Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Μηχανολόγων Μηχανικών
Τομέας Ρευστών
Εργαστήριο Βιορευστομηχανικής & Βιοϊατρικής Τεχνολογίας

ΥΠΟΚΑΤΑΣΤΑΣΤΑ ΜΟΝΤΕΛΑ ΡΟΗΣ ΑΙΜΑΤΟΣ ΣΕ ΑΓΓΕΙΑΚΕΣ ΓΕΩΜΕΤΡΙΕΣ ΜΕ ΧΡΗΣΗ ΤΕΧΝΗΤΩΝ ΝΕΥΡΩΝΙΚΩΝ ΔΙΚΤΥΩΝ ΚΑΘΟΔΗΓΟΥΜΕΝΩΝ ΑΠΟ ΤΗ ΦΥΣΙΚΗ

Διπλωματική Εργασία

του

ΜΙΧΑΗΛ ΑΘΑΝΑΣΙΟΥ

Επιβλέπων: Χρήστος Μανόπουλος, Επίκουρος Καθηγητής

Αθήνα, Σεπτέμβριος 2024

National Technical University of Athens
School of Mechanical Engineering
Fluids Section
Laboratory of Biofluid Mechanics & Biomedical Technology

SURROGATE MODELS OF BLOOD FLOW IN VESSEL GEOMETRIES USING PHYSICS-INFORMED NEURAL NETWORKS

MASTER'S THESIS

by

MICHAIL ATHANASIOY

Supervisor: Christos Manopoulos, Assistant Professor

Athens, September 2024

Acknowledgements

Firstly, I would like to thank my supervisor, Assistant Professor Mr. Christos Manopoulos, Director of Biofluid Mechanics and Biomedical Engineering Laboratory, for the opportunity he gave me to carry out my master's thesis on such an important subject, as well as for the continuous and immediate support he provided and the interest he inspired in me throughout its duration. I would also like to thank Laboratory Teaching Staff Dr. Anastasios Raptis, who, with his technical knowledge and the time he devoted, contributed decisively to the completion of the Thesis. Finally, I would like to thank my parents, who, with their sacrifices and support, helped me to fulfill my academic journey so far, as well as my my friends for accompanying me along the journey creating glorious memories and unforgettable experiences.

Μιχαήλ Αθανασίου
Αθήνα, Σεπτέμβριος 2024

Περίληψη

Η προσομοίωση της ροής ρευστού σε σωληνοειδείς γεωμετρίες είναι μια απαραίτητη συνιστώσα της υπολογιστικής βιοϊατρικής μηχανικής, έχοντας εφαρμογή στη ρευστομηχανική ανάλυση αγγειακών και αναπνευστικών οδών. Ακριβείς αιμοδυναμικοί υπολογισμοί είναι αναγκαίοι για την κατανόηση της σοβαρότητας των ασθενειών, της φυσιολογίας των φαινομένων μεταφοράς και της αιμάτωσης καθώς και των αιτιών που προκαλούνται και εξελίσσονται οι ασθένειες. Συνήθως, η διαδικασία αυτή περιλαμβάνει την εξαγωγή της ανατομικής γεωμετρίας από ιατρικές απεικονίσεις και τη διενέργεια προσομοιώσεων υπολογιστικής ρευστοδυναμικής. Ωστόσο, παρά την αποδοτικότητα της, η διαδικασία αυτή εξακολουθεί να απαιτεί σημαντικό χρόνο υπολογισμού, ο οποίος μπορεί να κυμαίνεται από ώρες έως ημέρες. Επιπλέον, αυτή η διαδικασία επαναλαμβάνεται για γεωμετρίες με ανατομική ομοιότητα, με αποτέλεσμα την περαιτέρω αύξηση του συνολικού υπολογιστικού κόστους. Έτσι, η επιτάχυνση των προσομοιώσεων για την παροχή λύσεων σε πραγματικό χρόνο μπορεί να ενθαρρύνει τη χρήση τους στην κλινική εφαρμογή.

Τα Νευρωνικά Δίκτυα ενημερωμένα από την φυσική (Physics-Informed Neural Networks - PINNs) έχουν πρόσφατα τραβήξει την προσοχή λόγω της ικανότητάς τους να προσεγγίζουν τη συμπεριφορά πολύπλοκων, μη γραμμικών φυσικών συστημάτων. Αυτά τα δίκτυα χρησιμοποιούν τις θεμελιώδεις αρχές της φυσικής και τις συνοριακές συνθήκες ενός συστήματος, επιτρέποντάς τους να προσεγγίζουν λύσεις με χαμηλά σφάλματα. Παρόλα αυτά, το απλό PINN απαιτεί ξεχωριστή εκπαίδευση για κάθε νέο σύστημα, με αποτέλεσμα μια διαδικασία που είναι πιο χρονοβόρα σε σύγκριση με τις συμβατικές ρευστομηχανικές προσομοιώσεις. Για να αντιμετωπιστεί αυτό το ζήτημα, έχει προταθεί μια στρατηγική πολλαπλών περιπτώσεων PINN (multi-case PINN). Αυτή η προσέγγιση περιλαμβάνει την παραμετροποίηση διαφορετικών περιπτώσεων γεωμετρίας και ροής και την προ-εκπαίδευση τους στο PINN, επιτρέποντας την ταχεία παραγωγή λύσεων σε νέες μελέτες. Αυτή η διπλωματική εργασία στοχεύει να αναλύσει και να συγκρίνει διαφορετικές αρχιτεκτονικές δικτύων με σκοπό τη βελτιστοποίηση του multi-case PINN μέσω πειραμάτων που θα διεξαχθούν σε ιδεατές 2D στενώσεις.

Λέξεις Κλειδιά

Αγγειακές Στενώσεις, Αιμοδυναμική Προσομοίωση, Υποκατάστατα μοντέλα, Βαθιά Μάθηση, Νευρωνικά Δίκτυα Ενημερωμένα από τη Φυσική.

Abstract

Simulating fluid dynamics in tube-like structures is an essential component of computational biomedical engineering, having important applications in vascular and airway fluid dynamics. Accurate fluid dynamics computations are necessary to comprehend disease severity, perfusion and transport physiology, and the biomechanical triggers that cause diseases to start and progress. Typically, this process entails extracting anatomical geometry from medical imaging and conducting computational fluid dynamics simulations. However, despite its efficiency, this process still requires a considerable amount of computational time, which can range from hours to days. Additionally, this procedure is repeated for anatomically similar geometries, resulting in an increase in the overall computational cost. Accelerating fluid dynamics simulations to provide real-time solutions may encourage clinical usage and lead to advancements in disease assessment and decision-making.

Physics-informed neural networks (PINNs) have recently attracted attention for their ability to approximate the behavior of intricate, non-linear physical systems. These networks utilize the fundamental principles of physics and the governing equations of a system, enabling them to approximate solutions with low errors. Nevertheless, single case vanilla PINN requires separate training for every new case, resulting in a more time-intensive process compared to conventional fluid dynamics simulations. In order to tackle this issue, a multi-case PINN strategy has been suggested. This approach involves parameterizing different geometry and flow cases and pre-training them on the PINN. This enables rapid production of flow solutions in new case studies.This thesis aims to analyze and compare different network architectures so as to optimize the multi-case PINN through experiments conducted on idealized 2D stenotic tubes.

Keywords

Vessel Geometries, Hemodynamic Simulation, Surrogate models, Deep Learning, Physics Informed Neural Networks

Contents

Acknowledgements 1

Περίληψη 2

Abstract 3

Abbreviations-Acronyms 13

Nomenclature 14

I Theoretical part 15

1 Basics of neural networks 16
- 1.1 Introduction to neural networks . . . 16
- 1.2 Fundamentals of activation functions . . . 18
- 1.3 Common activation functions . . . 19
- 1.4 Loss functions . . . 24
- 1.5 First order optimization algorithms . . . 25
- 1.6 Second order optimization algorithms . . . 27
- 1.7 Adaptive activation functions . . . 28

2 Physics informed neural networks 29
- 2.1 Introduction to Physics informed neural networks . . . 29
- 2.2 PINNs for solving PDEs . . . 29
- 2.3 Automatic differentiation . . . 31
- 2.4 Error analysis in PINNs . . . 32
- 2.5 Comparison between PINNs and FEM . . . 33

3 Nvidia Modulus Sym 34
- 3.1 Introduction to Nvidia Modulus Sym . . . 34
- 3.2 PINNs in Nvidia Modulus Sym . . . 34
- 3.3 Nvidia Modulus Sym building blocks . . . 34
- 3.4 Modulus Sym workflow . . . 36

4 PINNs in Hemodynamics 38

II Methods 44

5 Hemodynamic predictions in 2D vessel stenoses using PINNs 45

5.1 Problem definition . . . 45
5.2 Network architecture . . . 47
5.3 Tube specifc parameters . . . 48
5.4 Computational Fluid Dynamics simulations in COMSOL . . . 49
5.5 Evaluation of errors . . . 53

III Results 54

6 Study of different parameters 55

6.1 Advantages of tube specifc parameters . . . 55
6.2 Activation functions . . . 62
6.3 Number of layers and number of neurons per layer . . . 65
6.4 Optimizers . . . 70
6.5 Adaptive activation functions . . . 73
6.6 Number of sampling points . . . 85
6.7 Single-case PINN vs multi-case PINN . . . 87

7 FEM vs. multi-case PINN 91

7.1 Introduction . . . 91
7.2 Case study 1: Re=500, fc =0.1 . . . 91
7.3 Case study 2: Re=500, fc =0.2 . . . 93
7.4 Case study 3: Re=500, fc =0.3 . . . 95
7.5 Case study 4: Re=750, fc =0.1 . . . 97
7.6 Case study 5: Re=750, fc =0.2 . . . 99
7.7 Case study 6: Re=750, fc =0.3 . . . 101
7.8 Case study 7: Re=1000, fc =0.1 . . . 103
7.9 Case study 8: Re=1000, fc =0.2 . . . 105
7.10 Case study 9: Re=1000, fc =0.3 . . . 107
7.11 Case study 10: Re=1250, fc =0.1 . . . 109
7.12 Case study 11: Re=1250, fc =0.2 . . . 111
7.13 Case study 12: Re=1250, fc =0.3 . . . 113
7.14 Case study 13: Re=1500, fc =0.1 . . . 115
7.15 Case study 14: Re=1500, fc =0.2 . . . 117
7.16 Case study 15: Re=1500, fc =0.3 . . . 119
7.17 Case study 16: Re=1750, fc =0.1 . . . 121
7.18 Case study 17: Re=1750, fc =0.2 . . . 123
7.19 Case study 18: Re=1750, fc =0.3 . . . 125
7.20 Errors PINNs vs FEM . . . 127

A Python code 133

Bibliography **150**

List of Figures

1.1 Comparison between a biological neuron and an artificial neuron [1]. . . . 16
1.2 Example of a neural network with 3 hidden layers [1]. 17
1.3 A neuron taking an input feature of dimension $d = 3$. 18
1.4 Plot of Sigmoid activation function. 19
1.5 Plot of tanh activation function. 20
1.6 Plot of ReLU activation function. 21
1.7 Plot of Leaky ReLU activation function. 22
1.8 Plot of ELU activation function. 23
1.9 Plot of SiLU activation function. 24

2.1 Schematic of a PINN solving the diffusion equation [2]. 30
2.2 Illustration of errors of a PINN. 33

3.1 A typical workflow in Modulus Sym [3]. 37
3.2 Modulus Sym training algorithm [3]. 37

4.1 2D channel flow over an obstacle [4]: A representative snapshot of the input data on the concentration field within the training domain is plotted in the top left panel alongside the prediction of our algorithm. The algorithm is capable of accurately reconstructing the velocity and the pressure fields without having access to any observations of these fields (shown in the second and third rows). Furthermore, no boundary conditions are specified on the boundaries of the training domain including the physical wall boundaries. 38
4.2 A 3D intracranial aneurysm [4]: Contours of the exact fields and model predictions are plotted on two perpendicular planes for concentration c, velocity u, v, w, and pressure p fields in each row. 39
4.3 Pre-processed 1000 fps HSA acquisition (left), its corresponding binary mask used to define the vessel lumen (middle), and the Sobel-filtered binary mask, used to define the walls of the model (right) [5]. 39
4.4 Spatial sampling at one time step showing vessel lumen in blue, and vessel walls in green (left). Spatial sampling is then extended temporally for both the vessel lumen and vessel wall (vessel wall omitted) (right) [5]. 40
4.5 n vitro HSA image sequence (left four images) compared to the normalized magnitude image (right) generated by the image-assisted PINN. Red arrows show progression of contrast media edges [5]. 40

4.6 Feed-forward neural network architecture used for the transient problem. After physics-informed training, the output of the network is used for reconstruct the 3D velocity field and the Windkessel parameters of all the domain outlets [6]. . . . 41
4.7 Velocity streamlines of the transient problem for the reference solution (a and d), the mean estimated velocity estimated by the PINNs (b and e), and the velocity obtained after a FEM simulation using the found parameters (c and f) at two time instants: t = 0.12 s, which correspond to peak systole, and at t = 0.45 s, during mid diastole [6]. . . . 41
4.8 Mean absolute relative error of the parameter estimation using the Kalman filter approach (ROUKF) and PINNs [6]. . . . 42
4.9 Visual comparison of hemodynamic calculation results between CFD simulation and deep learning method [7]. . . . 42

5.1 Example of 2 stenotic geometries, with (a) $fc = 0.1$ and (b) $fc = 0.3$. . . . 46
5.2 The FCNN of our problem with 4 hidden layers and 16 nodes per hidden layer. . . . 48
5.3 Example of a parametric curve used to simulate the blood flow in a stenosis with fc=0.17. . . . 49
5.4 Definition of fluid properties. . . . 50
5.5 Inlet parabolic velocity profile with a maximum value of 0.46 m/s. . . . 50
5.6 Zero pressure condition at the outlet. . . . 51
5.7 No slip conditions at the walls. . . . 51
5.8 Statistics of an extra fine physics-controlled mesh. . . . 52
5.9 Extraction of COMSOL data. . . . 52

6.1 Total loss for the 2 networks. . . . 55
6.2 Time per step for the 2 networks. . . . 56
6.3 Prediction of u velocity in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 57
6.4 Prediction of v velocity in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 57
6.5 Prediction of pressure in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 58
6.6 Prediction of u velocity in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 58
6.7 Prediction of v velocity in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 59
6.8 Prediction of pressure in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 59
6.9 Prediction of u velocity in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. . . . 60

6.10 Prediction of v velocity in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. 61
6.11 Prediction of pressure in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL. 61
6.12 Total loss for different activation functions. 63
6.13 Continuity loss for different activation functions. 63
6.14 x-momentum loss for different activation functions. 64
6.15 y-momentum loss for different activation functions. 64
6.16 Total loss for different numbers of neuron per layer. 65
6.17 Total loss for different numbers of neuron per layer with respect to wall clock time. 66
6.18 Total loss for different number of layers. 66
6.19 Total loss for different number of layers with respect to wall clock time. . . 67
6.20 Total loss for the different architectures. 67
6.21 Total loss for the different architectures with respect to wall clock time. . . 68
6.22 Continuity loss for the different architectures. 68
6.23 x-momentum loss for the different architectures. 69
6.24 y-momentum loss for the different architectures. 69
6.25 Total loss for different optimizers. 70
6.26 Time per step for different optimizers. 71
6.27 Continuity loss for different optimizers. 71
6.28 x-momentum loss for different optimizers. 72
6.29 y-momentum loss for different optimizers. 72
6.30 Influence of adaptive activation functions. 73
6.31 Influence of adaptive activation functions on time per step. 74
6.32 Prediction of u velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 75
6.33 Prediction of v velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 76
6.34 Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 76
6.35 Prediction of u velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 77
6.36 Prediction of v velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 77
6.37 Prediction of pressure in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 78
6.38 Prediction of u velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 78
6.39 Prediction of v velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 79
6.40 Prediction of pressure in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 79

6.41 Prediction of u velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 80
6.42 Prediction of v velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.. . . . 81
6.43 Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 81
6.44 Prediction of u velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 82
6.45 Prediction of v velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 82
6.46 Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 83
6.47 Prediction of u velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 83
6.48 Prediction of v velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 84
6.49 Prediction of pressure in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL. 84
6.50 Total loss for the 2 Samplings. 86
6.51 Time per step for the 2 Samplings. 86
6.52 Total loss for the 2 networks. 87
6.53 Time per step for the 2 networks. 88
6.54 Prediction of u velocity using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL. 88
6.55 Prediction of v velocity using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL. 89
6.56 Prediction of pressure using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL. 89

7.1 u velocity in multi-case PINN vs FEM for Re=500 and fc=0.1. 91
7.2 v velocity in multi-case PINN vs FEM for Re=500 and fc=0.3. 92
7.3 pressure in multi-case PINN vs FEM for Re=500 and fc=0.3. 92
7.4 u velocity in multi-case PINN vs FEM for Re=500 and fc=0.2. 93
7.5 v velocity in multi-case PINN vs FEM for Re=500 and fc=0.2. 93
7.6 pressure in multi-case PINN vs FEM for Re=500 and fc=0.2. 94
7.7 u velocity in multi-case PINN vs FEM for Re=500 and fc=0.3. 95
7.8 v velocity in multi-case PINN vs FEM for Re=500 and fc=0.3. 95
7.9 pressure in multi-case PINN vs FEM for Re=500 and fc=0.3. 96
7.10 u velocity in multi-case PINN vs FEM for Re=750 and fc=0.1. 97
7.11 v velocity in multi-case PINN vs FEM for Re=750 and fc=0.1. 97
7.12 pressure in multi-case PINN vs FEM for Re=750 and fc=0.1. 98
7.13 u velocity in multi-case PINN vs FEM for Re=750 and fc=0.2. 99
7.14 v velocity in multi-case PINN vs FEM for Re=750 and fc=0.2. 99

7.15 pressure in multi-case PINN vs FEM for Re=750 and fc=0.2. 100
7.16 u velocity in multi-case PINN vs FEM for Re=750 and fc=0.3. 101
7.17 v velocity in multi-case PINN vs FEM for Re=750 and fc=0.3. 101
7.18 pressure in multi-case PINN vs FEM for Re=750 and fc=0.3. 102
7.19 u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.1. 103
7.20 v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.1. 103
7.21 pressure in multi-case PINN vs FEM for Re=1000 and fc=0.1. 104
7.22 u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.2. 105
7.23 v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.2. 105
7.24 pressure in multi-case PINN vs FEM for Re=1000 and fc=0.2. 106
7.25 u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.3. 107
7.26 v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.3. 107
7.27 pressure in multi-case PINN vs FEM for Re=1000 and fc=0.3. 108
7.28 u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.1. 109
7.29 v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.1. 109
7.30 pressure in multi-case PINN vs FEM for Re=1250 and fc=0.1. 110
7.31 u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.2. 111
7.32 v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.2. 111
7.33 pressure in multi-case PINN vs FEM for Re=1250 and fc=0.2. 112
7.34 u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.3. 113
7.35 v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.3. 113
7.36 pressure in multi-case PINN vs FEM for Re=1250 and fc=0.3. 114
7.37 u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.1. 115
7.38 v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.1. 115
7.39 pressure in multi-case PINN vs FEM for Re=1500 and fc=0.1. 116
7.40 u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.2. 117
7.41 v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.2. 117
7.42 pressure in multi-case PINN vs FEM for Re=1500 and fc=0.2. 118
7.43 u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.3. 119
7.44 v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.3. 119
7.45 pressure in multi-case PINN vs FEM for Re=1500 and fc=0.3. 120
7.46 u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.1. 121
7.47 v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.1. 121
7.48 pressure in multi-case PINN vs FEM for Re=1750 and fc=0.1. 122
7.49 u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.2. 123
7.50 v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.2. 123
7.51 pressure in multi-case PINN vs FEM for Re=1750 and fc=0.2. 124
7.52 u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.3. 125
7.53 v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.3. 125
7.54 pressure in multi-case PINN vs FEM for Re=1750 and fc=0.3. 126
7.55 Norm RMSE in u velocity. 127
7.56 Norm RMSE in v velocity. 128
7.57 Norm RMSE in p pressure. 128

List of Tables

2.1 Example of AD to compute the partial derivatives $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ at $(x_1, x_2) = (2, 1)$ [8]. 32
2.2 Comparison between FEM and PINNs. 33

6.1 The parameters of the 3 case studies for the evaluation of the TSPs implementation errors. 56
6.2 Errors for the 2 networks 62
6.3 The parameters of the 3 case studies for the accuracy evaluation of the adaptive activation functions. 74
6.4 Errors for the different networks and studies. 85
6.5 Number of points for the 2 samplings. 85
6.6 The parameters of the 3 case studies for the accuracy evaluation of different samplings. 85
6.7 Errors for the 2 samplings. 87
6.8 Errors for different networks. 90

7.1 Errors for the different studies. 127

Abbreviations-Acronyms

ANN	Artificial Neural Network
PINN	Physics Informed Neural Network
FCNN	Fully Connected Neural Network
ML	Machine Learning
SciML	Scientific Machine Learning
MLP	Multilayer Perceptron
CFD	Computational Fluid Dynamics
PDE	Partial Differential Equation
FDM	Finite Difference Method
FEM	Finite Element Method
BC	Boundary Condition
IC	Initial Condition
AD	Automatic Differentiation
ReLU	Rectified Linear Unit
LReLU	Leaky Rectified Linear Unit
ELU	Exponential Linear Unit
SiLU	Sigmoid-weighted Linear Unit
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
Norm RMSE	Normalized Root Mean Squared Error
SGD	Stohastic Gradient Descent
AdaGrad	Adaptive Gradient
RMSProp	Root Mean Square Propagation
Adam	Adaptive Momentum
Re	Reynolds Number
TSP	Tube Specific Parameter
API	Application Programming Interface
YAML	Yet Another Markup Language
BFGS	Broyden–Fletcher–Goldfarb–Shann
L-BFGS	Limited-memory BFGS

Nomenclature

Symbol	Description	Units
u	Velocity x axis	m/s
v	Velocity y axis	m/s
p	Pressure	Pa
μ	Dynamic viscocity	Kg/(m*s)
ν	Kinematic viscocity	m^2/s
ρ	Density	Kg/m^3
fc	Stenosis severity	-

Part I

Theoretical part

Chapter 1

Basics of neural networks

1.1 Introduction to neural networks

Artificial Neural Networks (ANNs) are adaptive systems inspired by the functioning of the human brain. These systems can modify their internal structure to achieve specific functional objectives, making them particularly effective at solving nonlinear problems by reconstructing the fuzzy rules that govern optimal solutions. In Fig. 1.1, the comparison between a biological and an artificial neuron is displayed.

