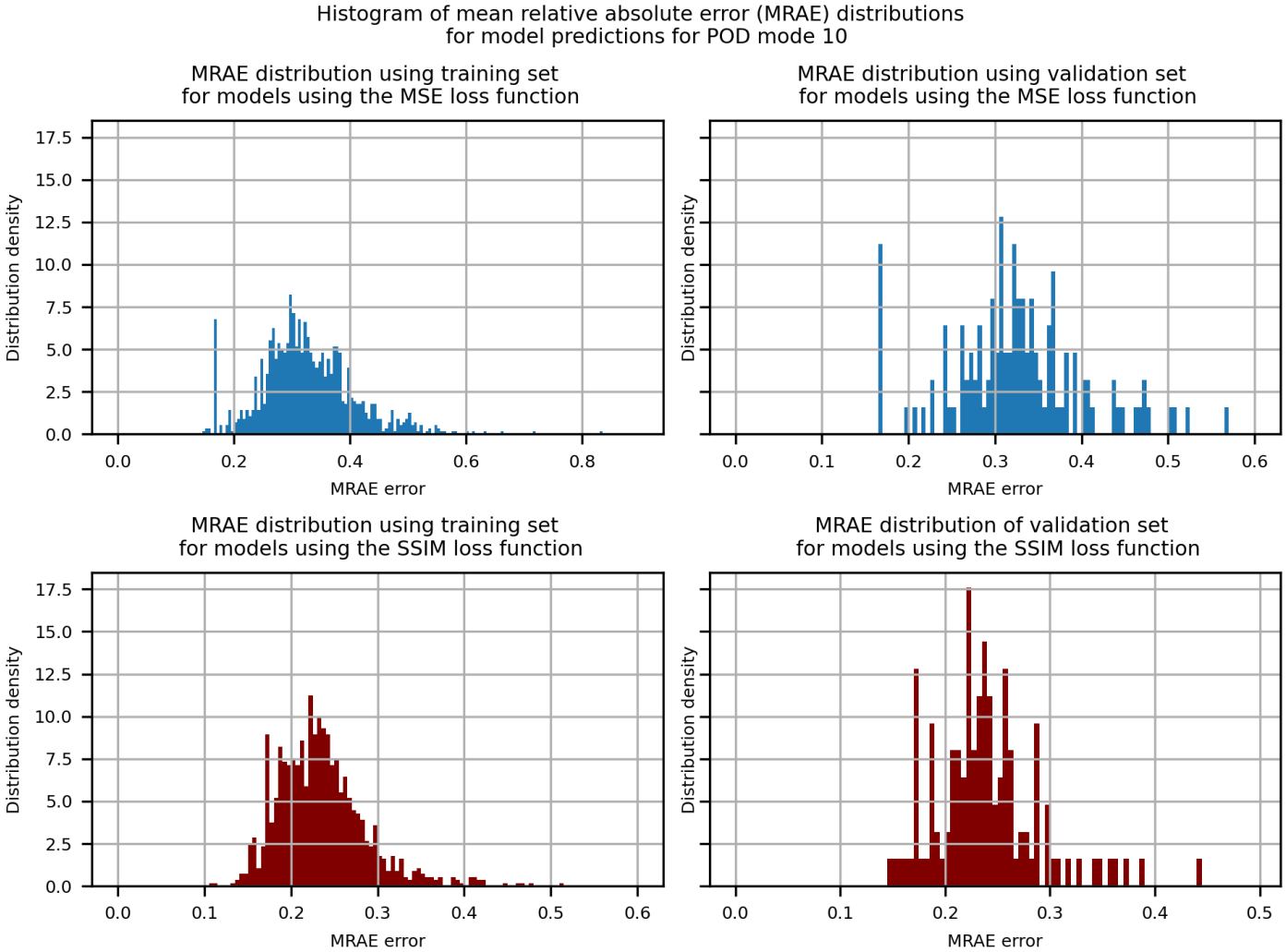

Fig. A.112.: The normalised histogram for the tenth POD mode showing the distribution of MRAE error for the CNN model using the MSE loss function and the SSIM function. We show the distribution of MRAE errors in the training set and validation set.