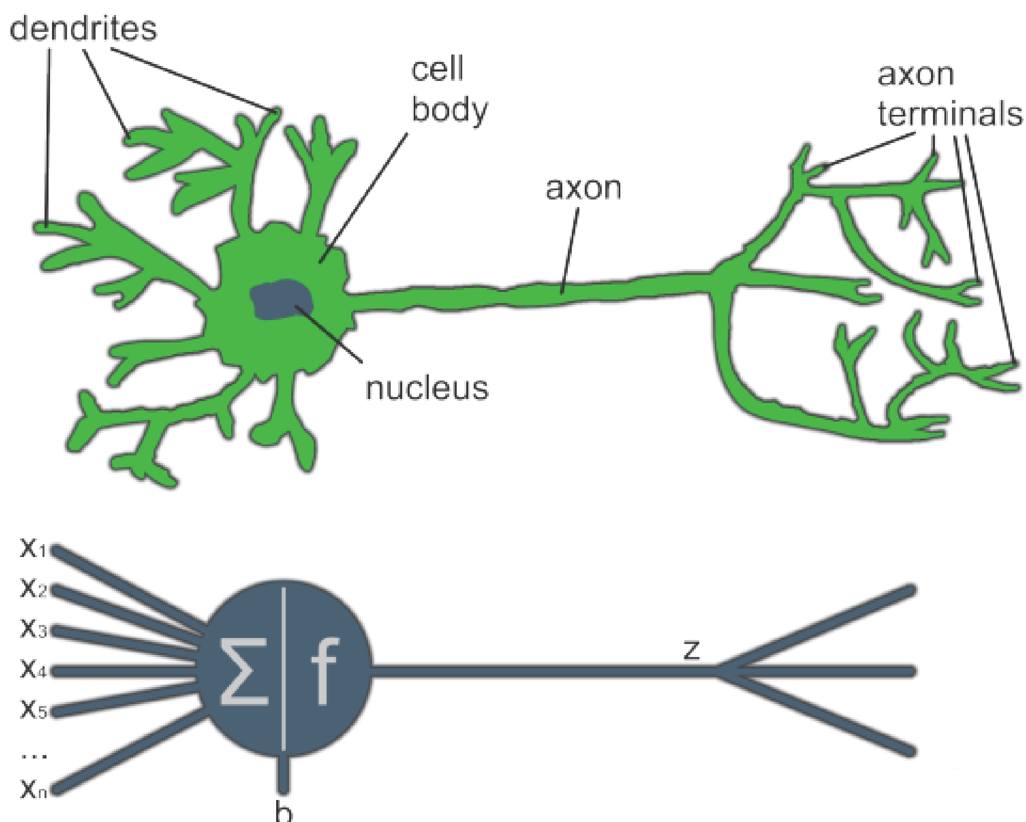


Figure 1.1: *Comparison between a biological neuron and an artificial neuron [1].*

More specifically neural networks are complex functions characterized by the weights of connections between neurons and the biases of these neurons. Let's represent these weights as W and the biases as b, with the complete set of parameters denoted as $\theta = \{W, b\}$.

Every neuron in the network calculates the weighted average of its input neurons and their accompanying weights. It then adds its bias to the result and passes the resulting value via a nonlinear activation function, which we refer to as α. Let x represent the inputs. The given notation represents a neural network that defines a function $f(x;\theta)$. This function f has the same dimension as the number of neurons in the output layer and can be a vector. The most basic arrangement of deep neural networks (DNNs) is known as the multilayer perceptron (MLP). In the case of these neural networks, the neurons are organized in sequential layers, and each neuron is linked to all the neurons in the adjacent layers. In Fig. 1.2, an example of a neural network with 3 hidden layers is demonstrated.

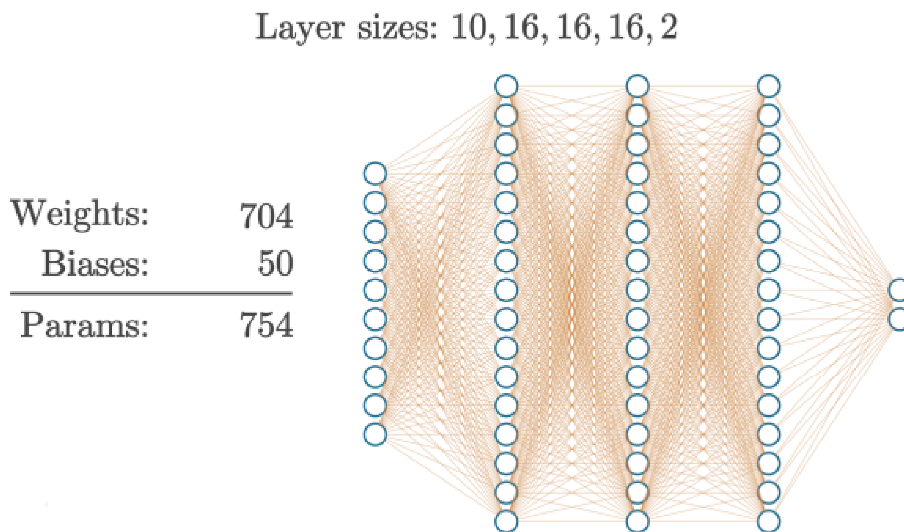


Figure 1.2: *Example of a neural network with 3 hidden layers [1].*

By employing vector notation, we represent the output of the ith layer as f_i and the biases as b_i. The weight connecting the $(i-1)^{th}$ layer to the ith layer is denoted by the matrix W_i. The input layer is designated with the index 0, whereas the output layer is designated with the index k. Thus the output of the ith layer is determined as follows:

$$f_i(x) = \begin{cases} x & \text{if } i = 0 \\ \sigma(W_i f_{i-1}(x) + b_i) & \text{if } i = 1, 2, \ldots, k-1 \\ W_i f_{i-1}(x) + b_i & \text{if } i = k \end{cases} \tag{1.1}$$

where σ is the vector that contains the outputs of the applied activation function. Also in this way, in Fig. 1.3 the output of the network can be given as $f_k(x)$:

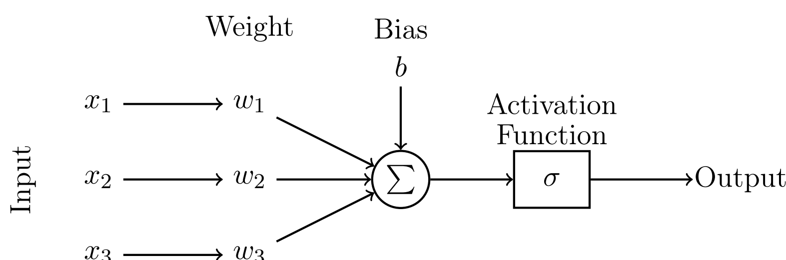


Figure 1.3: *A neuron taking an input feature of dimension* $d = 3$.

1.2 Fundamentals of activation functions

Each neuron's computation in an ANN is divided into two stages. In the first stage, the input value is multiplied by a weighted value. In the second stage, an activation function determines whether the neuron should be activated based on a threshold value, thereby facilitating the network's ability to recognize patterns between inputs and outputs accurately. Consequently, the activation function is crucial for neuron activation. In this section we will highlight some of the most important properties of activation functions:

- Non-linearity: Activation functions are vital for managing non linearity. Without them, data would only move through the network's nodes and layers using linear functions, resulting in linear outputs regardless of the number of layers, as the composite of linear functions remains linear.
- Computational Cost: The computation of activation functions should be straightforward.
- Differentiability: This property ensures a function is differentiable at every point in a specific domain, allowing for easy calculation of gradients and optimization of neuron weights. Complex activation functions can slow down computation.
- Vanishing Gradient: This problem occurs when the gradient diminishes significantly during backpropagation, causing a loss of information. An effective activation functions should mitigate this issue.
- Saturation: This term refers to the gradient approaching zero at certain points, making it difficult to adjust parameter values.
- Monotonicity: The function's graph should have no local minima, and the function's derivative should maintain a consistent sign.
- Less Parameterization: Most activation functions do not require additional parameters, simplifying the computational calculations of the neuron model.

For this reason we have to employ various activation functions for different applications. Thus, the understanding of their mechanisms can be helpful for the selection of the most suitable ones for the task.

1.3 Common activation functions

In this section some of the most commonly used activation functions are explained.

Sigmoid

The Sigmoid activation function, Eq.1.2, is continuous and constrained between 0 and 1. Its gradient also falls within the range of 0 to 1. However, in practice, the gradient often becomes zero in complex neural networks. As a result, error information cannot effectively propagate through the neurons during backpropagation, which reduces the network's performance.

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{1.2}$$

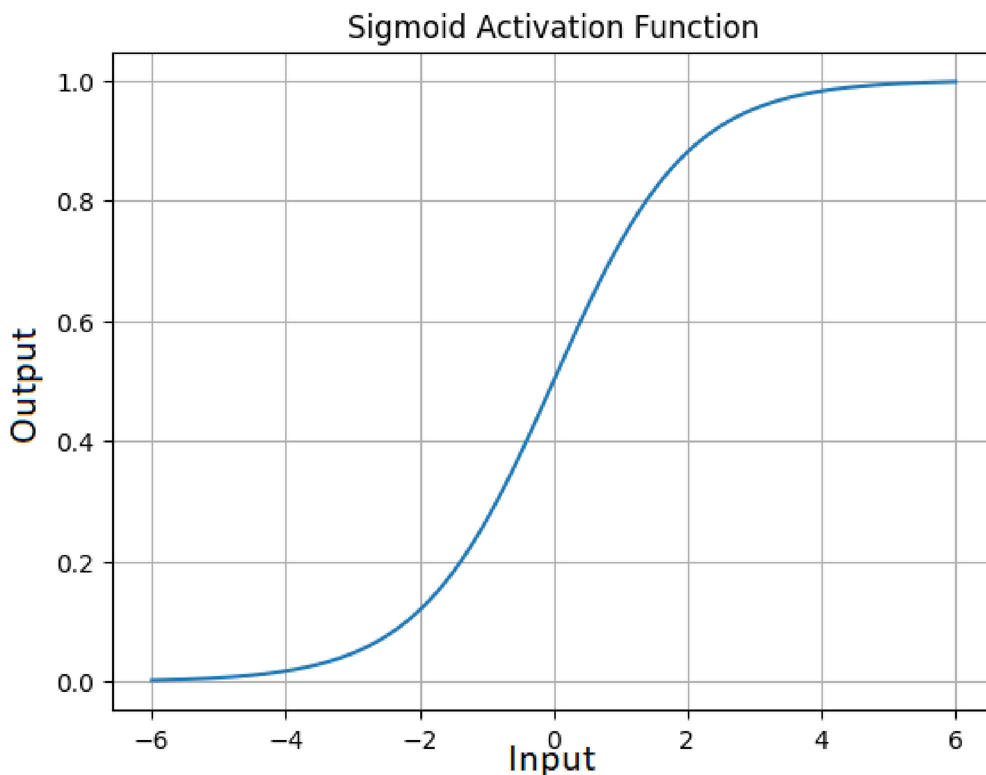


Figure 1.4: *Plot of Sigmoid activation function.*

Hyperbolic tangent activation function

The hyperbolic tangent (tanh) activation function, Eq.1.3, is a modified variant of the sigmoid function. It has the same S shape and it is continuous within the range of -1 and +1. When the input data consists of high positive or small negative numbers, the tanh activation function will become saturated and cease to respond to minor changes in

the input data, as it also experiences the gradient disappearing issue. Consequently, the weight will remain unchanged and the gradient will gradually diminish, rendering the DNN incapable of completing its training.

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1.3}$$

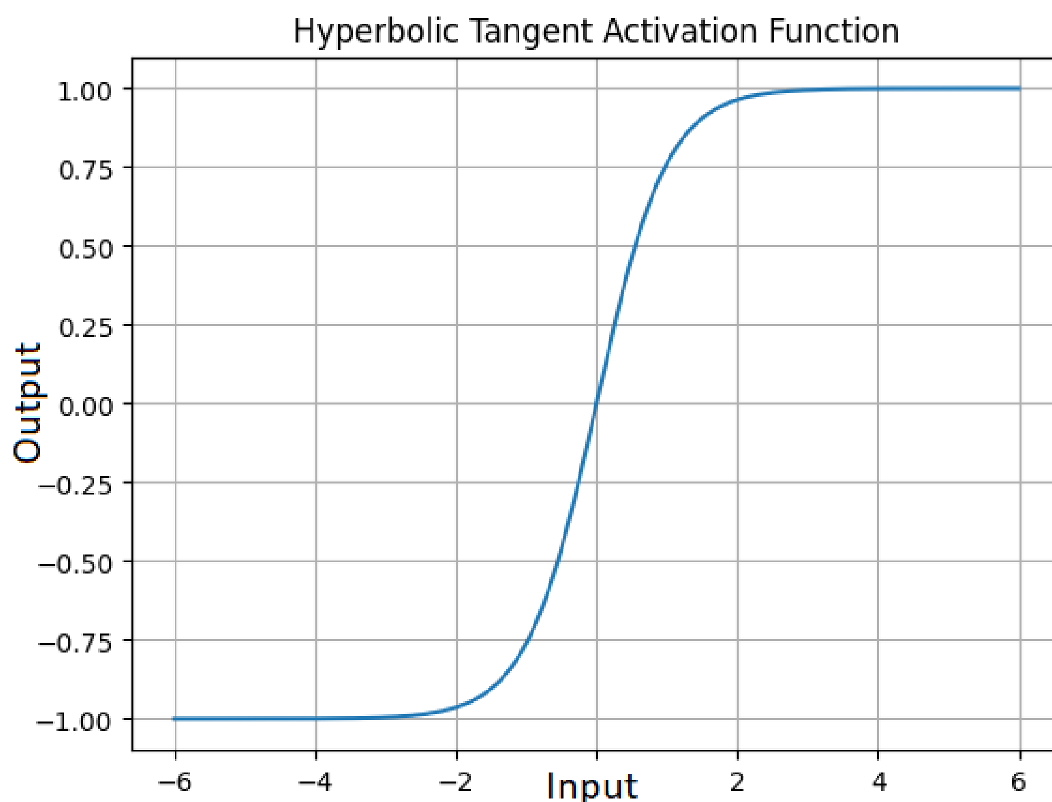


Figure 1.5: *Plot of tanh activation function.*

Rectified Linear Unit

The Rectified Linear Unit (ReLU), Eq.1.4, is a fast and effective activation function for training complicated neural networks. Compared to the Sigmoid and tanh activation functions, it provides superior performance in deep learning. The ReLU function preserves the fundamental properties of linear models, which allows for straightforward optimization using gradient descent methods. This is due to the fact that the ReLU function closely approximates the linear function. The ReLU activation function performs a threshold operation on each input element as it assigns a value of zero to negative arguments and retains the original value for positive arguments. Despite the fact that it offers certain benefits, it suffers from the ”dying ReLU” problem. More specifically several neurons during training always output zero and do not contribute to the learning process, causing reduced model capacity and slower convergence.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{1.4}$$

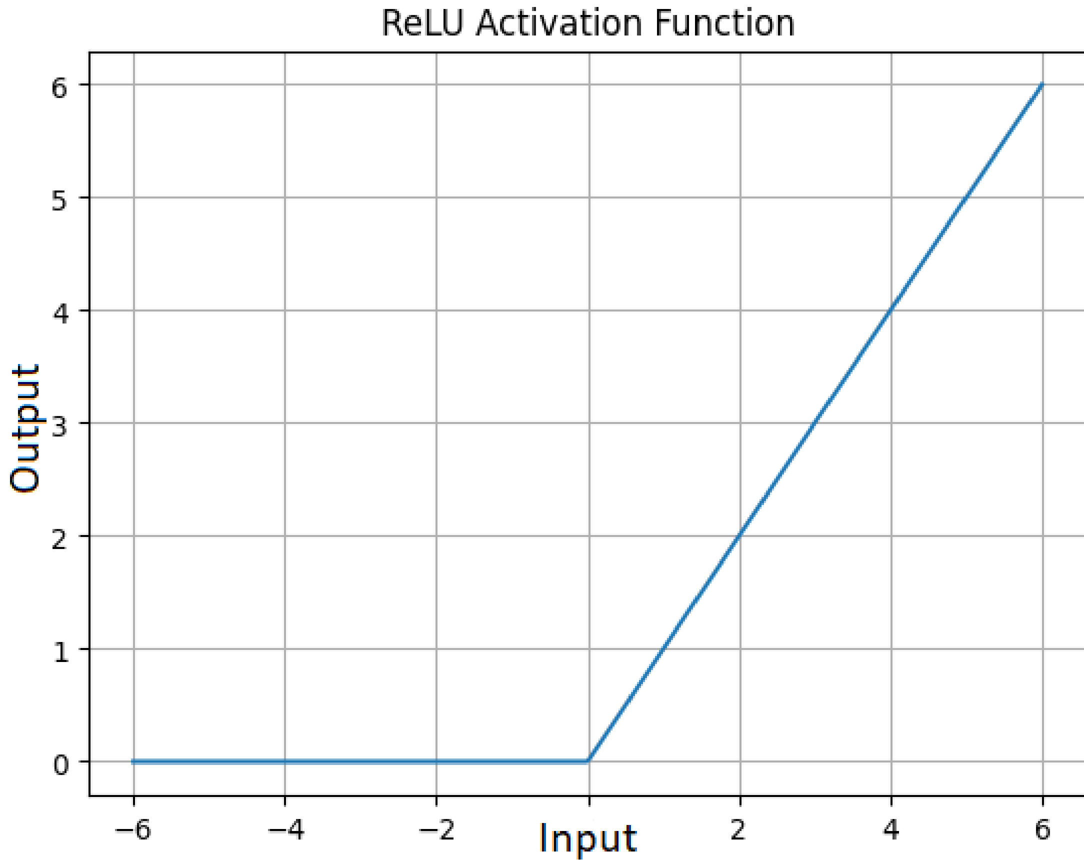


Figure 1.6: *Plot of ReLU activation function.*

Leaky ReLU

Leaky ReLU, Eq.1.5, is an early rectified-based activation function that was created based on ReLU. The LReLU was developed as a potential solution to address the potential problems of the ReLU mentioned before. It allows a small negative input to generate a large gradient. Yet, LReLU functions in a manner that is nearly identical to regular rectifiers. It has a negligible impact on network performance. Its linearity prevents its usage in complex tasks. In specific application circumstances, it exhibits inferior performance compared to Sigmoid and tanh.

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{1.5}$$

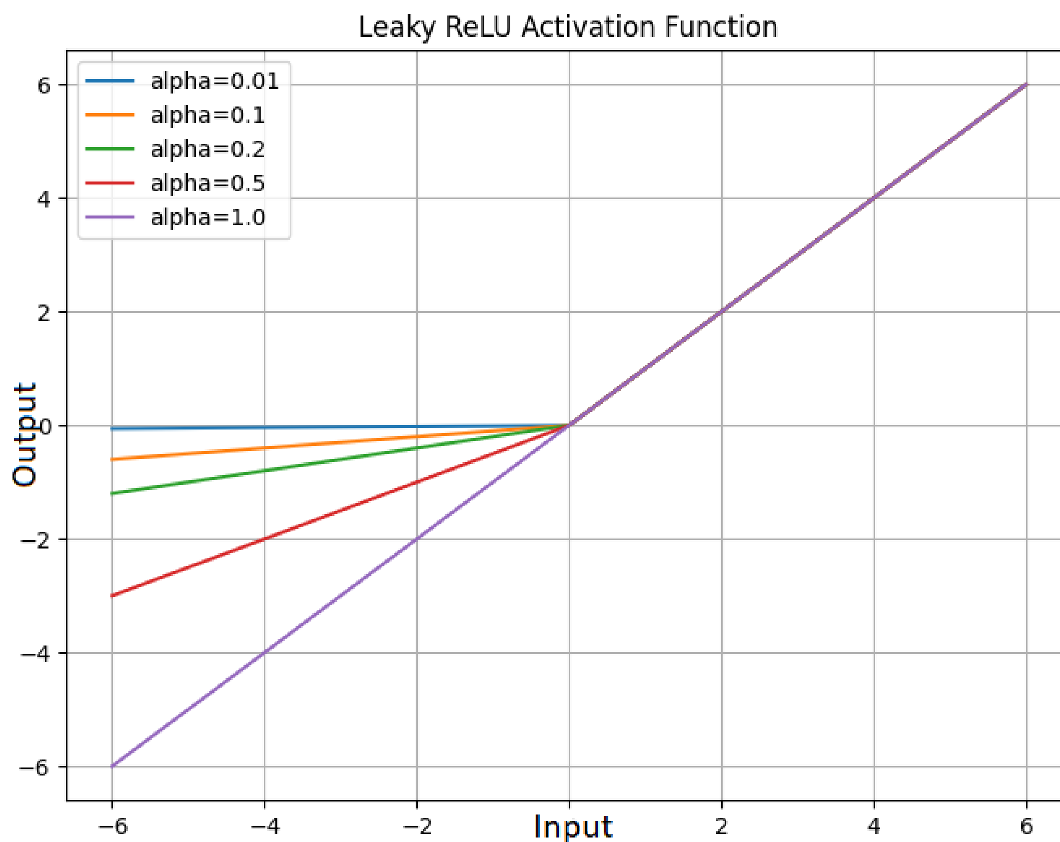


Figure 1.7: *Plot of Leaky ReLU activation function.*

Exponential Linear Unit

The Exponential Linear Unit (ELU), Eq.1.6, is an activation function that maintains the input's value when it is positive, but assigns non-zero values to negative arguments. As denoted below, α is a hyper-parameter that determines the value for negative inputs.

$$ELU(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases} \tag{1.6}$$

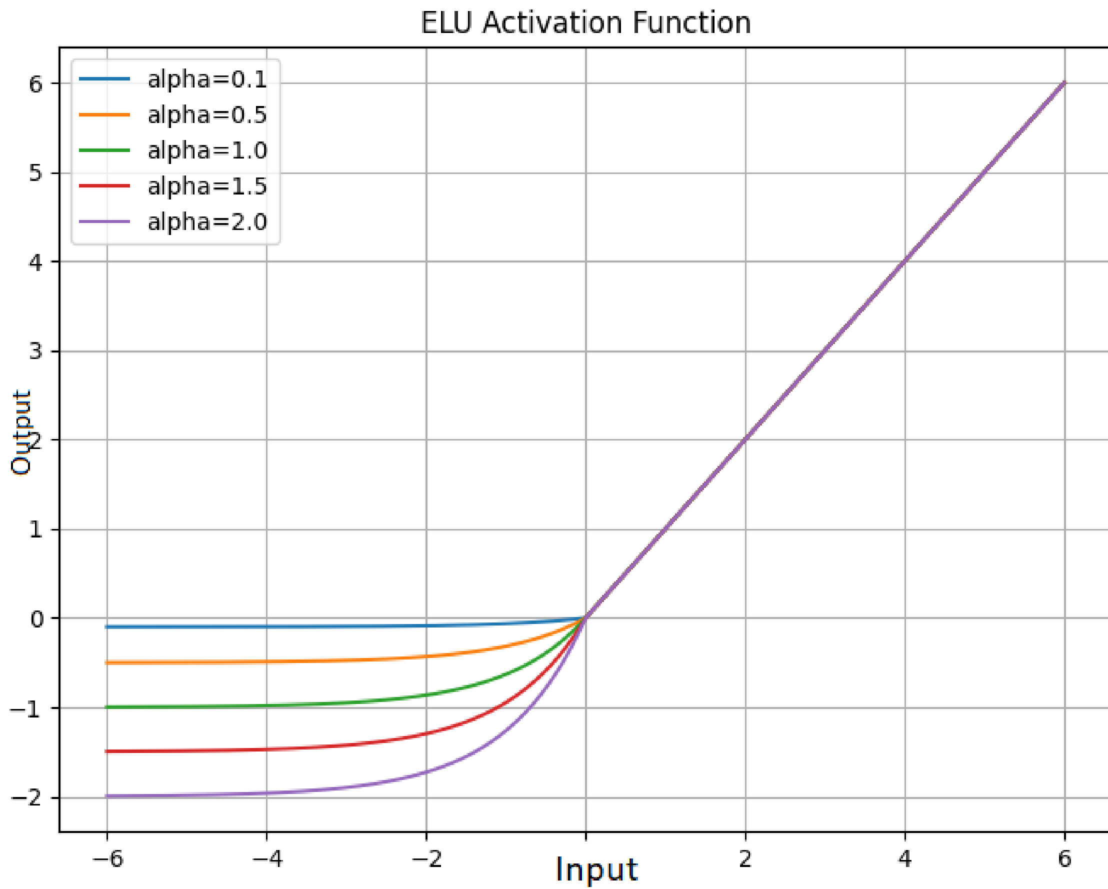


Figure 1.8: *Plot of ELU activation function.*

Sigmoid-weighted Linear Unit

In the Sigmoid-weighted Linear Unit (SiLU), Eq.1.7, the input is multiplied by the output of the sigmoid function. The SiLU function exhibits smoothness, non-monotonicity, and is bounded below while being unbounded above. For large input, the SiLU activation function behaves similarly to the ReLU activation function.

$$\text{SiLU}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{1 + e^{-x}} \tag{1.7}$$

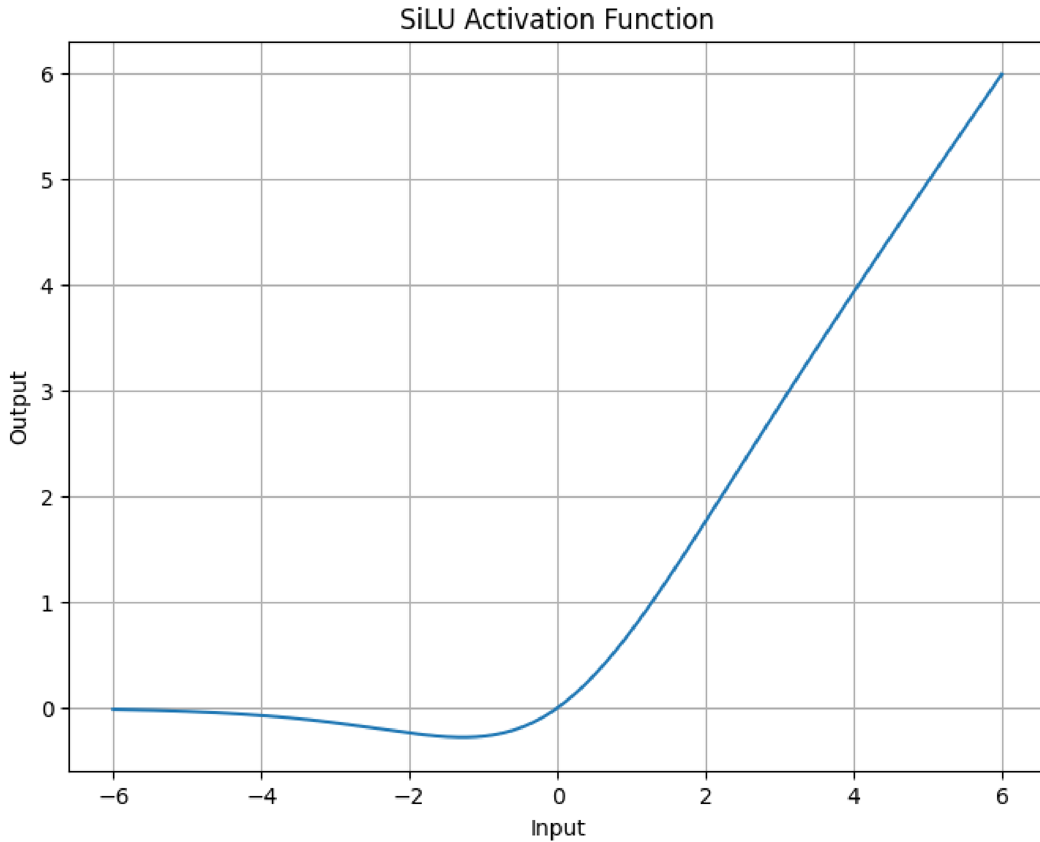


Figure 1.9: *Plot of SiLU activation function.*

1.4 Loss functions

While neural networks offer appealing features in theory, their practical implementation is not simple. Finding sets of weights that yield correct approximations has proven to be highly tough. There appear to be no straightforward methods for determining these weights. Nevertheless, the task of determining optimal weights for neural networks can be formulated as a minimization problem by defining an error measure between the parameterized function given by the neural network and the desired function to be approximated. These error measures are commonly known as loss functions. Since, a regression task is investigated in this Thesis, the mean squared error (MSE) loss is the suitable loss function,

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (f(x_i, \theta) - y_i)^2 \tag{1.8}$$

where $f_{(x_i,\theta)}$ is the neural network's approximation and y_i is the ground truth.
The optimizers are responsible for identifying the ideal parameters that minimize the loss function. This process in the context of neural networks is called training.

1.5 First order optimization algorithms

The first order optimization algorithms use the first order derivative of the loss function and by gradually decreasing it, converge to the desired loss level. They are widely used in machine learning applications with the most popular being the Stochastic Gradient Descent (SGD) algorithm and the Adam algorithm.

Stohastic Gradient Descent

The SGD algorithm first calculates the first partial derivative of the loss function with respect to its parameter vector and then subtracts this number multiplied by a learning rate from the parameter values. This process is repeated until a local or global minimum of the function is found. It is true that for small values of the learning rate more iterations occur until the minimum is found, while for large values there is the risk of not finding or jumping out of that minimum since the slope of the function is constantly changing. Historically SGD refers to an optimizer that fits a single sample at the time and should not be confused with Batch Gradient Descent and Mini-batch Gradient Descent which fit a whole dataset at once or batches respectively.

Algorithm 1.1: *Stochastic Gradient Descent.*

Input: Learning rate η, initial parameters θ_0, number of epochs E
Output: Optimized parameters θ
1: Initialize parameters $\theta \leftarrow \theta_0$
2: **for** epoch $= 1$ to E **do**
3:     **for** each training example (x_i, y_i) **do**
4:         Compute the gradient: $g_i = \nabla_\theta L(\theta; x_i, y_i)$
5:         Update parameters: $\theta \leftarrow \theta - \eta \cdot g_i$
6:     **end for**
7: **end for**
8: **return** θ

AdaGrad

The AdaGrad is an abbreviation of "Adaptive Gradient" and is a method that imposes a separate learning rate on each parameter instead of a single one shared by all parameters. An advantage of this method is the normalization of the updates made to the parameters, since, during the training, the values of some weights may increase significantly compared to other weights and thus, not all the neurons of the model are used effectively. AdaGrad accomplishes this by maintaining a history of previous updates which works as follows:

Algorithm 1.2: *AdaGrad.*

Input: Learning rate η, initial parameters θ_0, number of epochs E, small constant ϵ
Output: Optimized parameters θ
1: Initialize parameters $\theta \leftarrow \theta_0$
2: Initialize accumulated gradients $G \leftarrow 0$
3: **for** epoch = 1 to E **do**
4: **for** each training example (x_i, y_i) **do**
5: Compute the gradient: $g_i = \nabla_\theta L(\theta; x_i, y_i)$
6: Accumulate the squared gradient: $G \leftarrow G + g_i^2$
7: Update parameters: $\theta \leftarrow \theta - \frac{\eta}{\sqrt{G+\epsilon}} \cdot g_i$
8: **end for**
9: **end for**
10: **return** θ

Root Mean Square Propagation

RMSProp, which stands for Root Mean Square Propagation is another adaptation of SGD. RMSProp, like AdaGrad, calculates an adaptive learning rate for each parameter, but it employs an alternative method for updating the parameters.

Algorithm 1.3: *RMSProp.*

Input: Learning rate η, initial parameters θ_0, number of epochs E, decay rate ρ, small constant ϵ
Output: Optimized parameters θ
1: Initialize parameters $\theta \leftarrow \theta_0$
2: Initialize squared gradient moving average $E[g^2] \leftarrow 0$
3: **for** epoch = 1 to E **do**
4: **for** each training example (x_i, y_i) **do**
5: Compute the gradient: $g_i = \nabla_\theta L(\theta; x_i, y_i)$
6: Update the moving average: $E[g^2] \leftarrow \rho E[g^2] + (1-\rho)g_i^2$
7: Update parameters: $\theta \leftarrow \theta - \frac{\eta}{\sqrt{E[g^2]+\epsilon}} \cdot g_i$
8: **end for**
9: **end for**
10: **return** θ

Adaptive Momentum

Adam, short for Adaptive Momentum, is currently the most widely-used optimizer. It builds upon RMSProp by incorporating the momentum concept from SGD. This means that instead of directly applying current gradients, Adam applies momentums as in SGD with momentum, followed by a per-weight adaptive learning rate using the cache, similar to RMSProp. Additionally, Adam includes a bias correction mechanism (not to be confused with the layer's bias). This mechanism is applied to both the cache and momentum to correct for their initial zero values, which can skew results in the early training steps.

Algorithm 1.4: *Adam.*

Input: Learning rate η, initial parameters θ_0, number of epochs E, decay rates β_1, β_2, small constant ϵ
Output: Optimized parameters θ
1: Initialize parameters $\theta \leftarrow \theta_0$
2: Initialize first moment vector $m_0 \leftarrow 0$, second moment vector $v_0 \leftarrow 0$
3: **for** epoch $= 1$ to E **do**
4: **for** each training example (x_i, y_i) **do**
5: Compute the gradient: $g_i = \nabla_\theta L(\theta; x_i, y_i)$
6: Update biased first moment estimate: $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_i$
7: Update biased second moment estimate: $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_i^2$
8: Compute bias-corrected first moment estimate: $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
9: Compute bias-corrected second moment estimate: $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
10: Update parameters: $\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
11: **end for**
12: **end for**
13: **return** θ

1.6 Second order optimization algorithms

In addition to using derivatives to find parameters that minimize error, second-order optimization methods utilize the Hessian matrix, or variations therefore, to determine the curvature of the cost function. Generally, second-order optimization methods have not yet proven to be as effective as first-order methods, primarily due to their significant computational burden.

Broyden–Fletcher–Goldfarb–Shanno algorithm

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is one of the optimization algorithms designed to exploit the advantages of the Newton-Raphson method without the computational complexity required to create the Hessian matrix. The Newton method calculates the Hessian matrix of the error function, which consists of the second partial derivatives of the function. This calculation, along with inverting the matrix for each iteration, is both time-consuming and computationally intensive. Instead of calculating the Hessian matrix in every iteration, the BFGS algorithm approximates it, avoiding the explicit computation of all second derivatives of the cost function. Consequently, it creates Hessian matrices with elements corresponding to the parameters of the neural network. However, a disadvantage of the BFGS algorithm is the need to store the matrix from the previous iteration to compute the next one. It becomes clear that for training an NN with a large number of parameters (weights and thresholds), storing the matrix between iterations is infeasible. The Limited-memory BFGS (L-BFGS) algorithm addresses this problem by simulating BFGS while using limited memory. Specifically, L-BFGS stores a limited number of vectors that represent the Hessian approximation, making it suitable for optimization problems with a vast number of variables.

ALGORITHM 1.5: *L-BFGS.*

Input: Initial parameters θ_0, number of epochs E, small constant ϵ, history size m
Output: Optimized parameters θ
1: Initialize parameters $\theta \leftarrow \theta_0$
2: Initialize empty lists for storing past updates: $\mathcal{S} \leftarrow \{\}$ and $\mathcal{Y} \leftarrow \{\}$
3: Initialize gradient $\nabla L_0 \leftarrow \nabla L(\theta_0)$
4: **for** epoch $t = 1$ to E **do**
5: Calculate search direction: $d_t \leftarrow -H_t \cdot \nabla L_t(\theta)$
6: Perform line search to determine step size α_t
7: Update parameters: $\theta \leftarrow \theta + \alpha_t \cdot d_t$
8: Compute new gradient: $\nabla L_{t+1} \leftarrow \nabla L(\theta)$
9: Store update differences: $s_t \leftarrow \theta_{t+1} - \theta_t$, $y_t \leftarrow \nabla L_{t+1} - \nabla L_t$
10: Update history: Append s_t to $\mathcal{S}$, y_t to $\mathcal{Y}$
11: **if** history size $> m$ **then**
12: Remove the oldest entry in $\mathcal{S}$ and $\mathcal{Y}$
13: **end if**
14: Update Hessian approximation H_t using the stored $\mathcal{S}$ and $\mathcal{Y}$
15: **end for**
16: **return** θ

1.7 Adaptive activation functions

During the training of neural networks, it is possible to increase the convergence and accuracy of the model by learning both linear and nonlinear transformations. This can be achieved by utilizing global adaptive activation functions, as suggested by Jagtap and Karniadakis [9]. Global adaptive activations involve a trainable parameter that is multiplied by the input to the activations to adjust the steepness of the activations. Thus, a non linear change at the layer ℓ will be expressed in the following form,

$$\mathcal{N}^{\ell}\left(H^{\ell-1};\theta,a\right) = \sigma\left(a\mathcal{L}^{\ell}\left(H^{\ell-1}\right)\right), \tag{1.9}$$

where $\mathcal{N}^{\ell}$ is the nonlinear transformation at layer ℓ, $H^{\ell-1}$ is the output of the hidden layer $\ell-1$, θ is the set of model weights and biases, a is the global adaptive activation parameter, σ is the activation function, and $\mathcal{L}^{\ell}$ is the linear transformation at layer ℓ. Like the network weights and biases, the global adaptive activation parameter a is also a trainable parameter. These trainable parameters are optimized by:

$$\theta^*, a^* = \arg\min_{\theta,a} L(\theta, a). \tag{1.10}$$

Chapter 2

Physics informed neural networks

2.1 Introduction to Physics informed neural networks

Over the past 20 years, deep learning has proven to be highly effective in various applications, including computer vision and natural language processing, through the utilization of DNNs. Although deep learning has achieved significant success in these and related domains, its adoption in the field of scientific computing remains limited. Recently, there has been a development in solving partial differential equations (PDEs) using deep learning techniques. This new sub-field, known as Scientific Machine Learning (SciML), involves solving PDEs in either the traditional differential form or the integral form. Specifically, we can substitute conventional numerical discretization techniques with a neural network that provides an approximation of the solution. In order to acquire an approximation solution of a PDE using deep learning, a crucial step is to restrict the neural network in such a way that it minimizes the residual of the PDE. Additionally it should be highlighted that deep learning offers a mesh-free approach that differs from standard approaches like the finite difference method (FDM) and the finite element method (FEM) as it utilizes automatic differentiation to overcome the curse of dimensionality.

2.2 PINNs for solving PDEs

Let's examine the following PDE parameterized by λ for the solution $u(\mathbf{x})$ with $\mathbf{x} = (x_1, \ldots, x_d)$ defined on a domain $\Omega \subset \mathbb{R}^d$:

$$f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \lambda\right) = 0, \quad \mathbf{x} \in \Omega, \tag{2.1}$$

with boundary conditions

$$\mathcal{B}(u, \mathbf{x}) = 0 \quad \text{on} \quad \partial\Omega, \tag{2.2}$$

where $\mathcal{B}(u, \mathbf{x})$ can take on any of the following boundary conditions: Dirichlet, Neumann, Robin, or periodic. In the context of transient problems, we treat time t as a distinct element of $\mathbf{x}$, and Ω represents the temporal region. The initial condition can be regarded as a specific form of Dirichlet boundary condition applied to the spatio-temporal domain. For the solution of the PDE using PINNs it is necessary to follow the methodology as

described below [2]:

1. Create a neural network $\hat{u}_\Theta$ with parameters Θ.

2. Define the training sets $\mathcal{T}_f$ and $\mathcal{T}_b$ for the PDE and boundary/initial conditions, respectively.

3. Formulate the loss function $\mathcal{L}_\mathcal{T}(\Theta)$ by combining the weighted MSE losses from the PDE and the boundary/initial condition residuals.

4. Optimize the neural network by minimizing the loss function $\mathcal{L}_\mathcal{T}(\Theta)$.

To illustrate the above points we use the schematic of a PINN solving the diffusion equation $\frac{\partial u}{\partial t} = \lambda \frac{\partial^2 u}{\partial x^2}$ with mixed boundary conditions (BC) $u(x,t) = g_D(x,t)$ on $\Gamma_D \subset \partial\Omega$ and $\frac{\partial u}{\partial n}(x,t) = g_R(u,x,t)$ on $\Gamma_R \subset \partial\Omega$. The initial condition (IC) is treated as a special type of boundary condition and $\mathcal{T}_f$ and $\mathcal{T}_b$ denote the two sets of residual points for the equation and BC/IC respectively:

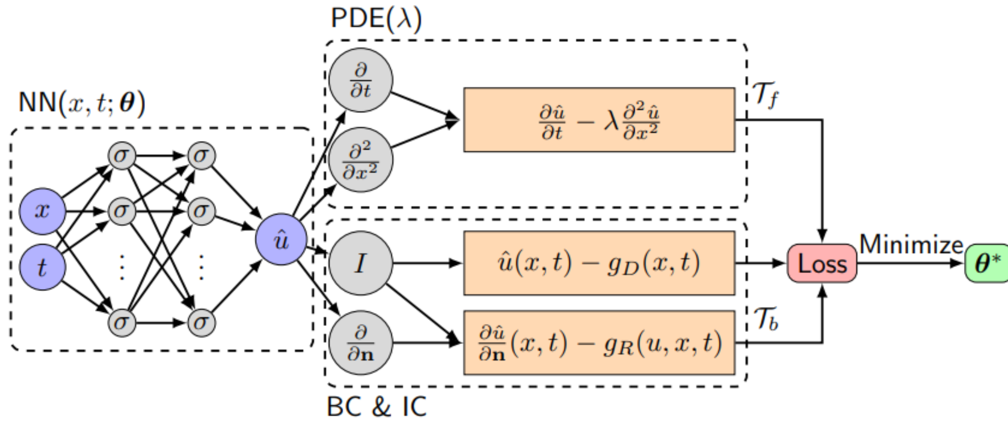


Figure 2.1: *Schematic of a PINN solving the diffusion equation [2].*

The computation of partial derivatives is crucial for solving the PDE. Through the utilization of AD, we can obtain the derivatives of any order of our network $\hat{u}_\Theta$ with respect to all related input variables, regardless of the programming code's structure. Therefore, we may incorporate the PDE residual into our calculations without requiring a computational mesh, as is typically done in the finite element approach. This inclusion is achieved by considering two subsets of the training data $\mathcal{T} \subset \tilde{\Omega}$. The set $\mathcal{T}_f \subset \tilde{\Omega}$ contains points within the domain while $\mathcal{T}_b \subset \partial\tilde{\Omega}$ contains points on the boundary and initial data. It should be emphasized that the training points are spread randomly throughout the domain.

In order to quantify the difference between the neural network approximation $\hat{u}$ and the constraints, we utilize a loss function that is defined as the weighted sum of the L^2 norm of the residuals for both the equation and boundary conditions.

$$\mathcal{L}(\theta; \mathcal{T}) = w_f \mathcal{L}_f(\theta; \mathcal{T}_f) + w_b \mathcal{L}_b(\theta; \mathcal{T}_b), \tag{2.3}$$

where

$$\mathcal{L}_f(\theta; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \lambda\right) \right\|_2^2, \tag{2.4}$$

$$\mathcal{L}_b(\theta; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_b} \|\mathcal{B}(\hat{u}, \mathbf{x})\|_2^2, \tag{2.5}$$

and w_f and w_b are the weights. The loss pertains to derivatives, such as the partial derivative $\partial\hat{u}/\partial x_1$ or the normal derivative at the boundary $\partial\hat{u}/\partial\mathbf{n} = \nabla\hat{u} \cdot \mathbf{n}$, which are computed with the help of AD.

Lastly our model is ready to enter the phase of training, similarly to classic neural networks. Due to the non-convex nature of the optimization problem, there is no theoretical assurance that this approach will converge to the global minimum. However, in their work Raissi, Perdikaris, and Karniadakis [8] highlight that if a given PDE is well-posed, the method of PINNs can provide accurate predictions by using a sufficient number of collocation points inside the domain T_f.

2.3 Automatic differentiation

When using PINNs, it is necessary to calculate the gradients of the network outputs in relation to the network inputs. There are four potential approaches for calculating the derivatives [10]:

- Manually calculating the derivative using analytical methods
- Approximating the derivative using finite differences or other numerical techniques
- Using symbolic differentiation in software programs like Mathematica, Maxima, and Maple
- Employing AD, also known as algorithmic differentiation

In deep learning a particular technique of AD is used, to evaluate derivatives, called back-propagation. Given that the neural network functions as a composition, the chain rule is applied iteratively by AD to calculate the derivatives. AD involves two distinct steps: a forward pass to calculate the values of all variables, and a backward pass to calculate the derivatives. In order to illustrate the concept of AD, we examine a Fully Connected Neural Network (FCNN) that consists of a single hidden layer with two input variables, x_1 and x_2, and one output variable, y:

$$\begin{aligned} v &= -2x_1 + 3x_2 + 0.5, \\ h &= \tanh(v), \\ y &= 2h - 1. \end{aligned}$$

The Table 2.1 illustrates the process of performing the forward pass and backward pass of AD to calculate the partial derivative $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ at $(x_1, x_2) = (2, 1)$

Table 2.1: *Example of AD to compute the partial derivatives* $\frac{\partial y}{\partial x_1}$ *and* $\frac{\partial y}{\partial x_2}$ *at* $(x_1, x_2) = (2, 1)$ *[8].*

Forward pass	**Backward pass**
$x_1 = 2$ $x_2 = 1$	$\frac{\partial y}{\partial y} = 1$
$v = -2x_1 + 3x_2 + 0.5 = -0.5$ $h = \tanh v \approx -0.462$	$\frac{\partial y}{\partial h} = 2$ $\frac{\partial y}{\partial v} = \frac{\partial y}{\partial h} \cdot \frac{\partial h}{\partial v} = 2 \cdot \text{sech}^2(v) \approx 1.573$
$y = 2h - 1 = -1.924$	$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial x_1} = 1.573 \times (-2) \approx -3.146$ $\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial x_2} = 1.573 \times 3 \approx 4.719$

It is evident that AD simply demands a single forward pass and a single backward pass to evaluate all the partial derivatives, regardless of the input dimension. On the other hand, when employing finite differences to calculate each partial derivative $\frac{\partial y}{\partial x_i}$ it is necessary to compute two function values: $y(x_1, \ldots, x_i, \ldots, x_{d_{in}})$ and $y(x_1, \ldots, x_i + \Delta x_i, \ldots, x_{d_{in}})$, where Δx_i is a small quantity. Consequently, a total of $d_{in} + 1$ forward passes are needed to assess all the partial derivatives. Therefore, AD is significantly more effective than finite difference when the input dimension is large.Additionally AD can be recursively applied n times to compute n^{th} order derivatives. Nevertheless, using a nested technique can result in inefficiency and numerical instability. As a result, other methods such as Taylor-Mode AD have been developed to address these issues [11].

2.4 Error analysis in PINNs

Previous study [8] has shown that feed-forward neural networks with a sufficient number of neurons have the ability to accurately and consistently approximate any function, as well as its partial derivatives. Still, neural networks implemented in real-world scenarios are constrained by their finite capacity. Let $\mathcal{F}$ represent the set of all functions that may be expressed using our selected neural network architecture. The solution u is improbable to be a member of the family $\mathcal{F}$. We define $u_{\mathcal{F}}$ as the argument that minimizes the difference between the functions in F and u, denoted as $u_{\mathcal{F}} = \arg\min_{f \in \mathcal{F}} \|f - u\|$. By training the neural network on the training dataset $\mathcal{T}$, we can define $u_{\mathcal{T}}$ as the neural network that minimizes the loss function $u_{\mathcal{T}} = \arg\min_{f \in \mathcal{F}} \mathcal{L}(f; \mathcal{T})$. To elucidate, we make the assumption that u, $u_{\mathcal{F}}$, and $u_{\mathcal{T}}$ are clearly formulated and distinct. Minimizing the loss to find $u_{\mathcal{T}}$ is typically expensive to compute and as a result our optimizer provides an approximate answer, denoted as $\tilde{u}_{\mathcal{T}}$. The total error $\mathcal{E}$ can be analyzed as it follows:

$$\mathcal{E} := \|\tilde{u}_{\mathcal{T}} - u\| \leq \underbrace{\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|}_{\mathcal{E}_{\text{opt}}} + \underbrace{\|u_{\mathcal{T}} - u_{\mathcal{F}}\|}_{\mathcal{E}_{\text{gen}}} + \underbrace{\|u_{\mathcal{F}} - u\|}_{\mathcal{E}_{\text{app}}}. \tag{2.6}$$

The approximation error $\mathcal{E}_{\text{app}}$ quantifies the degree to which $u_{\mathcal{F}}$ can accurately approximate u. The generalization error $\mathcal{E}_{\text{gen}}$ is dependent on the amount and distribution of residual points inside $\mathcal{T}$ along with the capacity of the family $\mathcal{F}$. Neural networks with greater complexity exhibit reduced approximation errors, but may also result in increased generalization errors, a phenomenon known as the bias-variance tradeoff. Overfitting arises

when the generalization mistake becomes predominant. Furthermore, the optimization error $\mathcal{E}_{\text{opt}}$ emerges from the intricacy of the loss function and the configuration of the optimization process, including the learning rate and the number of epochs. Currently, there is no error quantification available for PINNs.