Bibliography

[1] Maryam Asghari Mooneghi and Ramtin Kargarmoakhar. “Aerodynamic Mitigation and Shape Optimization of Buildings: Review”. In: *Journal of Building Engineering* 6 (2016), pp. 225–235 (cit. on p. 1).

[2] Peter Bailey, Joe Myre, Stuart D.C. Walsh, David J. Lilja, and Martin O. Saar. “Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors”. In: *2009 International Conference on Parallel Processing*. 2009, pp. 550–557 (cit. on p. 41).

[3] Pierre Baldi and Kurt Hornik. “Neural networks and principal component analysis: Learning from examples without local minima”. In: *Neural Networks* 2.1 (1989), pp. 53–58 (cit. on p. 9).

[4] Richard Bellman and Robert Kalaba. “A mathematical theory of adaptive control processes”. In: *Proceedings of the National Academy of Sciences* 45.8 (1959), pp. 1288–1290. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.45.8.1288 (cit. on p. 12).

[5] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305 (cit. on pp. 27, 28).

[6] P. L. Bhatnagar, E. P. Gross, and M. Krook. “A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems”. In: *Phys. Rev.* 94 (3 1954), pp. 511–525 (cit. on p. 33).

[7] Saakaar Bhatnagar, Yaser Afshar, Shaowu Pan, Karthik Duraisamy, and Shailendra Kaushik. “Prediction of aerodynamic flow fields using convolutional neural networks”. In: *Computational Mechanics* 64.2 (2019), pp. 525–545. arXiv: 1905.13166 (cit. on p. 14).

[8] M’hamed Bouzidi, Mouaouia Firdaouss, and Pierre Lallemand. “Momentum transfer of a Boltzmann-lattice fluid with boundaries”. In: *Physics of Fluids* 13.11 (2001), pp. 3452–3459. eprint: https://doi.org/10.1063/1.1399290 (cit. on p. 39).

[9] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. “Machine Learning for Fluid Mechanics”. In: *Annual Review of Fluid Mechanics* 52 (2020), pp. 477–508. arXiv: 1905.11075 (cit. on pp. 7, 13).

[10] Shiyi Chen and Gary D. Doolen. “LATTICE BOLTZMANN METHOD FOR FLUID FLOWS”. In: *Annual Review of Fluid Mechanics* () (cit. on p. 31).

[11] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314 (cit. on p. 17).

[12] Gregory L. Eyink and Katepalli R. Sreenivasan. “Onsager and the theory of hydrodynamic turbulence”. In: *Rev. Mod. Phys.* 78 (1 2006), pp. 87–135 (cit. on p. 4).

[13] K. J. Falconer. *Fractal geometry : mathematical foundations and applications*. Third edition. 2014 (cit. on p. 5).

[14] R. L. Figueroa, Q. Zeng-Treitler, S. Kandula, et al. “Predicting sample size required for classification performance”. In: *BMC Med Inform Decis Mak 12* 8 (2012) (cit. on p. 28).

[15] Olga Filippova and Dieter Hänel. “Grid Refinement for Lattice-BGK Models”. In: *Journal of Computational Physics* 147.1 (1998), pp. 219–228 (cit. on p. 40).

[16] Uriel Frisch. *Turbulence: The Legacy of A. N. Kolmogorov*. Cambridge University Press, 1995 (cit. on p. 5).

[17] M. B. Giles and I. Reguly. “Trends in high-performance computing for engineering calculations”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2022 (2014), p. 20130319. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2013.0319 (cit. on p. 41).

[18] I. Ginzbourg and P. M. Adler. “Boundary flow condition analysis for the three-dimensional lattice Boltzmann model”. In: *Journal de Physique II* 4.2 (Feb. 1994), pp. 191–214 (cit. on p. 39).

[19] Irina Ginzburg and Dominique d’Humières. “Multireflection boundary conditions for lattice Boltzmann models”. In: *Phys. Rev. E* 68 (6 2003), p. 066614 (cit. on pp. 38, 39).

[20] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256 (cit. on p. 23).

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. 2015 (cit. on pp. 20, 28).