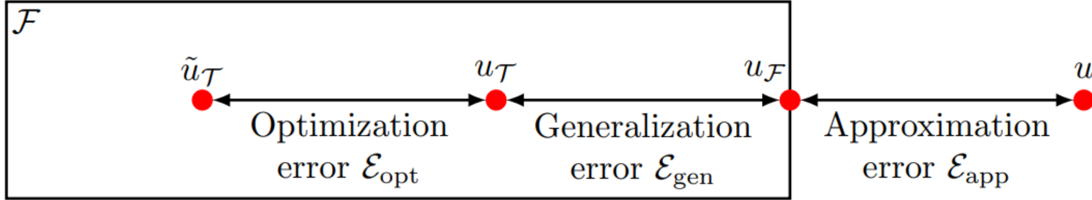


Figure 2.2: *Illustration of errors of a PINN.*

2.5 Comparison between PINNs and FEM

To clarify the concepts behind PINNs and make them more accessible to those familiar with FEM, a point-by-point comparison between PINNs and FEM is offered [8]:

- In FEM, the solution u is usually represented by a piecewise polynomial, whereas in PINNs, a neural network serves as the surrogate model, parameterized by its weights and biases.
- FEM demands the construction of a mesh, while PINNs are entirely mesh-free, allowing for the use of either a grid or randomly distributed points.
- FEM transforms a PDE into an algebraic system by utilizing stiffness and mass matrices, whereas PINNs incorporate the PDE and boundary conditions directly into the loss function.
- In the final step, FEM solves the algebraic system using a linear solver, while in PINNs, the network is trained via a gradient-based optimization algorithm.

Fundamentally, PINNs offer a nonlinear approximation of the function and its derivatives, while FEM provides a linear approximation.

Table 2.2: *Comparison between FEM and PINNs.*

	FEM	**PINN**
Basis function	Piecewise polynomial (linear)	Activation function (nonlinear)
Unknowns	Nodal values	Weights and biases
Spacial discretization	Computational mesh	Scattered points (mesh-free)
PDE embedding	Algebraic system	Loss function
Solver	Linear solver	Gradient-based optimizer
Errors	Approximation/quadrature errors	$\mathcal{E}_{\text{app}}$, $\mathcal{E}_{\text{gen}}$, and $\mathcal{E}_{\text{opt}}$
Error bounds	Partially available	Not available yet

Chapter 3

Nvidia Modulus Sym

3.1 Introduction to Nvidia Modulus Sym

Nvidia Modulus Sym is a deep learning framework that integrates the power of physics and PDEs with AI to create more resilient models for enhanced analysis.

Machine learning (ML) and deep learning models can be applied to physics-based systems in various ways, depending on the availability of observational data and the depth of understanding of the underlying physics. Based on these factors, ML/DL methodologies can be generally categorized into three types: forward (physics-driven) approaches, data-driven approaches, and hybrid approaches that combine both physics and data assimilation.

3.2 PINNs in Nvidia Modulus Sym

Modulus Sym has an alternative approach to calculating losses, which differs from the traditional method outlined in section 1.4. The losses are expressed in integral form, and Eq.1.8 can be represented as:

$$L_f = \int_\Omega \left\| \frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2} \right\|_2^2 dx \approx \left(\int_\Omega dx \right) \cdot \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| \frac{\partial \hat{u}}{\partial t} - \lambda \frac{\partial^2 \hat{u}}{\partial x^2} \right\|_2^2 \tag{3.1}$$

Following that, the integral is approximated using Monte Carlo integration. Thus, Eq.3.1 is derived in a manner that is equivalent to Eq.1.4, but with the additional consideration of scaling by the area/volume of the domain, to preserve consistent loss per area across all domains. The losses associated with the boundary conditions are handled in a similar manner.

3.3 Nvidia Modulus Sym building blocks

Nvidia Modulus sym is neural network framework built on Pytorch. It offers APIs that enable the user to develop his own applications using the pre-existing modules.

Geometry and data

The geometry module in Modulus Sym allows users to create a geometry from scratch using basic shapes or to import an existing geometry from a mesh. For data-driven problems, Modulus Sym provides various methods for accessing data, including standard in-memory datasets and lazy loading techniques for handling large-scale datasets.

Nodes

In Modulus Sym, Nodes represent elements that are executed during the forward pass in training. A Node can encapsulate a torch.nn.Module and includes additional information about the required input and output variables. This enables Modulus Sym to construct execution graphs and automatically add missing components to compute necessary derivatives. Nodes can include built-in PyTorch neural networks, user-defined PyTorch networks, feature transformations, models, functions, or even equations.

Constraints

Constraints refer to the specific objectives that are set for training in Modulus Sym. A Constraint comprises the loss function and the group of Nodes that Modulus Sym use to construct a computational graph for execution. Several physical problems require the inclusion of many training objectives in order to establish a clear and precise definition of the problem. Constraints serve as the mechanism to define such problems.

Domain

The Domain contains all Constraints along with other essential components for the training process, such as Inferencers, Validators, and Monitors. In Modulus Sym, user-defined Constraints are added to the training Domain, forming a set of training objectives.

Solver

A Solver is the core Modulus Sym trainer responsible for the implementation of the optimization loop and management of the training process. It utilizes a defined Domain and invokes the Constraints, Inferencers, Validators, and Monitors as needed. During each iteration, the Solver calculates the global loss from all Constraints and optimizes any trainable models within the Nodes specified by the Constraints.

Hydra

Hydra is a configuration package integrated into Modulus Sym. It allows users to set hyperparameters, which define the neural network's structure and govern its training, using YAML configuration files. Hydra is the initial component activated when solving a problem with Modulus Sym and directly influences all other components.

Inferencers

An Inferencer executes only the forward pass of a group of Nodes. Inferencers can be utilized during the training process to evaluate training quantities or obtain predictions for the purpose of visualization or deployment. Hydra configuration options dictate the frequency at which Inferencers are invoked.

Validators

Validators function similarly to Inferencers, except they additionally incorporate validation data. The accuracy of the model is quantified during training by validating it against ground truth data obtained by an alternative method.

Monitors

Monitors function similarly to Inferencers, but they estimate specific measures rather than fields. These metrics can refer to global quantities, such as the total energy of a system.

3.4 Modulus Sym workflow

The figure below depicts a standard workflow for development in Modulus Sym. While not all problems will necessitate this identical approach, it can be used as a general guideline. The essential stages of this procedure comprise:

- **Initialize Hydra**: Use the Modulus Sym `main` decorator to load the YAML configuration file.
- **Load Data**: Import datasets if necessary.
- **Set Geometry**: Define the system's geometry if required.
- **Create Nodes**: Establish any necessary Node, such as the neural network model.
- **Establish Domain**: Generate a Domain object for training.
- **Add Constraints**: Develop each of the N_c Constraints sequentially and integrate them into the `Domain`.
- **Include Validators, Inferencers, and Monitors**: Generate any Inferencer, Validator, or Monitor as required and incorporate them into the Domain.
- **Initialize Solver**: Create a Solver using the training Domain.
- **Execute Solver**: Run the Solver. The training process will optimize the neural network to solve the physics problem.

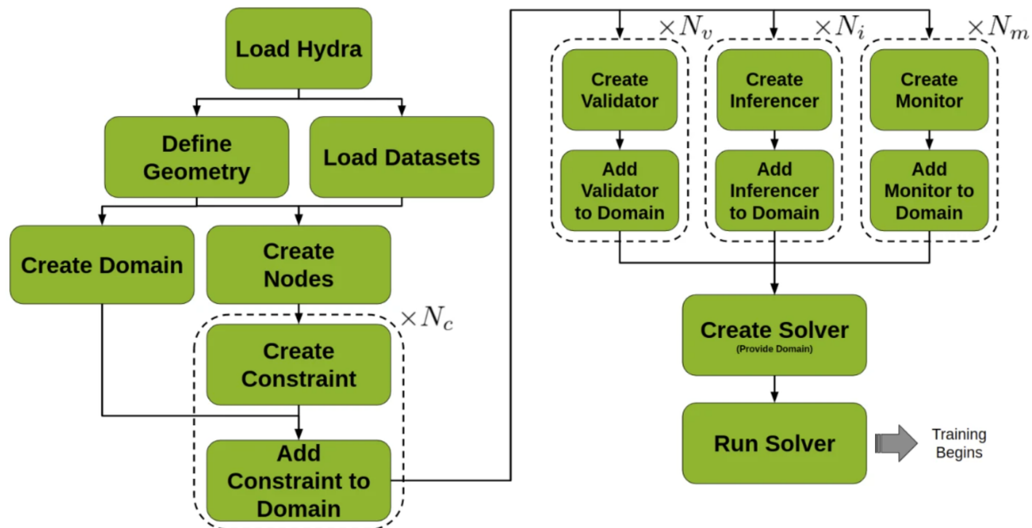


Figure 3.1: *A typical workflow in Modulus Sym [3].*

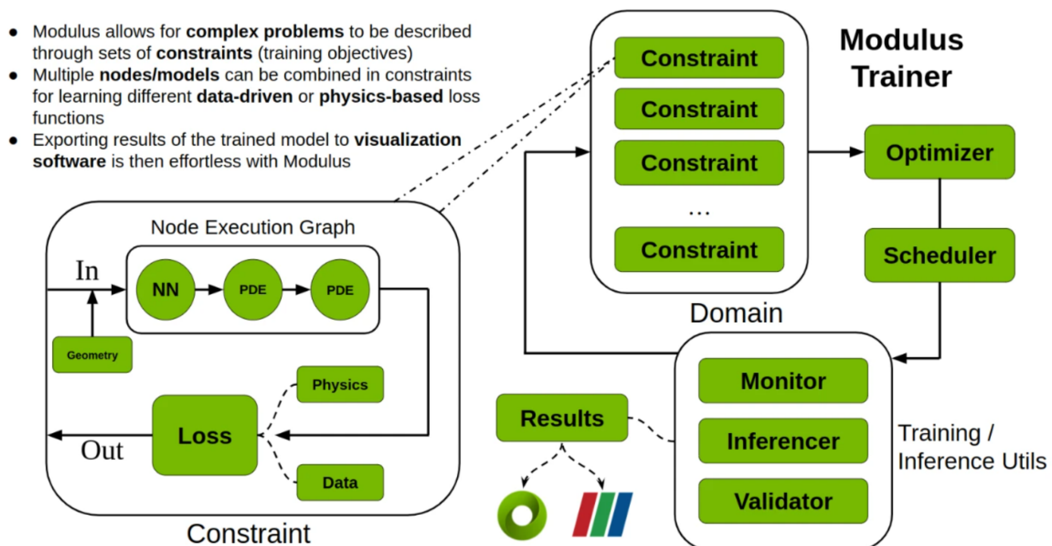


Figure 3.2: *Modulus Sym training algorithm [3].*

Chapter 4

PINNs in Hemodynamics

The study of Hemodynamics is crucial for the clinical identification and treatment of cardiovascular diseases such as aneurysms and stenosis. Hemodynamic modelling relies on CFD, which often demand expertise and significant amount of computational resources. However, PINNs have recently emerged as an innovative approach for addressing those issues by including physics laws into the training algorithm of deep learning models, enabling real time flow predictions even with limited data.

One of the foundational studies showcasing the utility of PINNs in hemodynamics was presented in the study by Raissi et al. [4]. The single case PINN proposed leverages the spatio-temporal visualizations of a passive scalar to inference the velocity and pressure fields. In this scenario, the passive scalar refers to the bolus dye commonly delivered into the bloodstream aiming to facilitate blood flow monitoring and medical imaging. It should be noted that this methodology does not need the implementation of initial and boundary conditions as the algorithm is agnostic to geometry and only utilizes the conservation laws. This early study paved the way for the wider adoption of PINNs in hemodynamics and served as a source of inspiration for many later investigations.

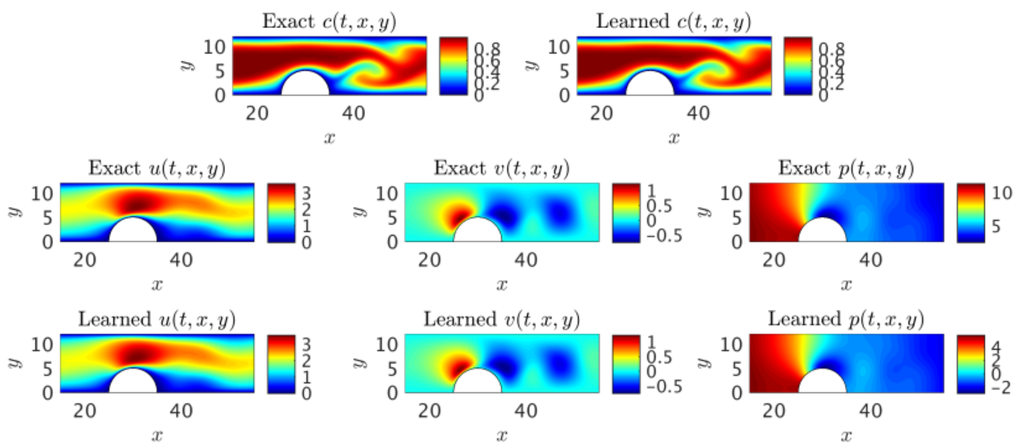


Figure 4.1: *2D channel flow over an obstacle [4]: A representative snapshot of the input data on the concentration field within the training domain is plotted in the top left panel alongside the prediction of our algorithm. The algorithm is capable of accurately reconstructing the velocity and the pressure fields without having access to any observations of these fields (shown in the second and third rows). Furthermore, no boundary conditions are specified on the boundaries of the training domain including the physical wall boundaries.*

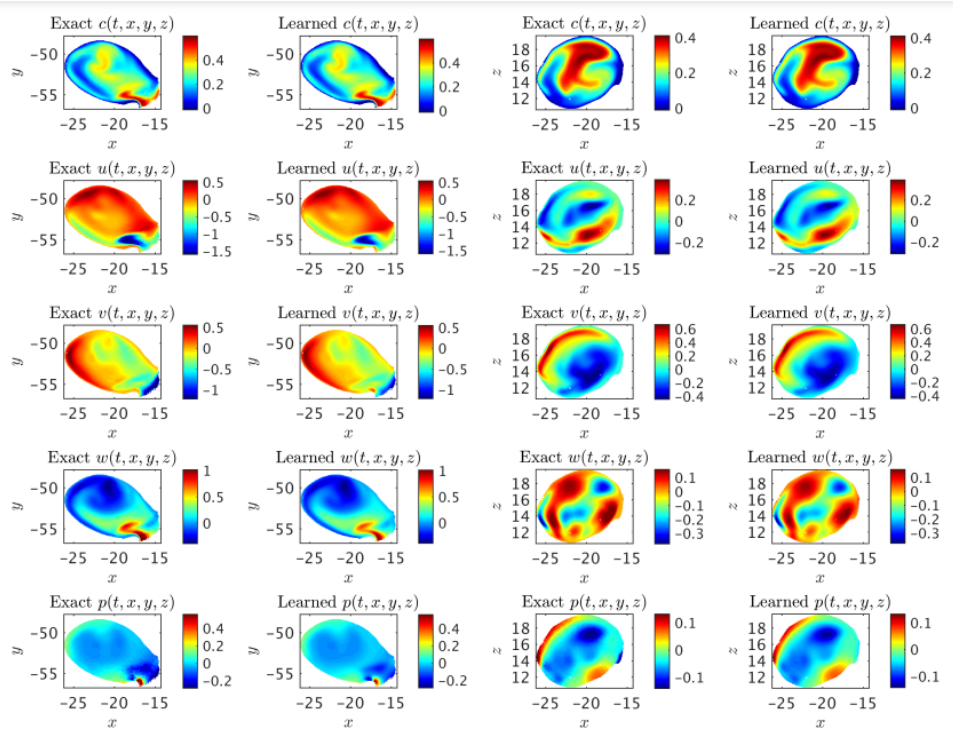


Figure 4.2: *A 3D intracranial aneurysm [4]: Contours of the exact fields and model predictions are plotted on two perpendicular planes for concentration c, velocity u, v, w, and pressure p fields in each row.*

Another critical advancement is the use of PINNs in combination with high-speed angiography (HSA) for neurovascular hemodynamics estimation by Williams et al. [5]. Like [4], it predicts velocity and pressure fields by utilizing the convection equation, without needing a predefined inlet velocity function. However, unlike [4] it enforces no slip boundary condition at the walls. Despite the fact that the PINN is single case and needs to be retrained for each new input model, the mean time for convergence was 23 minutes including all the time needed for the pre-processing of the data.

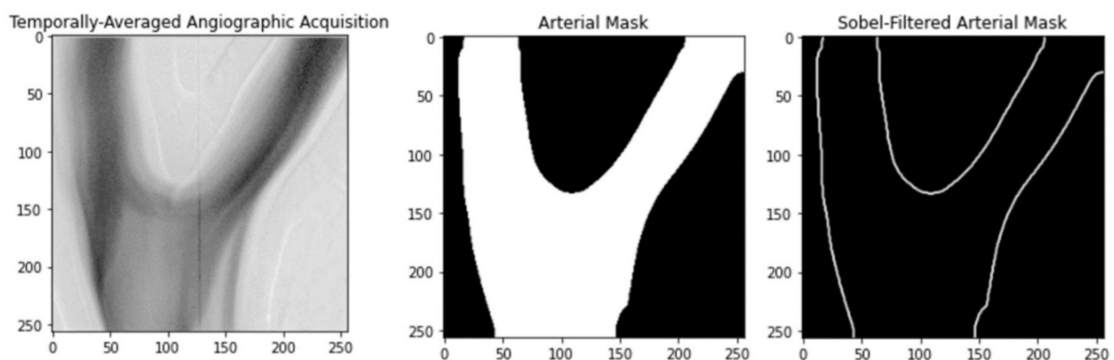


Figure 4.3: *Pre-processed 1000 fps HSA acquisition (left), its corresponding binary mask used to define the vessel lumen (middle), and the Sobel-filtered binary mask, used to define the walls of the model (right) [5].*

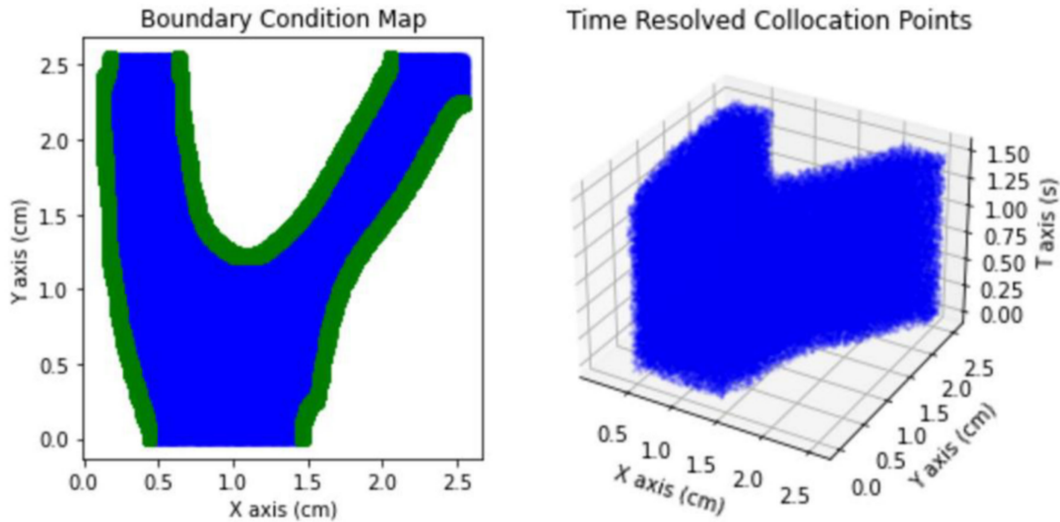


Figure 4.4: *Spatial sampling at one time step showing vessel lumen in blue, and vessel walls in green (left). Spatial sampling is then extended temporally for both the vessel lumen and vessel wall (vessel wall omitted) (right) [5].*

Figure 4.5: *n vitro HSA image sequence (left four images) compared to the normalized magnitude image (right) generated by the image-assisted PINN. Red arrows show progression of contrast media edges [5].*

In the study by Garay et al. [6], a PINN architecture was demonstrated that could predict blood flow by coupling a reduced order model with Navier-Stokes in 3 dimensions. By utilizing geometry data and the inlet velocity profile obtained from simulated 2D MRI images, a mean pressure curve measured at the left-subclavian artery, the no slip boundary condition at the walls, 3D Navier-Stokes equations and the equations of the 3 element Windkessel model, the PINN successfully solved the coupled system. It not only succeeded to solve the steady problem but also the transient despite the huge computational cost. Moreover at the end of the training process the network accurately identified the most suitable physical parameters that correspond to the proximal and distal resistance of the vasculature and the distal's vessels compliance. In summary the suggested architecture offers an innovative approach to personalized hemodynamic models based on patient's clinical data.

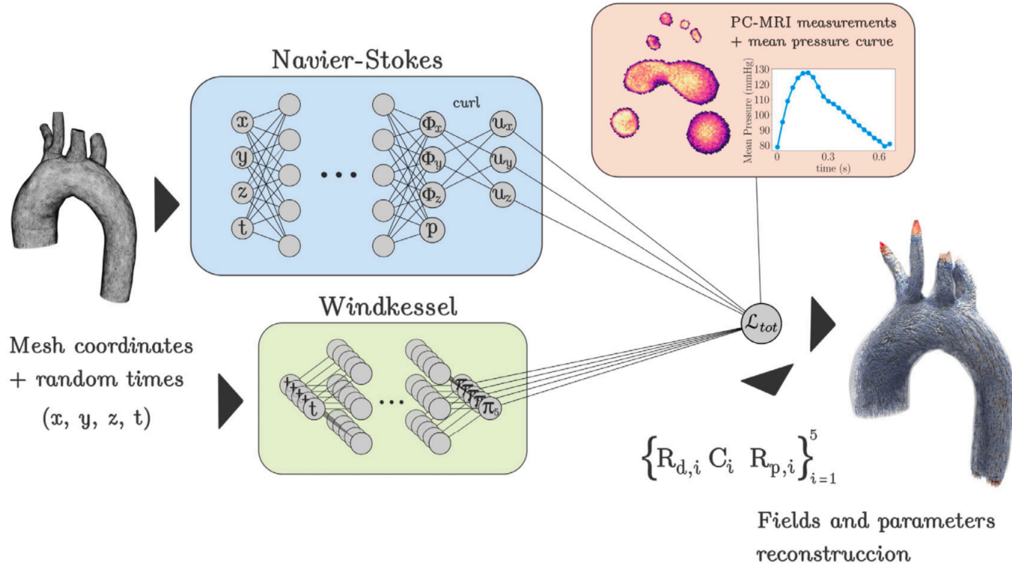


Figure 4.6: *Feed-forward neural network architecture used for the transient problem. After physics-informed training, the output of the network is used for reconstruct the 3D velocity field and the Windkessel parameters of all the domain outlets [6].*

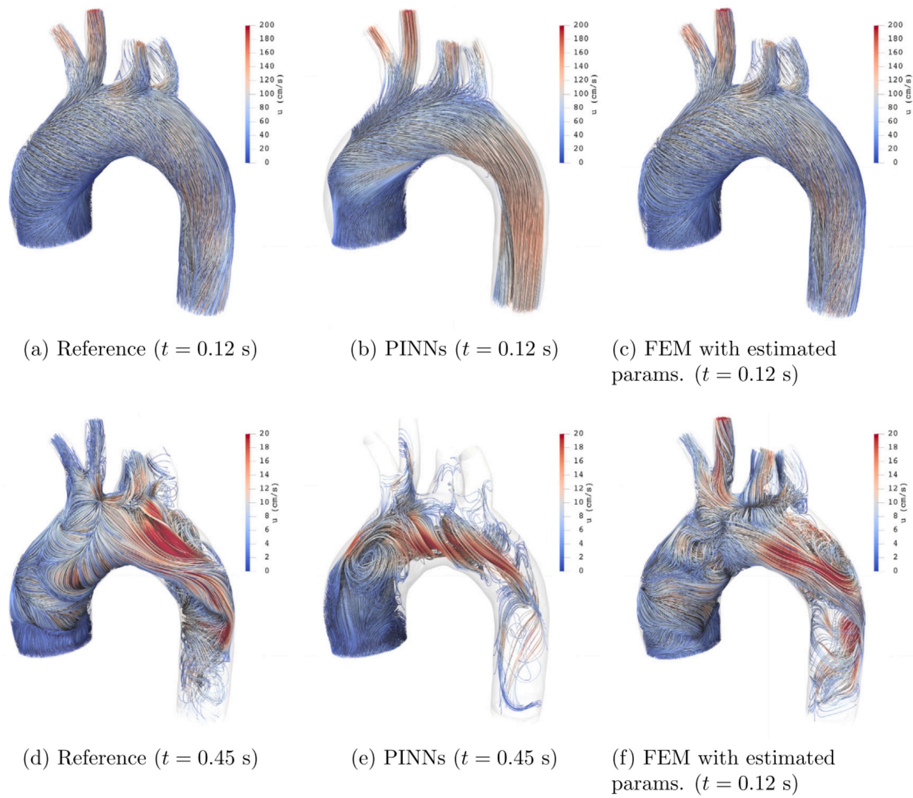


Figure 4.7: *Velocity streamlines of the transient problem for the reference solution (a and d), the mean estimated velocity estimated by the PINNs (b and e), and the velocity obtained after a FEM simulation using the found parameters (c and f) at two time instants: $t = 0.12$ s, which correspond to peak systole, and at $t = 0.45$ s, during mid diastole [6].*

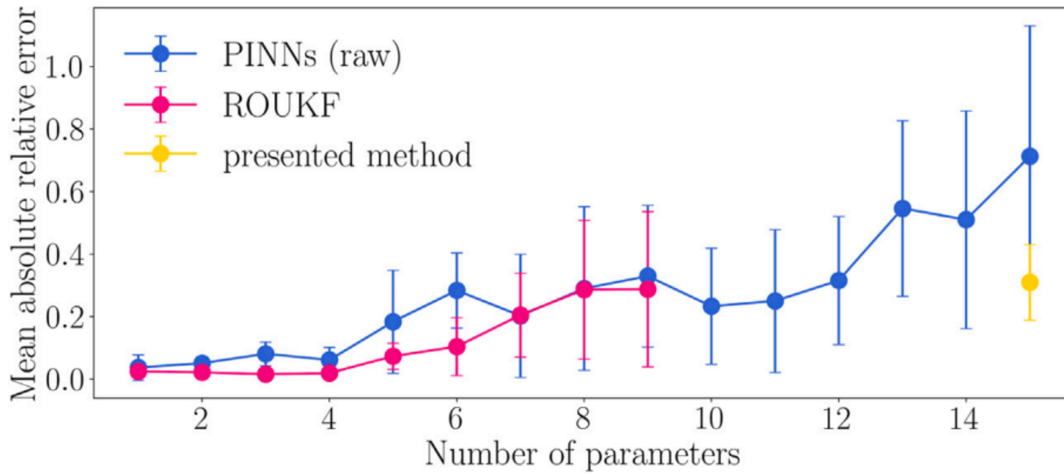


Figure 4.8: *Mean absolute relative error of the parameter estimation using the Kalman filter approach (ROUKF) and PINNs [6].*

The work by Zhang et al. [7], introduces a deep learning framework designed to map the 4D hemodynamic profile (3d space and time) of blood flow in morphologically diverse arteries. The framework integrates vessel structure and time series data with Navier-Stokes equations to predict velocity and pressure fields. This implementation of PINN exhibits superior performance and accuracy compared to previous deep learning studies. However this architecture is still not yet ready for clinical use, primarily due to the fact that it needs patient's specific boundary conditions obtained by 4D flow MRI or particle image velocimetry experiments to replace the CFD simulations, that do not represent the real flow field.

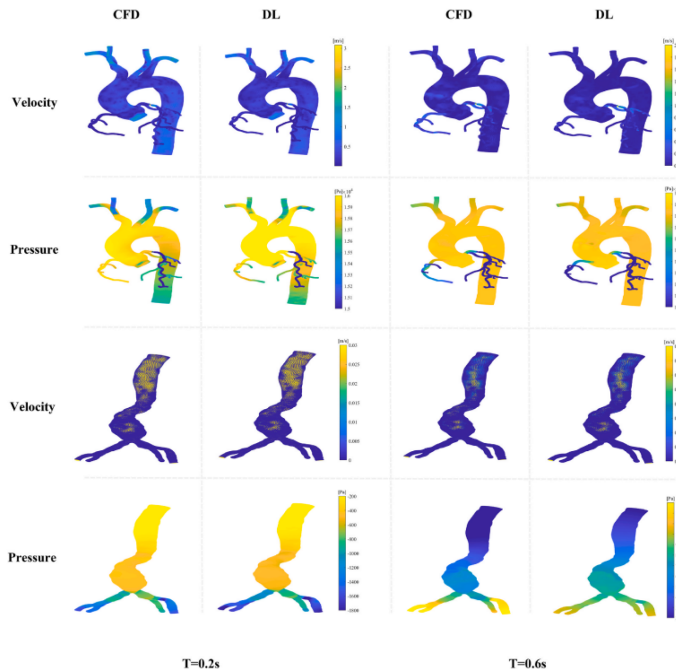


Figure 4.9: *Visual comparison of hemodynamic calculation results between CFD simulation and deep learning method [7].*

Lastly the study by Wong et al. [12] that inspired the current thesis, explored a multi case PINN approach for modelling fluid dynamics in idealized geometries. It employed an unsupervised method by just enforcing the Navier-Stokes equations along with the boundary conditions to the network's loss function, demonstrating very low errors despite the fact that it was not informed with data from simulations. As such, they can not be used for clinical applications but the strategies identified are crucial for the future 3D scale up.

Part II

Methods

Chapter 5

Hemodynamic predictions in 2D vessel stenoses using PINNs

5.1 Problem definition

Computational fluid dynamics (CFD) for tubular geometries play a vital role in hemodynamic assessments in vessels. PINNs are a potential substitute for conventional CFD techniques. Nevertheless, vanilla PINNs typically require extended training durations compared to traditional CFD techniques for individual flow scenarios, hence restricting their extensive adoption. In order to tackle this issue, a multi-case PINN strategy has been suggested. This approach involves parameterizing different geometric cases and flow conditions and pre-training them on the PINN. This allows for quick computation of hemodynamic results in unseen geometries at different Reynolds numbers.

This study aims to find the steady-state solutions for the flow of incompressible fluid (blood) in idealized 2D stenotic vessels (channel with a narrowing in the middle). The geometric shape of the stenosis and the flow conditions are described by a parameter called λ. The equations that govern this problem are as follows:

$$\nabla \cdot \mathbf{u} = 0, \quad \mathbf{x} \in \Omega, \quad \lambda \in \mathbb{R}^n \tag{5.1}$$

$$(\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u}, \quad \mathbf{x} \in \Omega, \quad \lambda \in \mathbb{R}^n \tag{5.2}$$

$$\mathbf{u} = 0, \quad \text{at } \mathbf{x} = \Gamma_{\text{wall}} \tag{5.3}$$

$$p = 0, \quad \text{at } \mathbf{x} = \Gamma_{\text{outlet}} \tag{5.4}$$

$$u = u_{\max} \cdot \left(1 - \frac{y^2}{{R_0}^2}\right), \quad v = 0, \quad \text{at } \mathbf{x} = \Gamma_{\text{inlet}}, \quad \lambda \in \mathbb{R}^n \tag{5.5}$$

where $p = p(\mathbf{x}, \lambda)$ is the fluid pressure, $\mathbf{x} = (x, y)$ is the spatial coordinates and $\mathbf{u} = \mathbf{u}(\mathbf{x}, \lambda) = [u(\mathbf{x}, \lambda), v(\mathbf{x}, \lambda)]^T$ denotes the fluid velocity with components u and v in x and y direction respectively across the fluid domain Ω and the domain boundaries Γ. A parabolic velocity profile is prescribed at the inlet and a zero pressure condition at the outlet. λ is

an n-dimensional parameter vector, consisting of two case parameters, fc (severity of the stenosis) and $u_{\max}$ (maximum inlet velocity). R_0 is the radius of the vessel away from the stenosis.

$$R(x) = \begin{cases} R_0, & \text{for } -4 \cdot R_0 \leq x < -2 \cdot R_0 \\ R_0\left(1 - \text{fc}(1 + \cos(x\pi)\right), & \text{for } -2 \cdot R_0 \leq x \leq 2 \cdot R_0 \\ R_0, & \text{for } 2 \cdot R_0 < x \leq 4 \cdot R_0 \end{cases} \tag{5.6}$$

where $R(x)$ is the radius of the vessel with respect to its length.

(a) (b)

Figure 5.1: *Example of 2 stenotic geometries, with (a)* $fc = 0.1$ *and (b)* $fc = 0.3$.

We can define A_b as the stenosis percentage of the cross section:

$$A_b = (1 - n) \cdot 100\% \tag{5.7}$$

where n is the ratio A_s/A_0, with A_s being the cross section at the point where the maximum stenosis appears (x = 0) and A_0 the cross section of the vessel away from the stenosis.

$$A_s = A_0 \cdot n => b \cdot d_s = b \cdot d_0 \cdot n => d_s = d_0 \cdot n => R_s = R_0 \cdot n => R_s/R_0 = 1 - 2 \cdot fc = n \tag{5.8}$$

where b is the depth of the vessel geometry, d_s is the diameter at the point where maximum stenosis appears and d_0 is the diameter of the vessel away from the stenosis. So A_b with respect to fc can be described as:

$$A_b = 2 \cdot fc \cdot 100\% \tag{5.9}$$

The blood was defined as a fluid with density $\rho = 1060\ Kg/m^3$ and dynamic viscosity $\mu = 0.004\ Kg/(m \cdot s)$. For the training of the multi-case PINN, the fc parameter was set to vary from 0.1 and 0.3, and u_{max} from 0.38 to 1.52 m/s, resulting at a Reynolds number range of 500-2000. Firstly the problem was selected to be dimensional however the results did not yield the expected accuracy and the problem was non dimensionalized with:

- length scale = 0.1 m (d_0)
- velocity scale = 1.52 m/s ($u_{\max}$)
- density scale = 1060 Kg/m^3 (blood density)
- kinematic viscosity scale = length scale · velocity scale

All the x, y coordinates that compose the geometry were divided by the length scale of 0.1 m. Each inlet velocity was divided by the velocity scale of 1.52 m/s ($u_{\max}$). To

instantiate the Navier-Stokes module in Modulus, we set the fluid density to a value of 1. To find the kinematic viscosity, we divided the dynamic viscosity by the density and then non-dimensionalized it by dividing it with the kinematic viscosity scale. In Eq 5.10 we get the non dimensionalized radius of the vessel geometry with respect to x.

$$R(x) = \begin{cases} 0.5, & \text{for } -2 \leq x < -1 \\ 0.5\,(1-\text{fc}(1+\cos(x\pi))\,, & \text{for } -1 \leq x \leq 1 \\ 0.5, & \text{for } 1 < x \leq 2 \end{cases} \tag{5.10}$$

5.2 Network architecture

In this thesis, we employ PINN to solve the above system of PDE. The predictions of the variables u and p are expressed as a constrained optimization problem. The network is then trained, without the use of labeled data, using the governing equations and specified boundary conditions. The loss function $L(\theta)$ for the physics constrained learning is defined as follows:

$$\mathcal{L}(\theta) = \omega_{\text{physics}}\mathcal{L}_{\text{physics}} + \omega_{\text{bc}}\mathcal{L}_{\text{BC}} \tag{5.11}$$

$$\theta^* = \arg\min_{w,b}(\mathcal{L}(\theta)) \tag{5.12}$$

where W and b refer to the weights and biases of the fully connected neural network. $\mathcal{L}_{\text{physics}}$ represents the loss function over the entire domain for the parameterized Continuity and Navier-Stokes equations, and $\mathcal{L}_{\text{BC}}$ represents the boundary conditions loss. ω_{physics} and ω_{bc} are the weights parameters for the terms and both take the value of 1. The loss expressions can be described as:

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_{\text{Domain}}}\sum_{i=1}^{N_{Dom}} |\nabla \cdot \hat{\mathbf{u}}|^2_{\Omega} + \frac{1}{N_{\text{Domain}}}\sum_{i=1}^{N_{Dom}} \left|(\hat{\mathbf{u}}\cdot\nabla)\,\hat{\mathbf{u}} + \frac{1}{\rho}\nabla\hat{p} - \nu\nabla^2\hat{\mathbf{u}}\right|^2_{\Omega} \tag{5.13}$$

$$\begin{aligned} \mathcal{L}_{\text{BC}} =& \frac{1}{N_{\text{wall}}}\sum_{i=1}^{N_{wall}} |\hat{u}|^2_{\Gamma_{\text{wall}}} + \frac{1}{N_{\text{outlet}}}\sum_{i=1}^{N_{out}} |\hat{p}|^2_{\Gamma_{\text{outlet}}} \\ &+ \frac{1}{N_{\text{inlet}}}\sum_{i=1}^{N_{in}} \left|\hat{u} - u_{\max}\cdot\left(1 - \frac{y^2}{R_0^2}\right)\right|^2_{\Gamma_{\text{inlet}}} + \frac{1}{N_{\text{inlet}}}\sum_{i=1}^{N_{in}} |\hat{v}|^2_{\Gamma_{\text{inlet}}} \end{aligned} \tag{5.14}$$

where N_{Domain}, N_{wall}, N_{inlet}, N_{outlet} is the number of randomly selected points inside the domain, at the walls, at the inlet and at the outlet respectively.

The PINN was trained and tested using a single GPU (Nvidia RTX 3070). In this study, a feedforward FCNN called f_θ was utilized where the surrogate network model is built to approximate the solutions, $\hat{y} = [\hat{u}(\mathbf{x},\lambda), \hat{v}(\mathbf{x},\lambda), \hat{p}(\mathbf{x},\lambda)]^T$. In the FCNN, the output from the network (a series of fully connected layers), $\hat{y}(\psi;\theta)$, where ψ represents the network inputs, was computed using trainable parameters θ, consisting of the weights W_i and biases

b_i of the i-th layer for n layers, according to the equation:

$$\hat{y}(\psi;\theta) = W_n \left(\Phi_{n-1} \circ \Phi_{n-2} \circ \cdots \circ \Phi_1\right)(\psi) + b_n \tag{5.15}$$

$$\Phi_i = \alpha \left(W_i \left(\Phi_{i-1}\right) + b_i\right), \quad \text{for } 2 < i < n-1 \tag{5.16}$$

where Φ_i represents the nodes of the ith layer in the network and α is the activation function. AD is used to compute partial differential operators and all networks and losses were constructed using NVIDIA's Modulus framework.

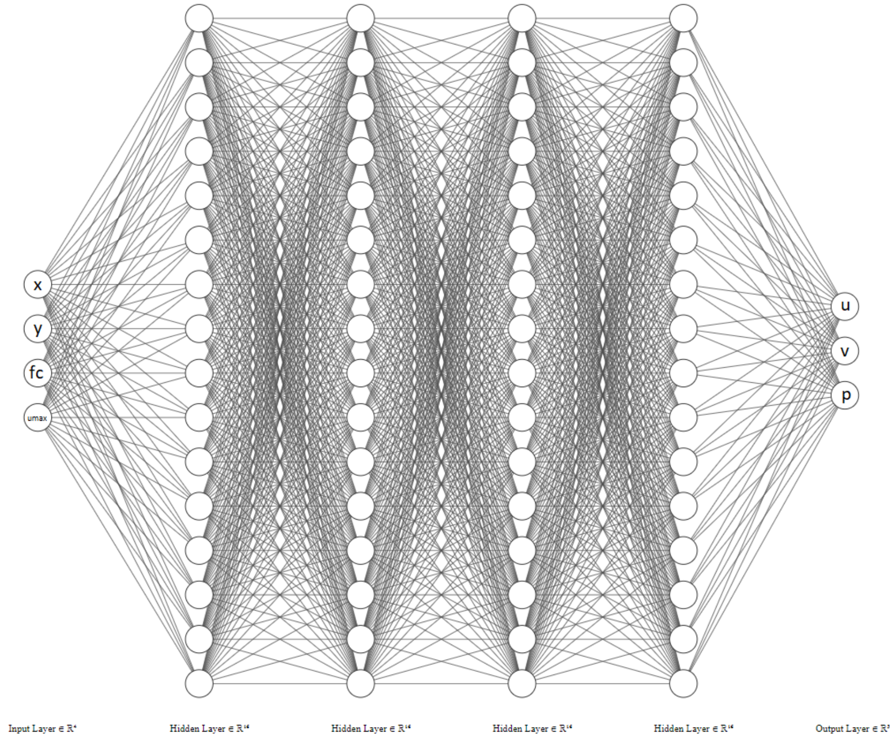


Figure 5.2: *The FCNN of our problem with 4 hidden layers and 16 nodes per hidden layer.*

5.3 Tube specifc parameters

When addressing the fluid dynamics of biomedical tube-like flows, such as vascular flows, it is crucial to consider tube-specific parameters as they directly affect it. These parameters include the distance along the tube, the centerline and the distance from the tube walls. For instance, at regions near the wall, solutions with low velocity magnitude are expected because of the physics of the no slip boundary condition. Additionally, it is expected that the fluid pressure will drop along the tube, as a result of energy losses in the flow. Therefore, it is proposed that incorporating those parameters as inputs into the tube flow PINN will improve its accuracy [12].

Thus we incorporated eight tube-specific parameters, referred to as TSPs into the PINN as inputs.

1. Normalized centerline distance, $c \in (-1, 1)$.

2. Normalized width, $L_n \in (-1, 1)$.
3. $dsq = 1 - L_n^2$
4. c^2
5. L_n^2
6. $c \times dsq$
7. $c \times L_n$
8. $L_n \times dsq$

5.4 Computational Fluid Dynamics simulations in COMSOL

CFD ground truth data of the training and prediction geometries were generated using COMSOL laminar flow steady state study. The problem was non dimensionalized as described before so that the exported data is suitable for evaluating the performance of PINNs.

The first step was the creation of the geometry. For the creation of the geometry, 4 parametric curves were defined, to generate 2 solid surfaces, that were subtracted from the rectangle 2×1 with the help of the boolean operations that are offered in COMSOL Multiphysics v6.2.

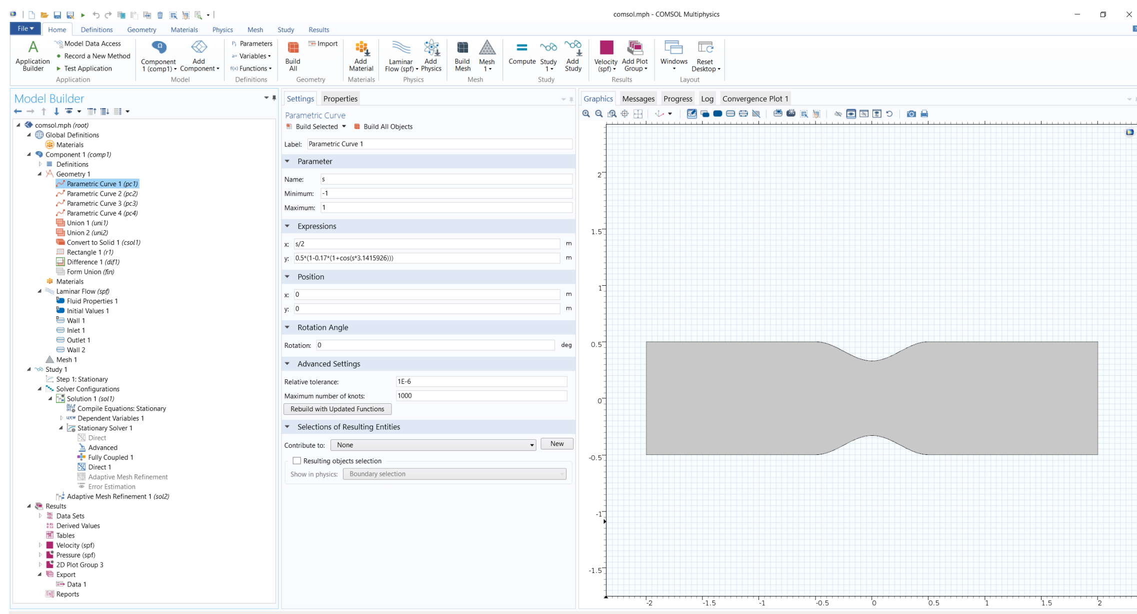


Figure 5.3: *Example of a parametric curve used to simulate the blood flow in a stenosis with fc=0.17.*

The next step was the definition of the fluid properties as shown in Fig. 5.4

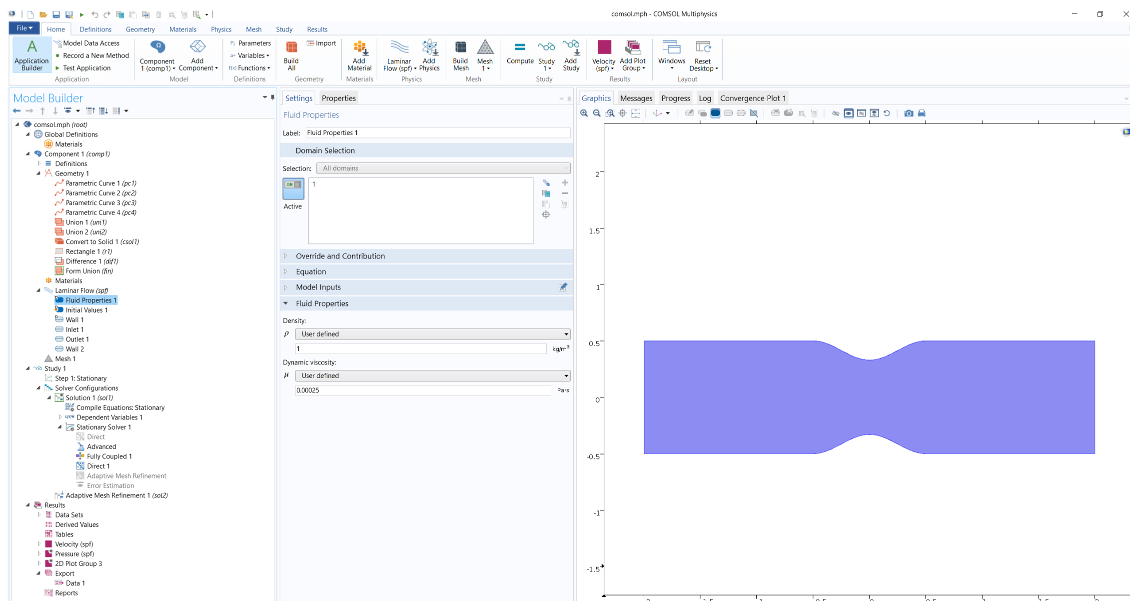


Figure 5.4: *Definition of fluid properties.*

To define the BCs, we selected the section of the geometry, where the BCs are enforced as shown in Figs 5.5, 5.6 and 5.7.