[22] Xiaoxiao Guo, Wei Li, and Francesco Iorio. “Convolutional neural networks for steady flow approximation”. In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 13-17-Augu (2016), pp. 481–490 (cit. on pp. 8, 13, 53, 61, 62, 89).

[23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362 (cit. on p. 59).

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385 (cit. on p. 21).

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Delving deep into rectifiers: Surpassing human-level performance on ImageNet Classification”. In: *2015 IEEE International Conference on Computer Vision (ICCV)* (2015) (cit. on p. 23).

[26] Andreas Heinecke, Jinn Ho, and Wen-Liang Hwang. “Refinement and Universal Approximation via Sparsely Connected ReLU Convolution Nets”. In: *IEEE Signal Processing Letters* 27 (2020), pp. 1175–1179 (cit. on p. 12).

[27] J. E. Higham and A. Vaidheeswaran. “Modification of modal characteristics in the wakes of blockages of square cylinders with multi-scale porosity”. In: *Physics of Fluids* 34.2 (2022), p. 025114. eprint: https://doi.org/10.1063/5.0078437 (cit. on p. 6).

[28] A.K. Jain and B. Chandrasekaran. “39 Dimensionality and sample size considerations in pattern recognition practice”. In: *Classification Pattern Recognition and Reduction of Dimensionality*. Vol. 2. Handbook of Statistics. Elsevier, 1982, pp. 835–855 (cit. on p. 28).

[29] Katarzyna Janocha and Wojciech Marian Czarnecki. “On loss functions for deep neural networks in classification”. In: *Schedae Informaticae* 1/2016 (2017) (cit. on p. 17).

[30] M Juniper, C Noakes, S Tobias, C Savy, and J Lincoln. *Our Fluid Nation: The Impact of Fluid Dynamics in the UK*. Report 10.5518/100/77. 2021 (cit. on p. 1).

[31] M Kahlbacher and S Volkwein. “Model reduction by proper orthogonal decomposition for estimation of scalar parameters in elliptic PDEs”. In: *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006*. Delft University of Technology; European Community on Computational Methods . . . 2006 (cit. on p. 44).

[32] A. Kolmogorov. “The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds’ Numbers”. In: *Akademiia Nauk SSSR Doklady* 30 (Jan. 1941), pp. 301–305 (cit. on p. 4).

[33] Mathias J. Krause, Adrian Kummerländer, Samuel J. Avis, et al. “OpenLB—Open source lattice Boltzmann code”. In: *Computers and Mathematics with Applications* 81 (2021), pp. 258–288 (cit. on p. 42).

[34] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, et al. *The Lattice Boltzmann Method*. 2017. arXiv: arXiv:1011.1669v3 (cit. on pp. 31, 55, 57).

[35] Timm Kruger, Halim Kusumaatmaja, Alexandr Kuzmin, et al. *The lattice boltzmann method, principles and practice*. Vol. 10. 207. 2017, pp. 1–705. arXiv: arXiv:1011.1669v3 (cit. on p. 41).

[36] Adrian Kummerländer, Sam Avis, Halim Kusumaatmaja, et al. *OpenLB User Guide*. English. Version Version 1.5. 187 pp. April 14, 2022 (cit. on p. 42).

[37] J. Nathan Kutz. “Deep learning in fluid dynamics”. In: *Journal of Fluid Mechanics* 814 (2017), pp. 1–4 (cit. on p. 13).

[38] Y. LeCun, B. Boser, J. S. Denker, et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (1989), pp. 541–551 (cit. on p. 20).

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 28).

[40] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural Networks* 6.6 (1993), pp. 861–867 (cit. on p. 17).

[41] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. “Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits”. In: *CoRR* abs/1603.06560 (2016). arXiv: 1603.06560 (cit. on p. 28).

[42] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. “Reynolds averaged turbulence modelling using deep neural networks with embedded invariance”. In: *Journal of Fluid Mechanics* 807 (2016), pp. 155–166 (cit. on p. 8).

[43] J. L. Lumley. “The Structure of Inhomogeneous Turbulent Flows”. In: *Atmospheric Turbulence and Radio Wave Propagation* (1967), pp. 166–177 (cit. on p. 43).

[44] Benoit Mandelbrot. “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension”. In: *Science* 156.3775 (1967), pp. 636–638. eprint: https://www.science.org/doi/pdf/10.1126/science.156.3775.636 (cit. on p. 5).

[45] Jacob H Marlow, Franck CGA Nicolleau, and Wernher Brevis. “Wake Characterisation of 3-Dimensional Multiscale Porous Obstacles”. In: *arXiv preprint arXiv:1907.06075* (2019) (cit. on p. 6).

[46] N. Mazellier and J. C. Vassilicos. “Turbulence without Richardson–Kolmogorov cascade”. In: *Physics of Fluids* 22.7 (2010), p. 075101. eprint: https://doi.org/10.1063/1.3453708 (cit. on p. 6).

[47] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56 –61 (cit. on p. 59).

[48] Mateusz Michalkiewicz, Jhony K Pontes, Dominic Jack, Mahsa Baktashmotlagh, and Anders Eriksson. “Implicit surface representations as layers in neural networks”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 4743–4752 (cit. on p. 64).

[49] Tharindu P. Miyanawala and Rajeev K. Jaiman. *An Efficient Deep Learning Technique for the Navier-Stokes Equations: Application to Unsteady Wake Flow Dynamics*. 2017 (cit. on p. 8).

[50] R.R. Nourgaliev, T.N. Dinh, T.G. Theofanous, and D. Joseph. “The lattice Boltzmann equation method: theoretical interpretation, numerics and implications”. In: *International Journal of Multiphase Flow* 29.1 (2003), pp. 117–169 (cit. on p. 41).

[51] Tom O’Malley, Elie Bursztein, James Long, et al. *Keras Tuner*. https://github.com/keras-team/keras-tuner. 2019 (cit. on p. 62).

[52] L. Onsager. “Statistical hydrodynamics”. In: *Il Nuovo Cimento (1943-1954)* 6.2 (1949), pp. 279–287 (cit. on p. 4).

[53] T. Pohl, F. Deserno, N. Thurey, et al. “Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures”. In: *SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. 2004, pp. 21–21 (cit. on p. 41).

[54] Sarunas J. Raudys and Anil K. Jain. “Small sample size effects in statistical pattern recognition: Recommendations for practitioners”. In: *IEEE Transactions on pattern analysis and machine intelligence 13. 3* 3.1991 () (cit. on p. 28).

[55] Suman Ravuri, Karel Lenc, Matthew Willson, et al. “Skilful precipitation nowcasting using deep generative models of radar”. In: *Nature* 597.7878 (2021), pp. 672–677 (cit. on p. 13).

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536 (cit. on p. 17).

[57] Vincent Sitzmann, Eric Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein. “Metasdf: Meta-learning signed distance functions”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 10136–10147 (cit. on p. 64).

[58] C. Bane Sullivan and Alexander Kaszynski. “PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)”. In: *Journal of Open Source Software* 4.37 (2019), p. 1450 (cit. on p. 59).

[59] Kunihiko Taira, Steven L. Brunton, Scott T.M. Dawson, et al. “Modal analysis of fluid flows: An overview”. In: *AIAA Journal* 55.12 (2017), pp. 4013–4041. arXiv: 1702.01453 (cit. on p. 44).

[60] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020 (cit. on p. 59).

[61] Ricardo Vinuesa and Steven L. Brunton. “Enhancing computational fluid dynamics with machine learning”. In: *Nature Computational Science* 2.6 (2022), pp. 358–366 (cit. on p. 7).

[62] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612 (cit. on p. 81).

[63] Julien Weiss. “A Tutorial on the Proper Orthogonal Decomposition”. In: *AIAA Aviation 2019 Forum*. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2019-3333 (cit. on p. 44).

[64] Qisu Zou and Xiaoyi He. “On pressure and velocity boundary conditions for the lattice Boltzmann BGK model”. In: *Physics of Fluids* 9.6 (1997), pp. 1591–1598. eprint: https://doi.org/10.1063/1.869307 (cit. on p. 39).