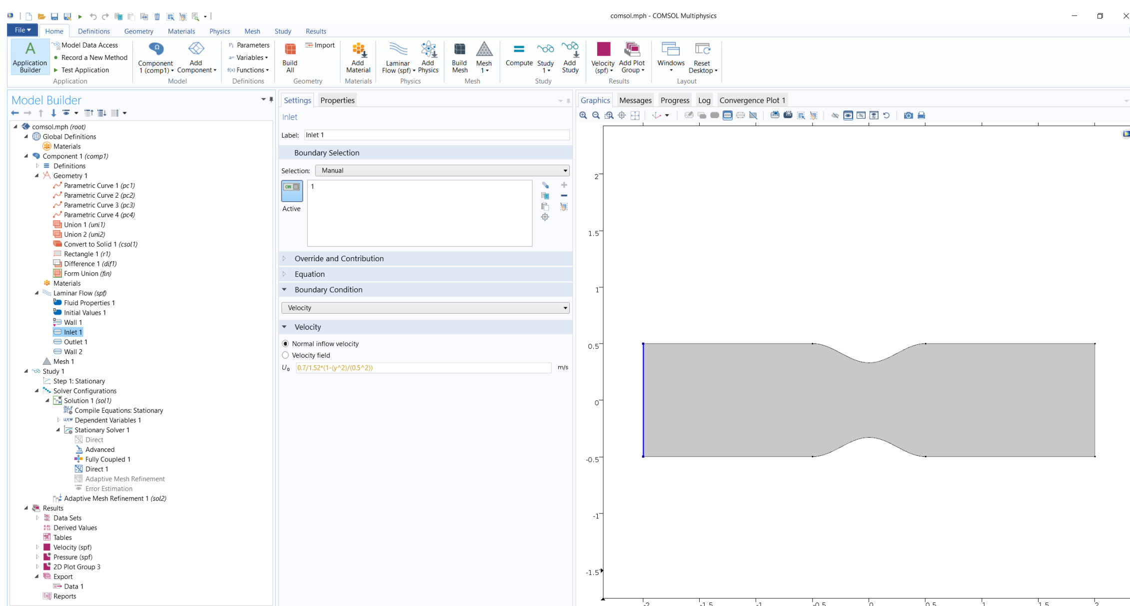


Figure 5.5: *Inlet parabolic velocity profile with a maximum value of 0.46 m/s.*

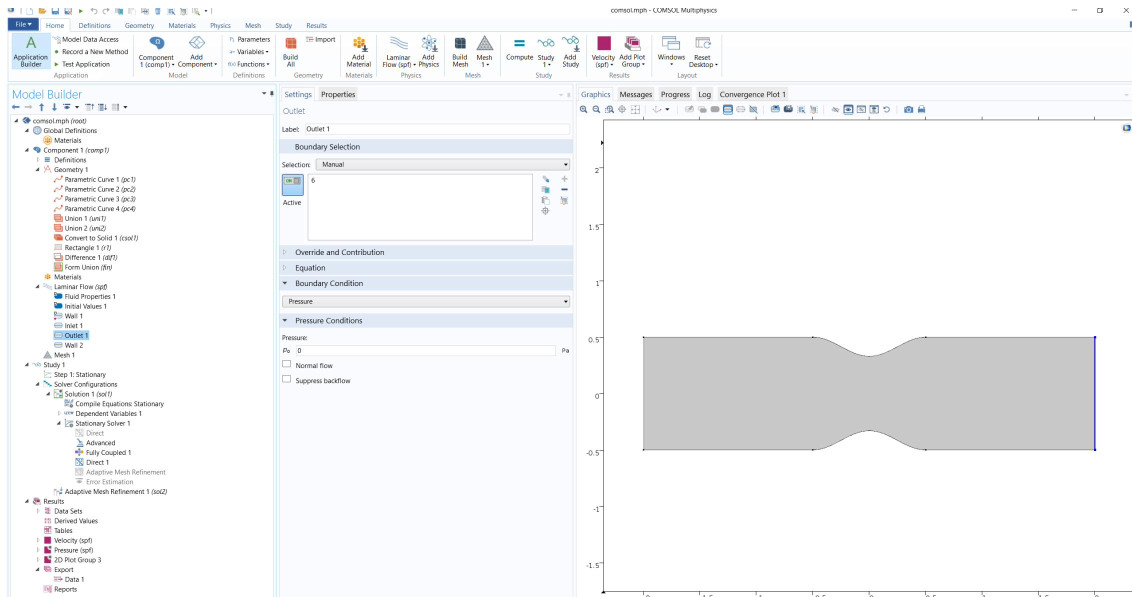


Figure 5.6: *Zero pressure condition at the outlet.*

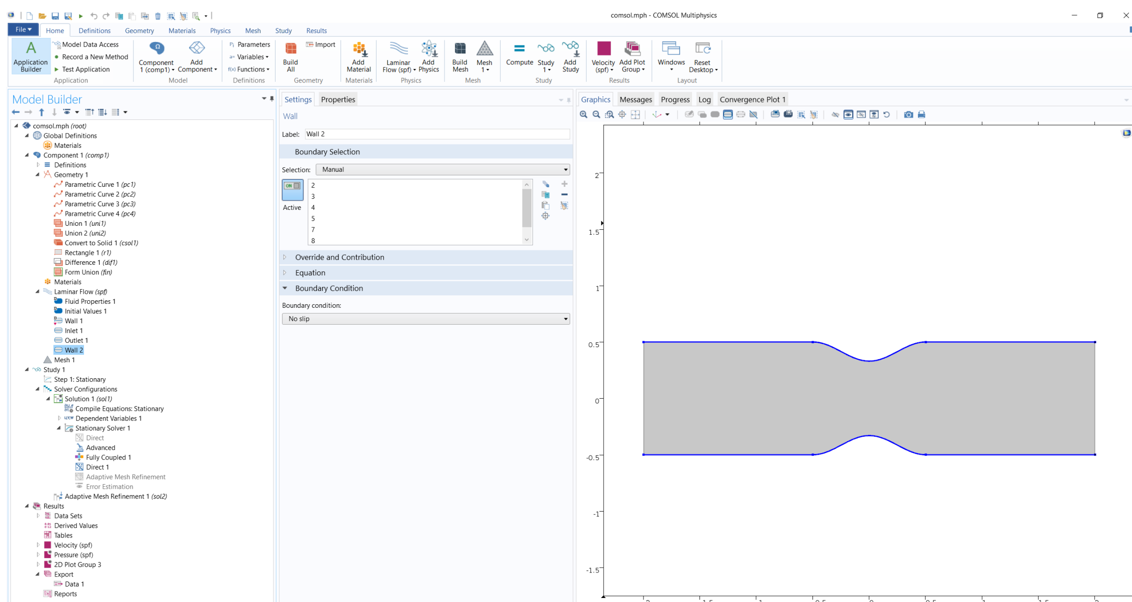


Figure 5.7: *No slip conditions at the walls.*

After the formulation of the constraints, the mesh was generated. COMSOL gives the option to simplify the process of creating a mesh by either allowing the software to do it automatically or by manually constructing a customized mesh. This is achieved using either a physics-controlled ora user-controlled mesh sequence type, accordingly. No matter which type of mesh sequence the user selects, there exist numerous options, settings, tools, and generators that can be employed to generate an ideal mesh for the given geometry and analysis.

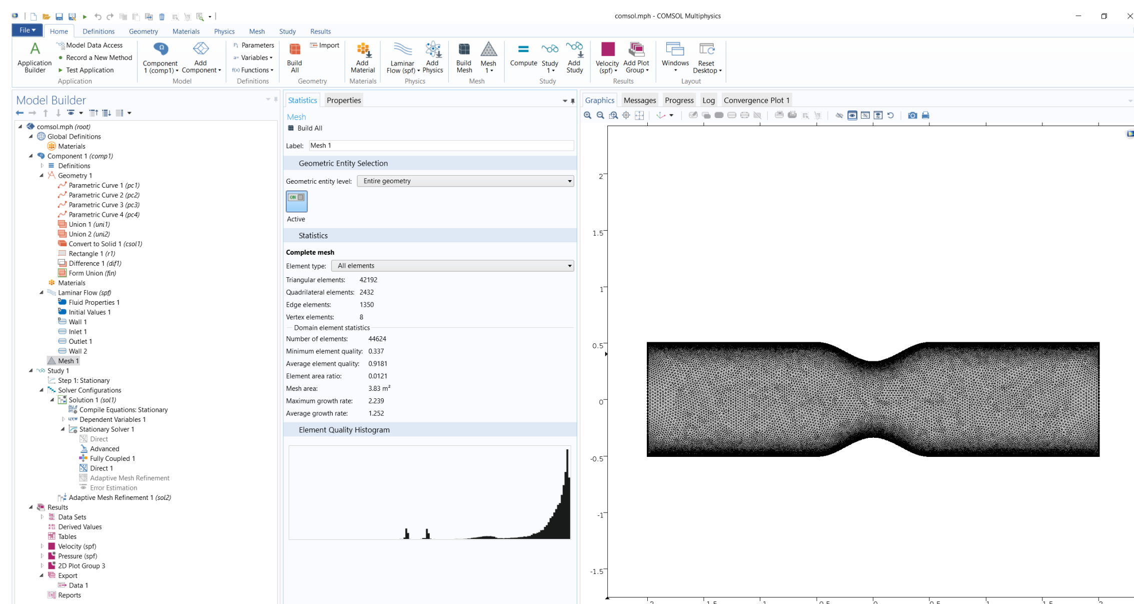


Figure 5.8: *Statistics of an extra fine physics-controlled mesh.*

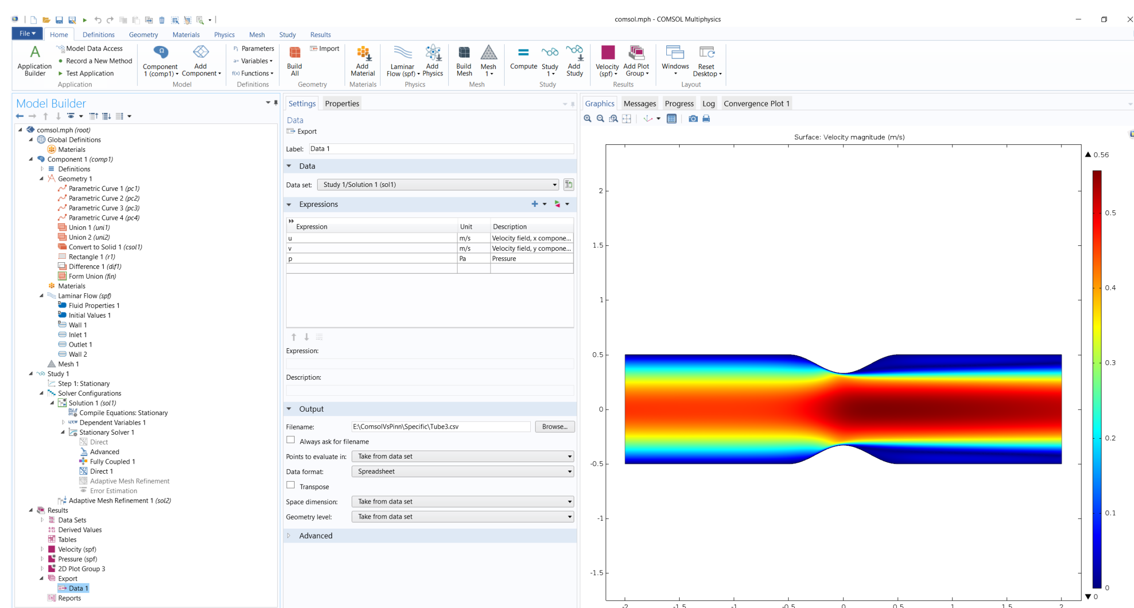


Figure 5.9: *Extraction of COMSOL data.*

The relative error tolerance was set to 10^{-4} and a mesh independence analysis was performed to reach the final accepted CFD solution. To compare the CFD and PINN solutions, we applied post-processing to transform the computational mesh grid points from COMSOL for proper PINN inference. Notably, the input parameters in the PINN model, aside from the other parameters, are the x and y coordinates of a channel without stenosis. Based on the fc parameter, the geometry is then transformed during inference to represent a stenotic vessel. Therefore, the COMSOL computational mesh, representing a vessel with a given stenosis severity, must first be transformed into a non-stenotic vessel before being submitted to the PINN model. This "antistenosis" transformation (see Appendix A) ensures that the comparison between CFD and PINN solutions is conducted using

consistent coordinates, obtaining more precise error measurements.

5.5 Evaluation of errors

To quantify the errors between the COMSOL values, $y^{\text{Com}} = [u, v, p]$,and PINN values, $y^{\text{PINN}} = [\hat{u}, \hat{v}, \hat{p}]$, we used normalized root mean squared error (Norm RMSE).

$$\text{Norm RMSE} = \frac{\text{RMSE}}{y^{\text{PINN}}_{max} - y^{\text{Com}}_{min}} \tag{5.17}$$

where RMSE is the root mean squared error,

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(y_i^{\text{PINN}} - y_i^{\text{Com}} \right)^2} \tag{5.18}$$

Norm RMSE is typically used to measure accuracy in regression models. It takes values from 0 to 1 and closer to zero means that the model is more accurate.

Part III

Results

Chapter 6

Study of different parameters

6.1 Advantages of tube specifc parameters

To test the performance of the PINN when employing TSPs, a network that consists of 6 hidden layers and 512 neurons per hidden layer was used. The activation function selected is SiLU and Adam algorithm is chosen as the optimizer.

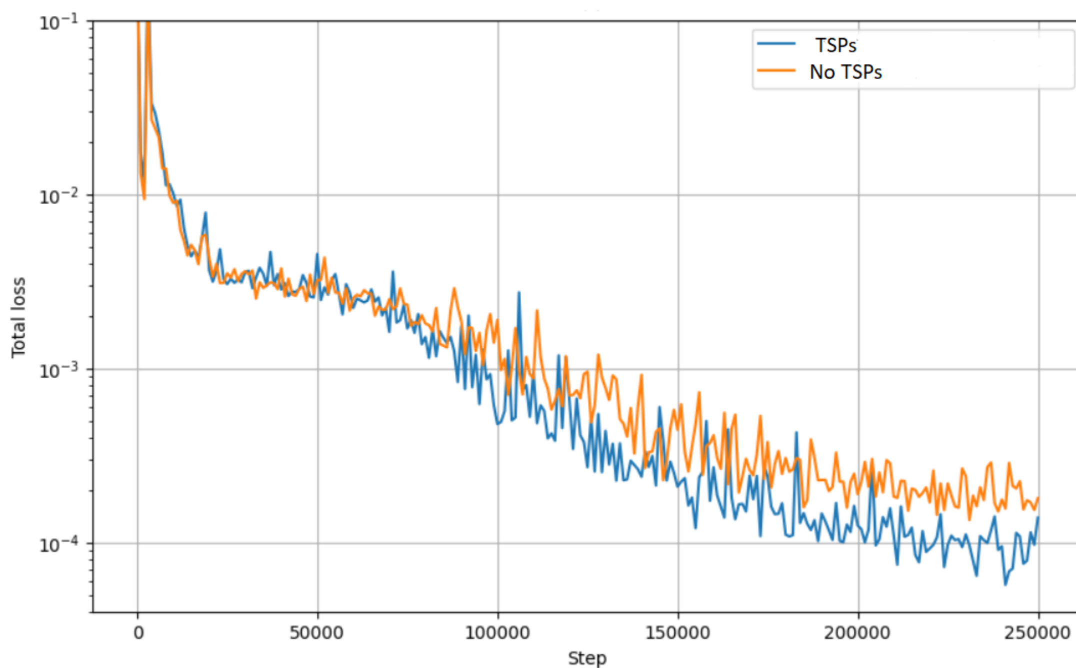


Figure 6.1: *Total loss for the 2 networks.*

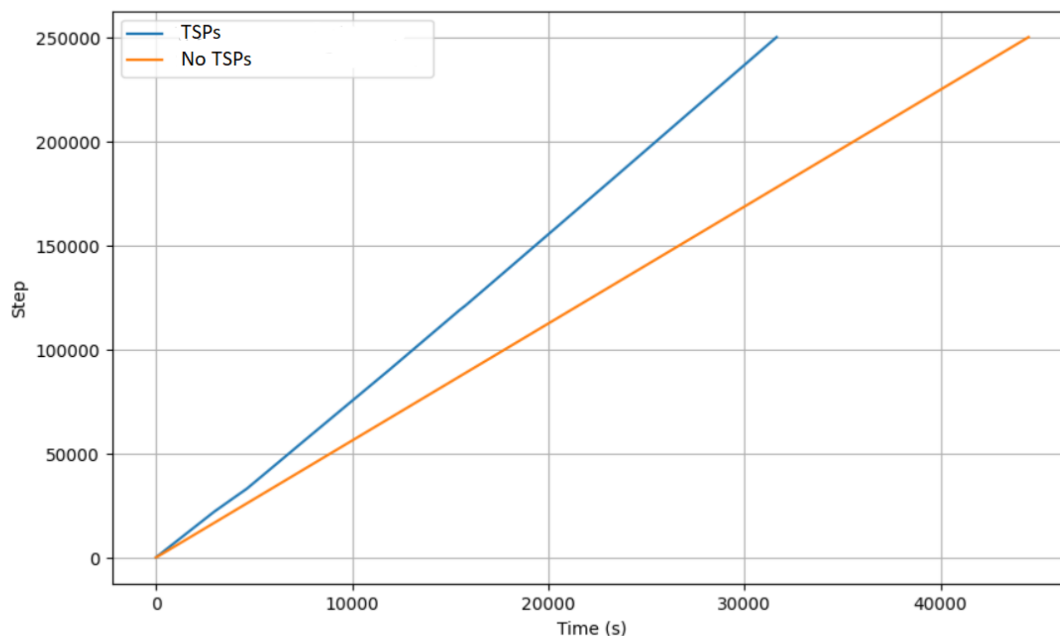


Figure 6.2: *Time per step for the 2 networks.*

As observed the employment of TSPs not only reduces the total loss (Fig. 6.1), but also lowers significantly the calculation time needed for each step (Fig. 6.2). More specifically the total loss achieved when employing TSPs is 70% lower compared to the network's loss without TSPs and the time per step is decreased from 0.18 seconds per step to 0.12 seconds per step. For the accuracy comparison, 3 random case studies were picked.

Table 6.1: *The parameters of the 3 case studies for the evaluation of the TSPs implementation errors.*

Study	fc	$u_{max}(m/s)$	Reynolds
1	0.24	0.62	816
2	0.11	1.1	1447
3	0.17	0.7	921

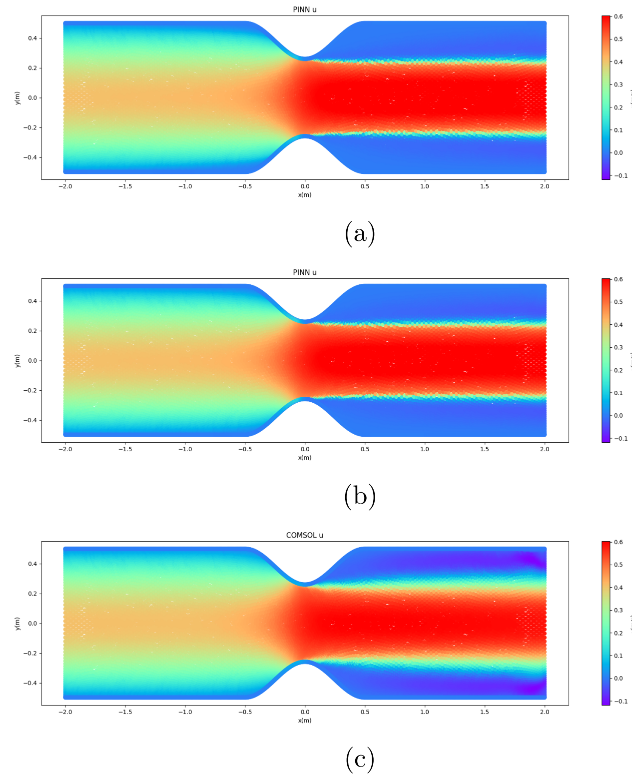


Figure 6.3: *Prediction of u velocity in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

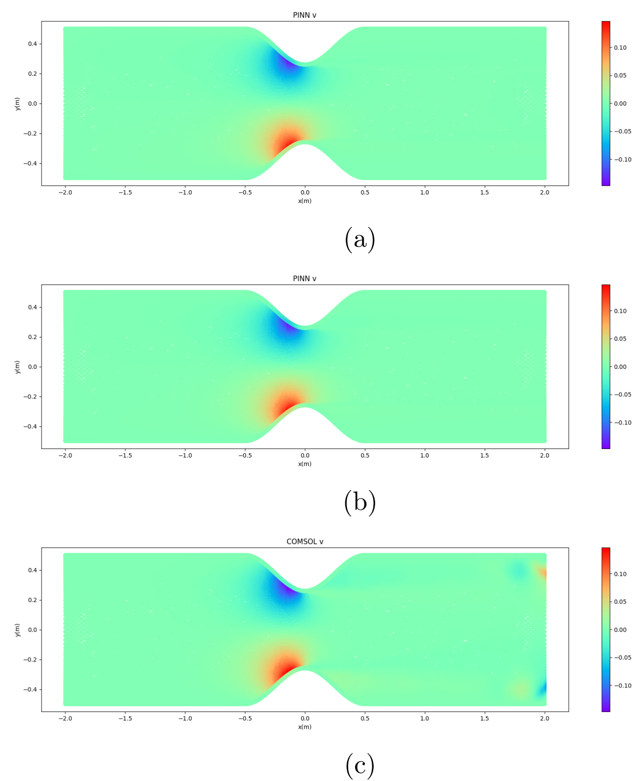
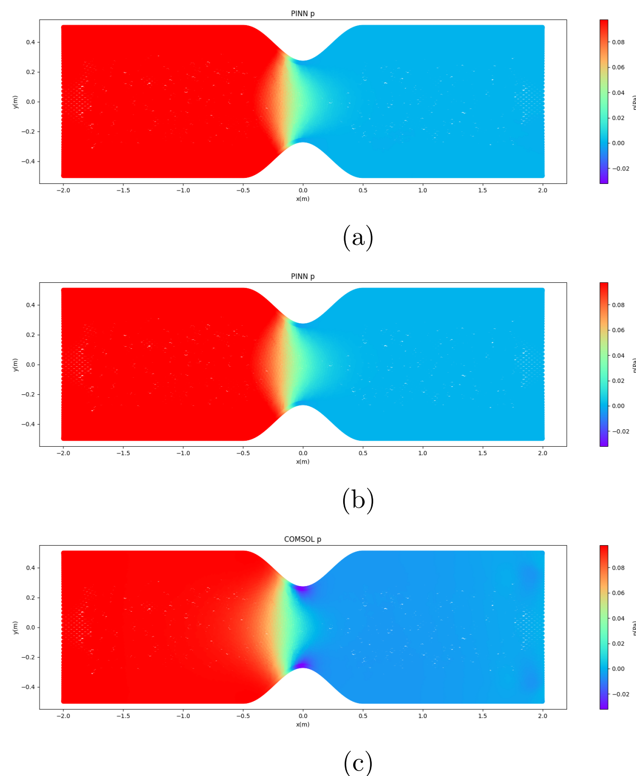


Figure 6.4: *Prediction of v velocity in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

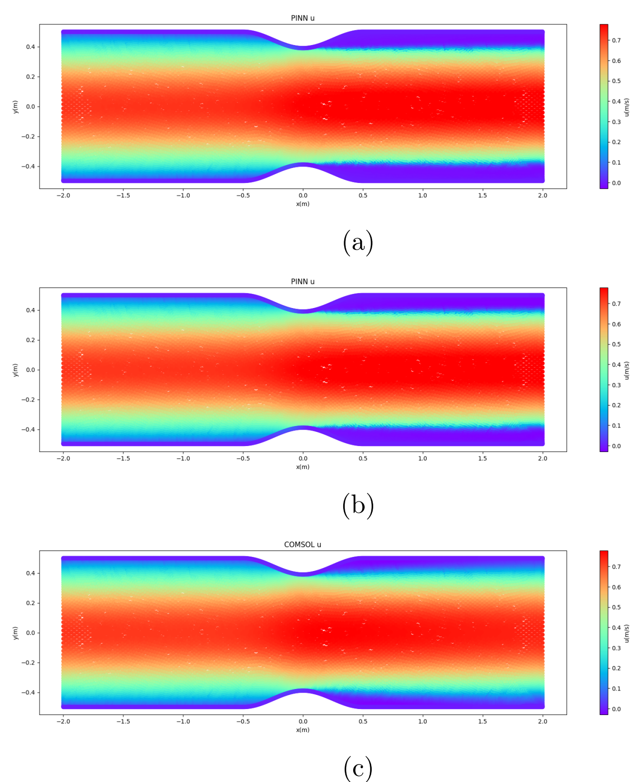


(a)

(b)

(c)

Figure 6.5: *Prediction of pressure in Study 1 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

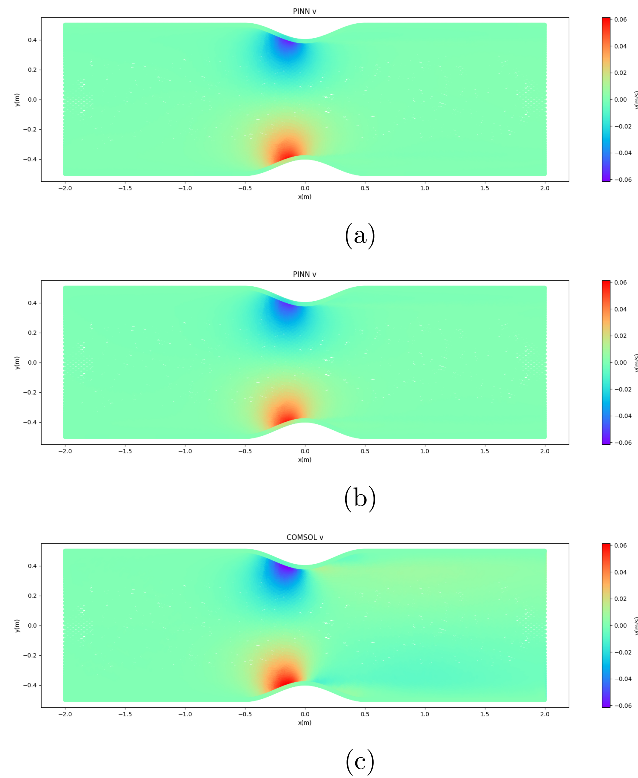


(a)

(b)

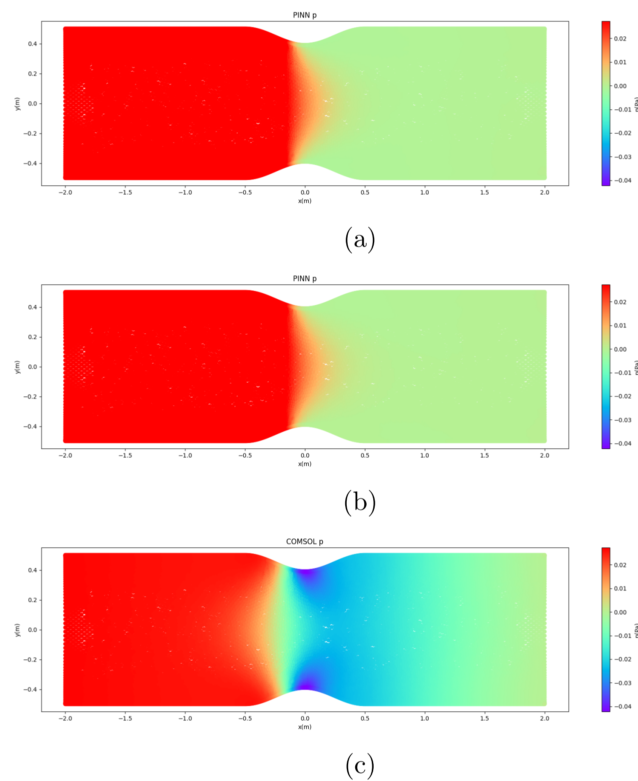
(c)

Figure 6.6: *Prediction of u velocity in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*



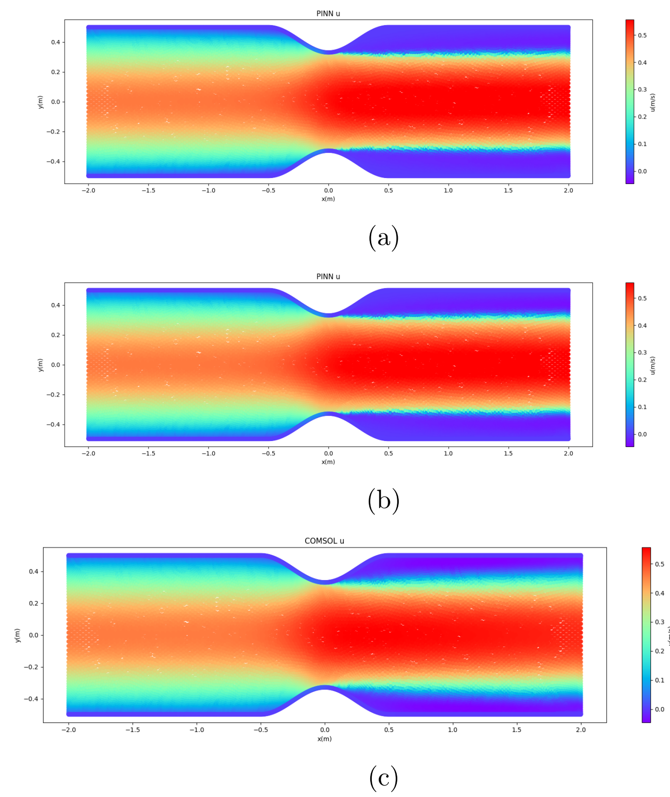
(a) (b) (c)

Figure 6.7: *Prediction of v velocity in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*



(a) (b) (c)

Figure 6.8: *Prediction of pressure in Study 2 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

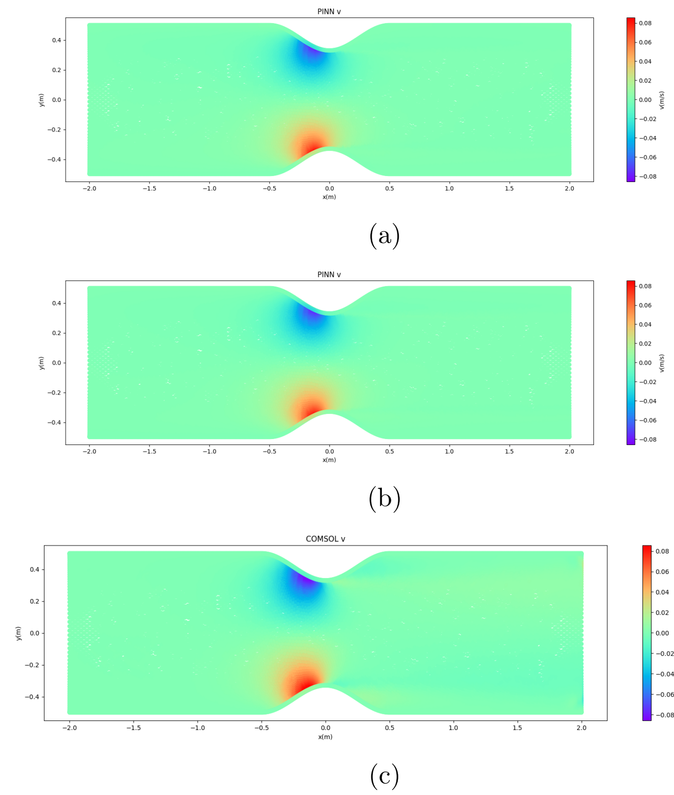


(a)

(b)

(c)

Figure 6.9: *Prediction of u velocity in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

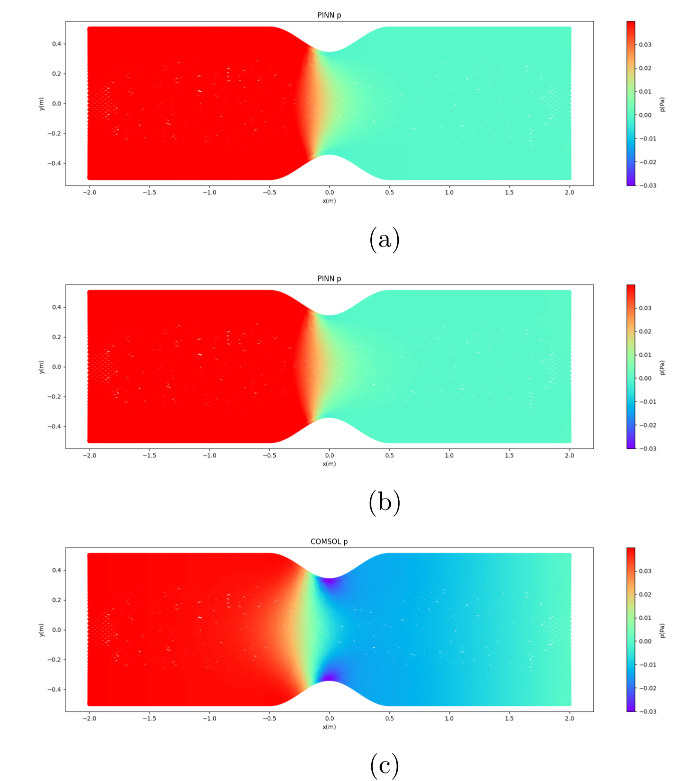


(a)

(b)

(c)

Figure 6.10: *Prediction of v velocity in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*



(a)

(b)

(c)

Figure 6.11: *Prediction of pressure in Study 3 (Table 6.1) using: (a) PINN without TSPs, (b) PINN with TSPs, (c) FEM in COMSOL.*

Table 6.2: *Errors for the 2 networks*

Study	Network	RMSE u	Norm RMSE u	RMSE v	Norm RMSE v	RMSE p	Norm RMSE p
1	TSPs	0.0354	0.0491	0.007	0.0236	0.0111	0.0855
1	NO TSPs	0.0367	0.0509	0.0067	0.0229	0.0125	0.0961
2	TSPs	0.0366	0.0453	0.0041	0.0331	0.0256	0.3694
2	NO TSPs	0.0363	0.0449	0.0040	0.0322	0.0258	0.3717
3	TSPs	0.0280	0.0465	0.0042	0.0244	0.0143	0.2032
3	NO TSPs	0.0289	0.042	0.00247	0.0249	0.0149	0.2123

The error measurements summarized in Table 6.2 clearly demonstrate that the use of TSPs significantly reduces the Norm RMSE of pressure when comparing the PINN and CFD solutions across all three case studies (Table 6.1), indicating an improvement in pressure accuracy. However, the impact on velocity errors is minimal, as these errors are already quite low. Given the reduced computational time per step and the improvement in pressure accuracy, TSPs are utilized in all subsequent studies in this thesis.

6.2 Activation functions

The selection of the activation function has a substantial influence on the effectiveness of training. When training using the backpropagation algorithm, it is necessary to compute the derivative of the loss function with respect to the weights and biases of each layer. As the studies have shown [13], training a neural network with multiple hidden layers may not be effective if the derivative of the activation function has a small range.

When it comes to Modulus Sym, SiLU is the default activation function, yet in this Thesis, we employed and evaluated the effects of the other activation functions discussed in section 1.2 on the rate of convergence. All the experiments were performed using Adam optimizer and a PINN with 4 hidden layers and 256 neurons per layer. It should be highlighted, that the training was discontinued when the total loss was stabilized, which is why certain trainings present with a lower number of epochs.

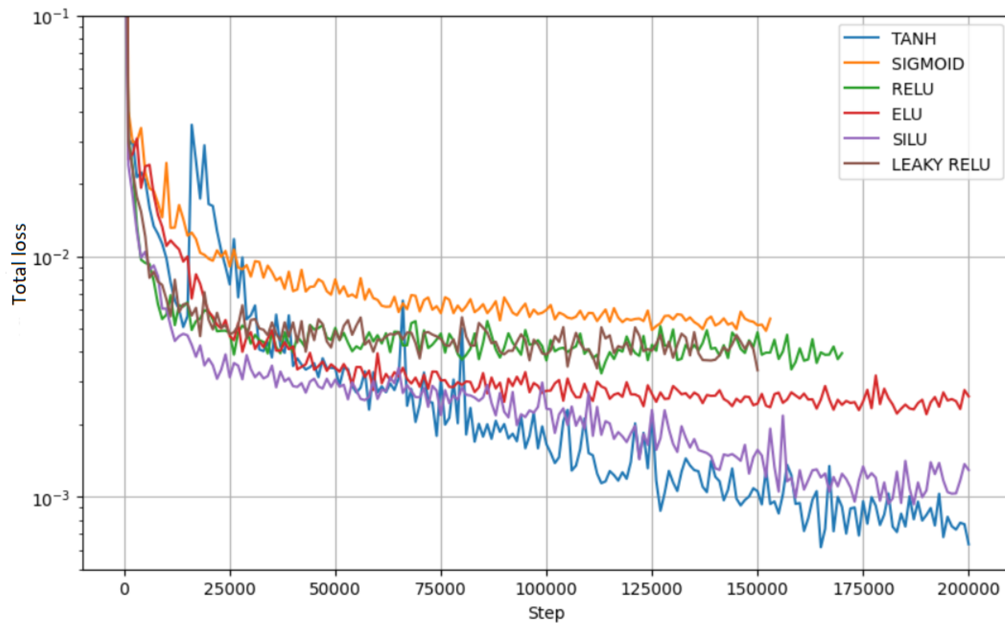


Figure 6.12: *Total loss for different activation functions.*

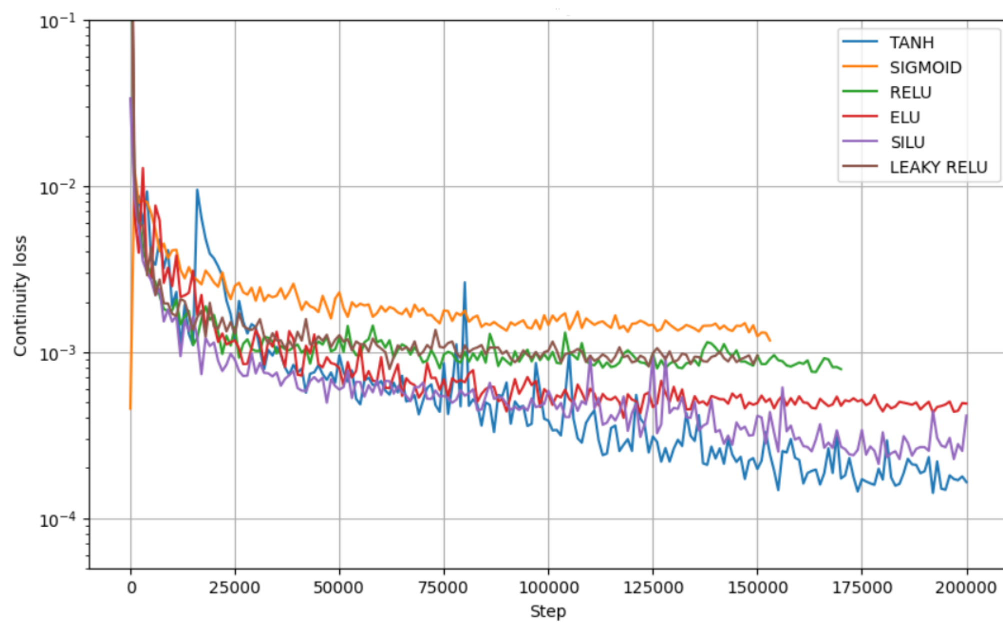


Figure 6.13: *Continuity loss for different activation functions.*

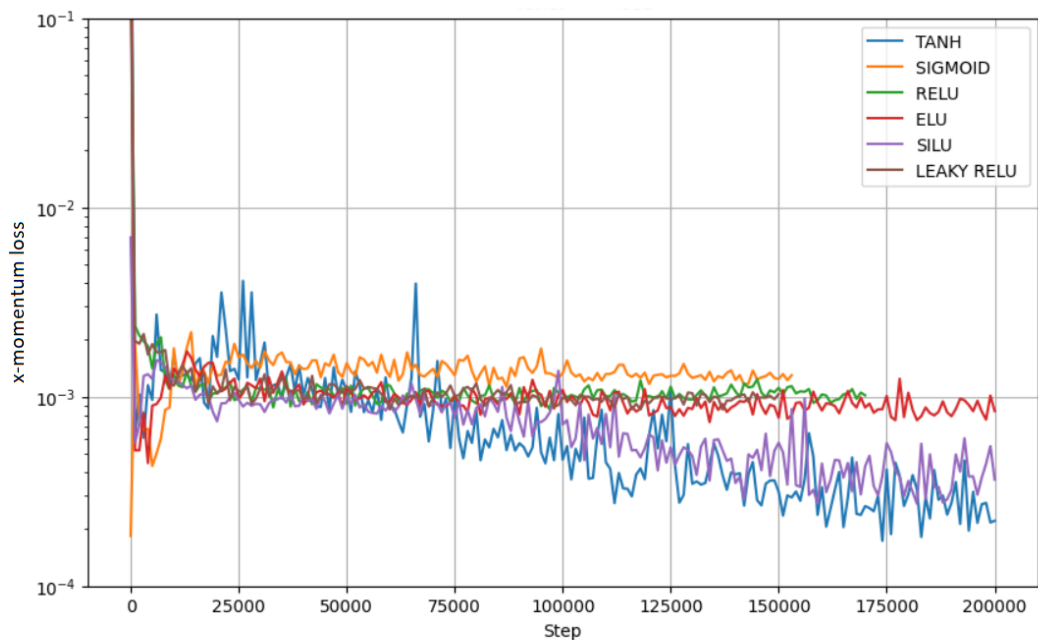


Figure 6.14: *x-momentum loss for different activation functions.*

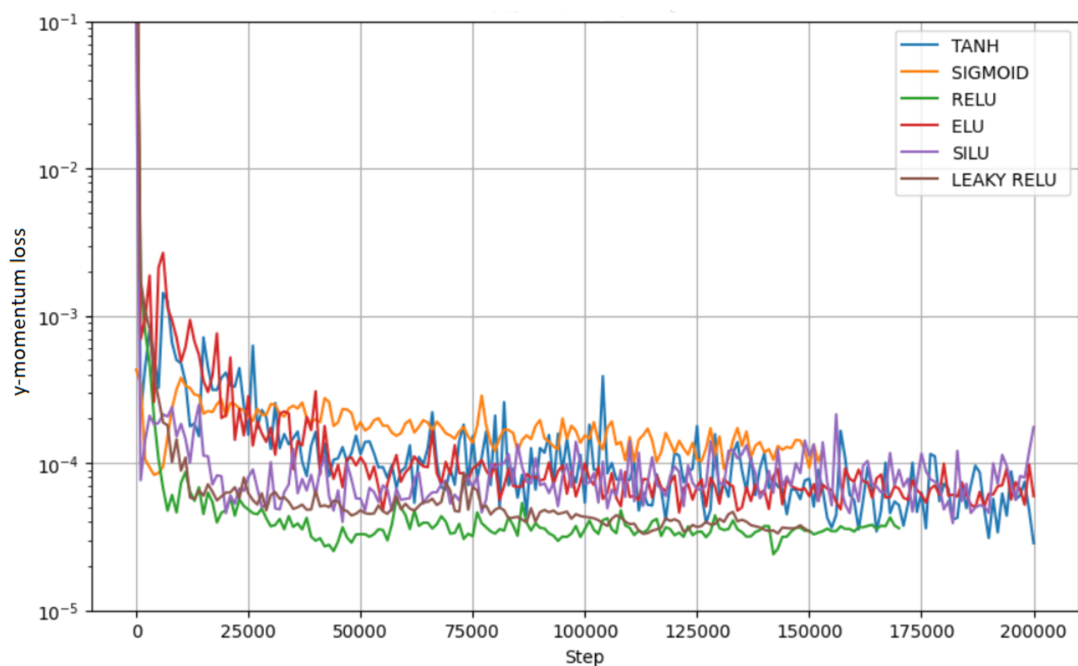


Figure 6.15: *y-momentum loss for different activation functions.*

Based on the data presented above it is evident that SiLU and tanh exhibit lower overall loss compared to the other AFs while Sigmoid possesses the largest total losses.

6.3 Number of layers and number of neurons per layer

In this section, we analyze how the the various combinations of number of layers and neurons per layer affect the total loss and computation time for each run. Again the optimizer used was the Adam with SiLU activation function. Firstly, we increased the number of neurons per layer while keeping a fixed layer size.

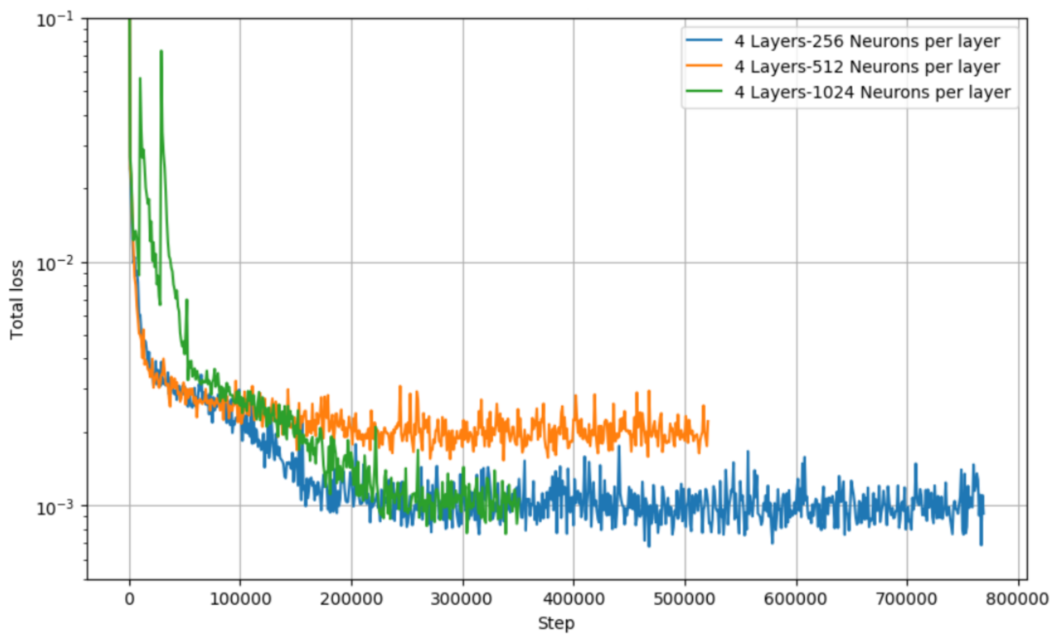


Figure 6.16: *Total loss for different numbers of neuron per layer.*

It is evident that the architectures of 256 neurons per layer and 1024 neurons per layer perform slightly better than the network with 512 neurons per layer. However if we analyze the plot of total loss with respect to wall clock time (Fig. 6.17), it is obvious that the architecture with 256 neurons per layer achieve the same total loss levels much faster.

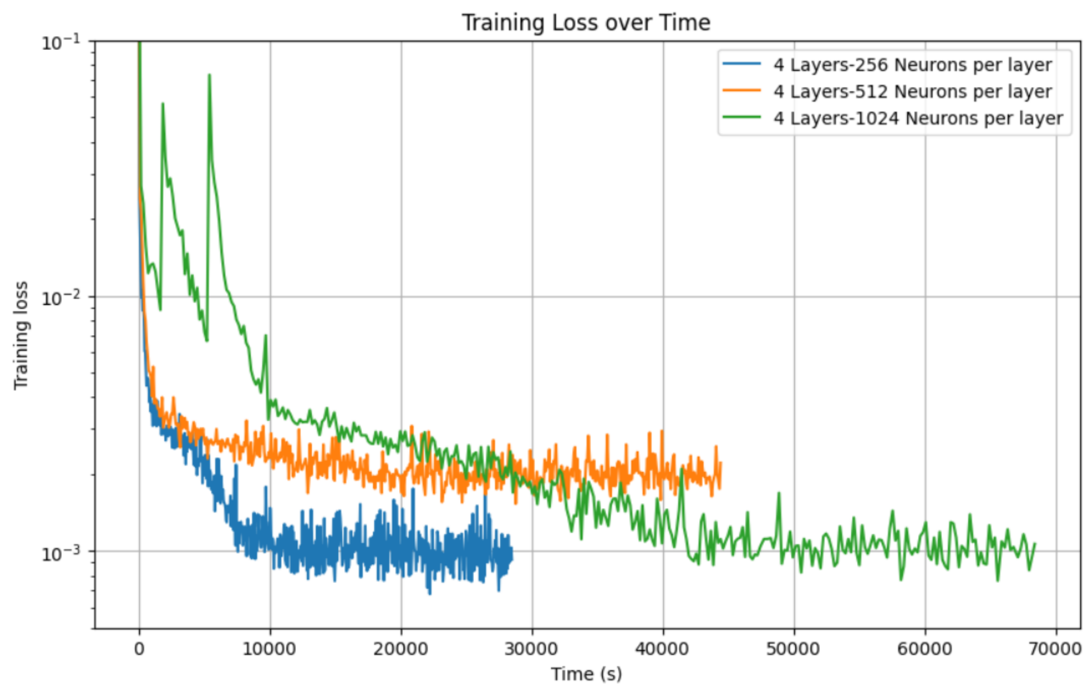


Figure 6.17: *Total loss for different numbers of neuron per layer with respect to wall clock time.*

To test how the number of layers influence the performance of the PINN, we increased the number of layers, while maintaining a constant number of neurons per layer.

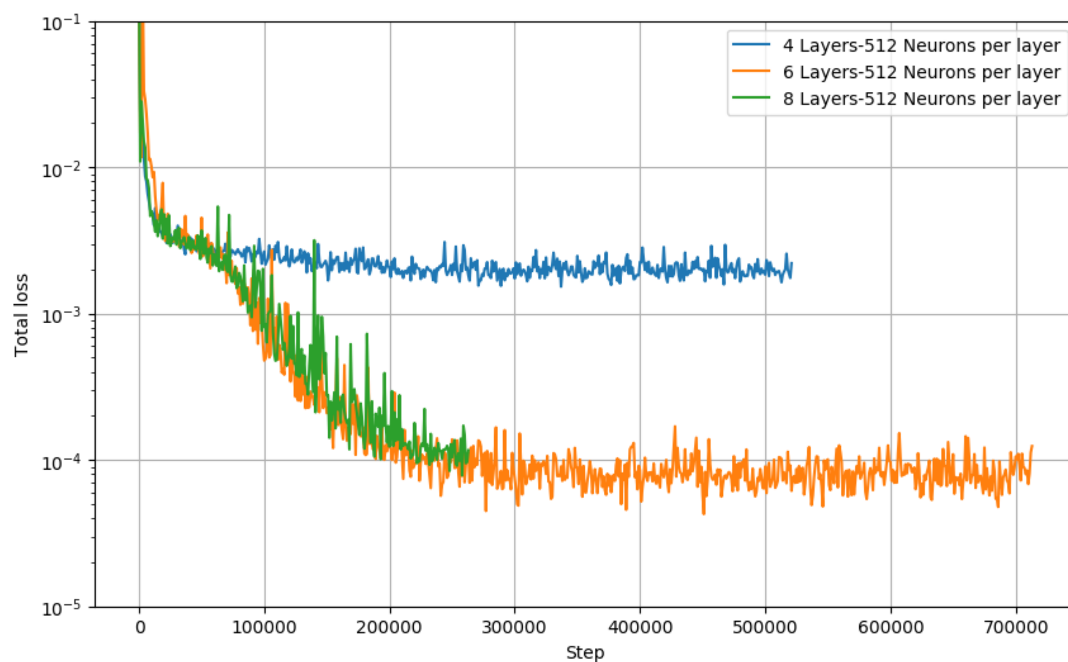


Figure 6.18: *Total loss for different number of layers.*

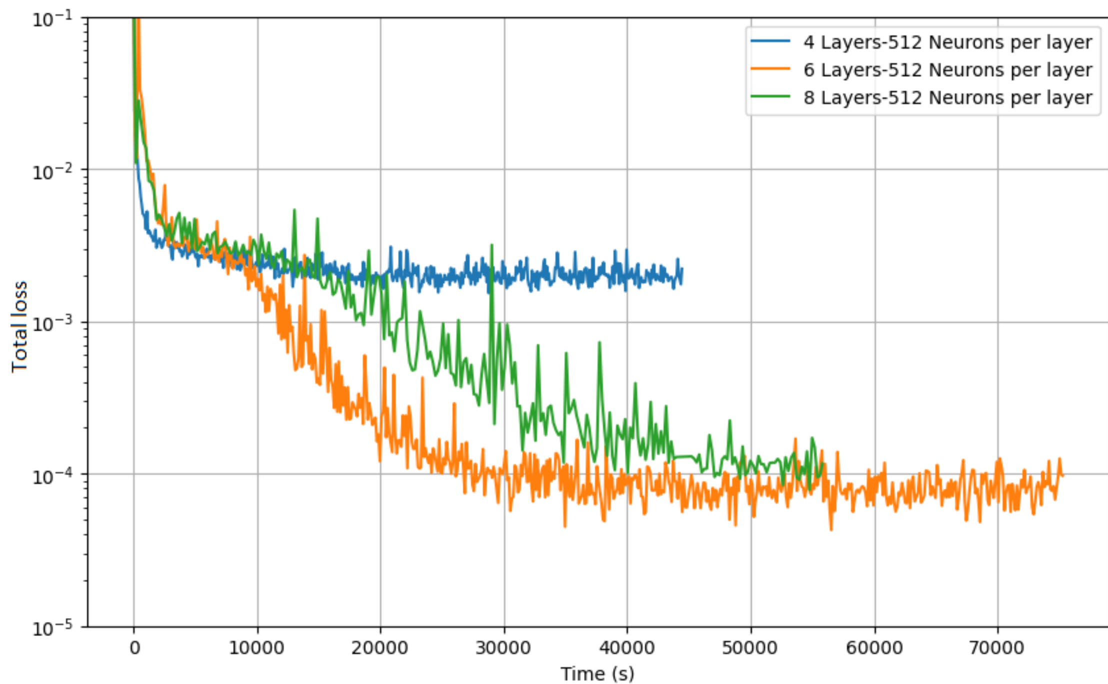


Figure 6.19: *Total loss for different number of layers with respect to wall clock time.*

As it can be observed in Figs 6.26 and 6.19, the network with the 4 layers does not manage to reach a satisfying loss level. Additionally we can observe that the most efficient architecture is the one with the 6 layers, as it achieves the lowest total loss in the least amount of time.

Comparison between all different architectures

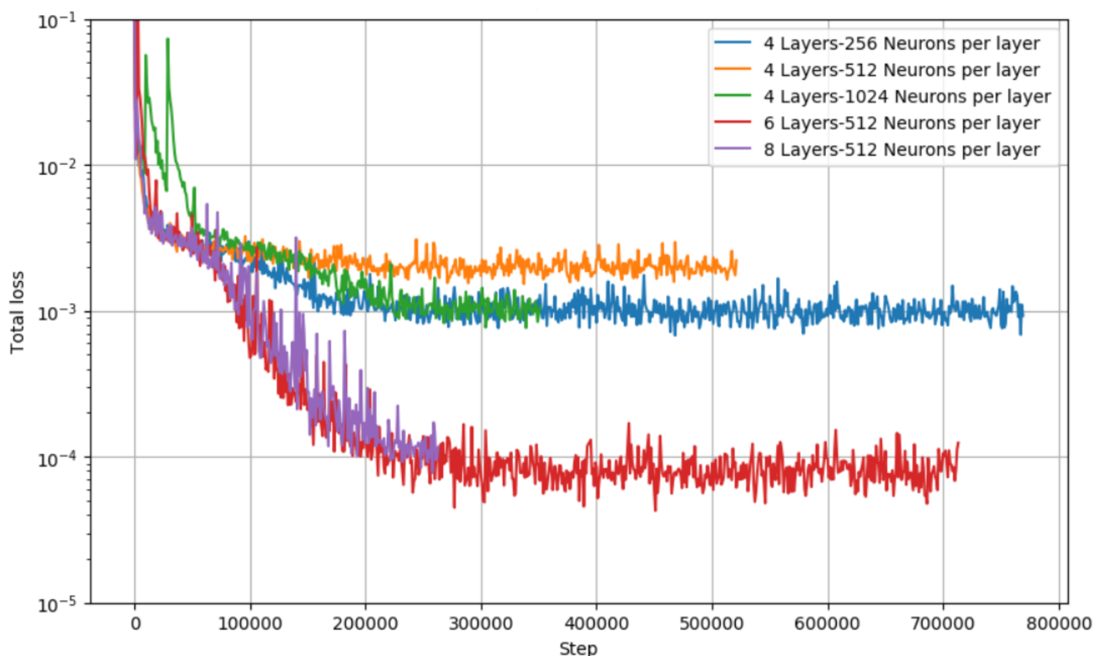


Figure 6.20: *Total loss for the different architectures.*

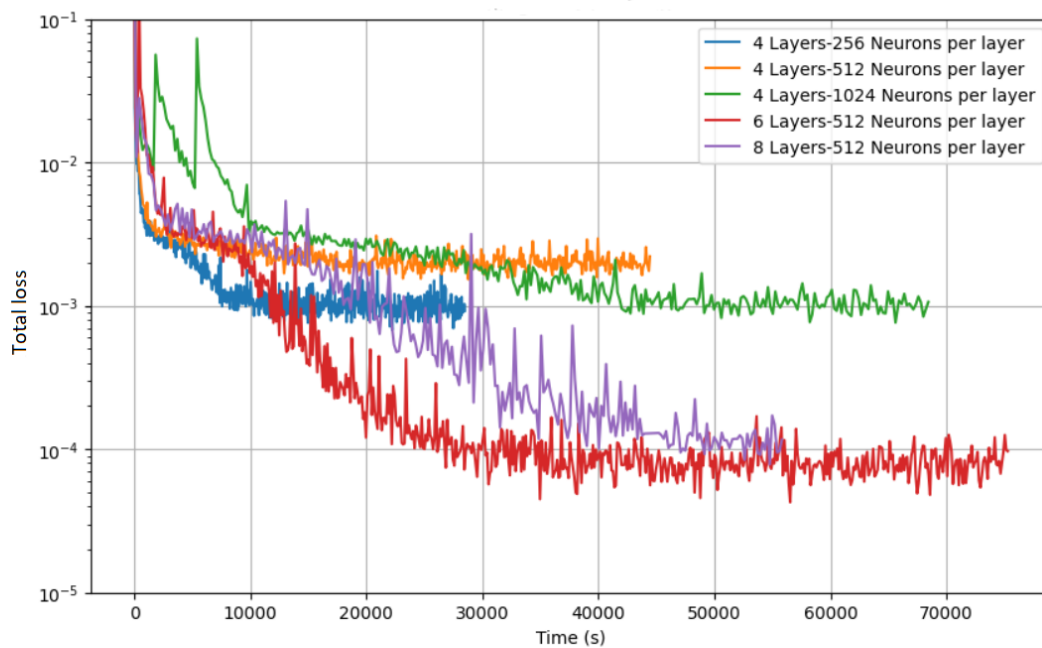


Figure 6.21: *Total loss for the different architectures with respect to wall clock time.*

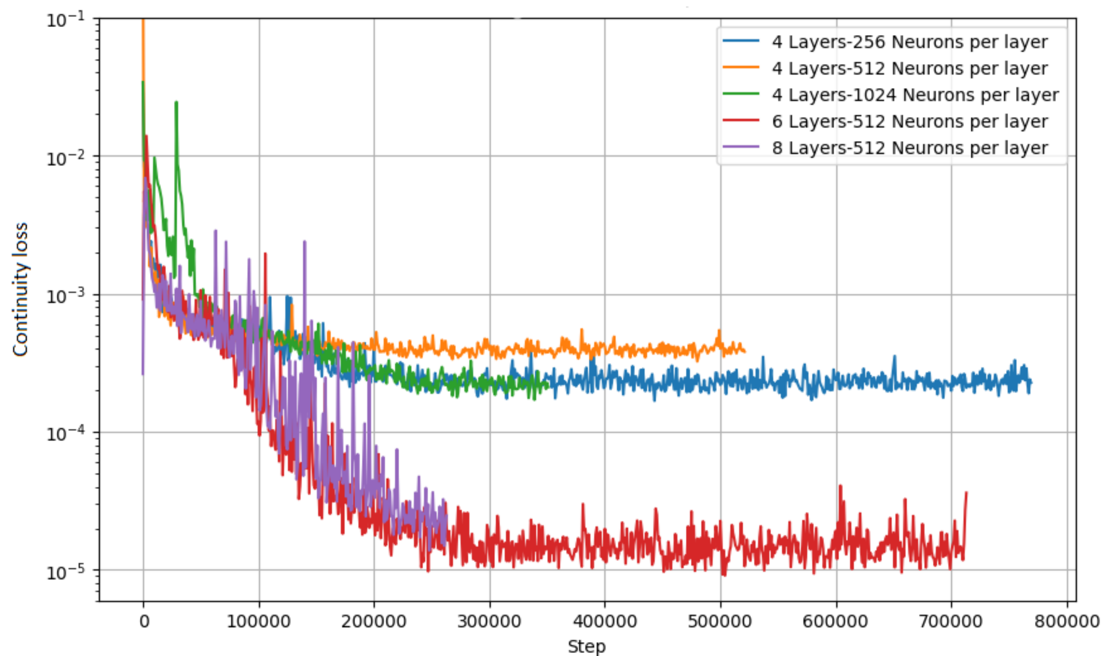


Figure 6.22: *Continuity loss for the different architectures.*

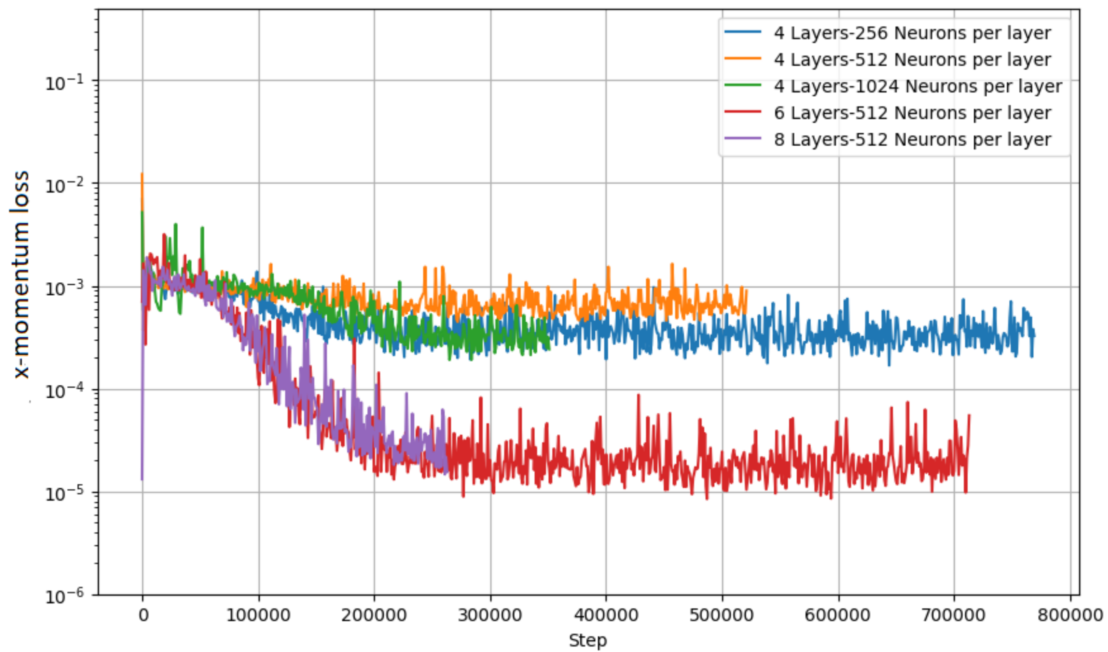


Figure 6.23: *x-momentum loss for the different architectures.*

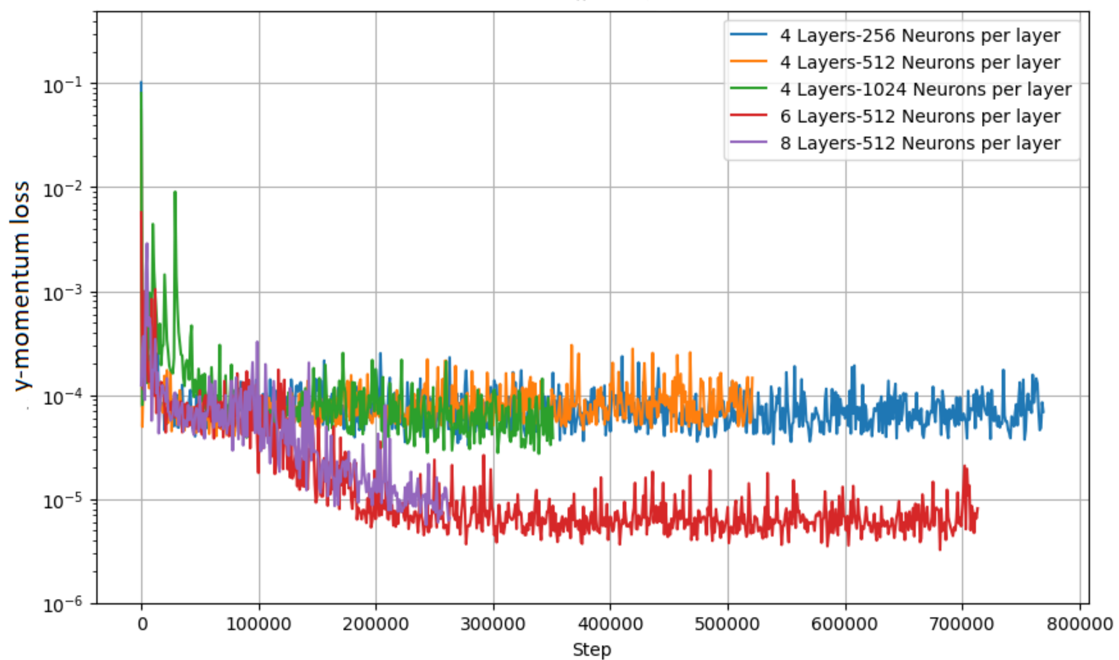


Figure 6.24: *y-momentum loss for the different architectures.*

The network with the 6 hidden layers and 512 neurons per layer manages to achieve the lowest total loss and the lowest individual losses. Therefore, we chose this architecture to evaluate the accuracy of our trained model in comparison with the CFD validation data provided by COMSOL.

6.4 Optimizers

In this section we investigated how various optimizers influence the different losses in our model. The optimizers are selected with their default hyperparameters from Nvidia Modulus Sym and the network deployed, is a PINN with 4 hidden layers and 256 neurons per layer. The selected activation function is SiLU.

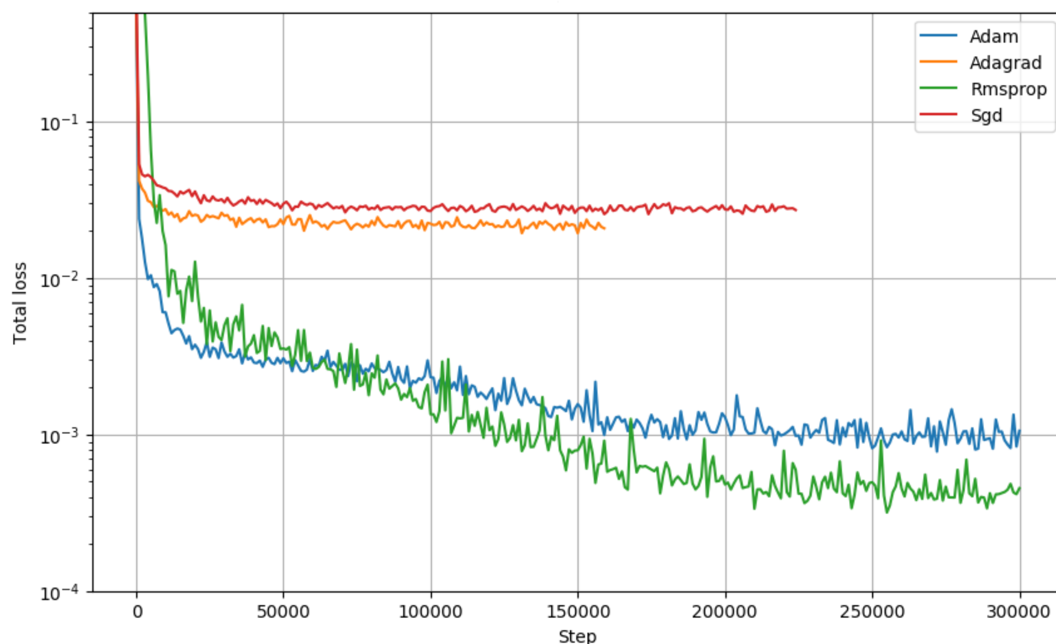


Figure 6.25: *Total loss for different optimizers.*

It appears that SGD and AdaGrad get stuck in a local minimum and do not manage to reach a lower total loss level (Fig 6.25). Moreover we observe that RMSProp surprisingly succeeds in obtaining better convergence rate than Adam but with higher computation time per step. The time required for each step in the Adam, AdaGrad, and SGD optimization algorithms is nearly the same (Fig 6.26).

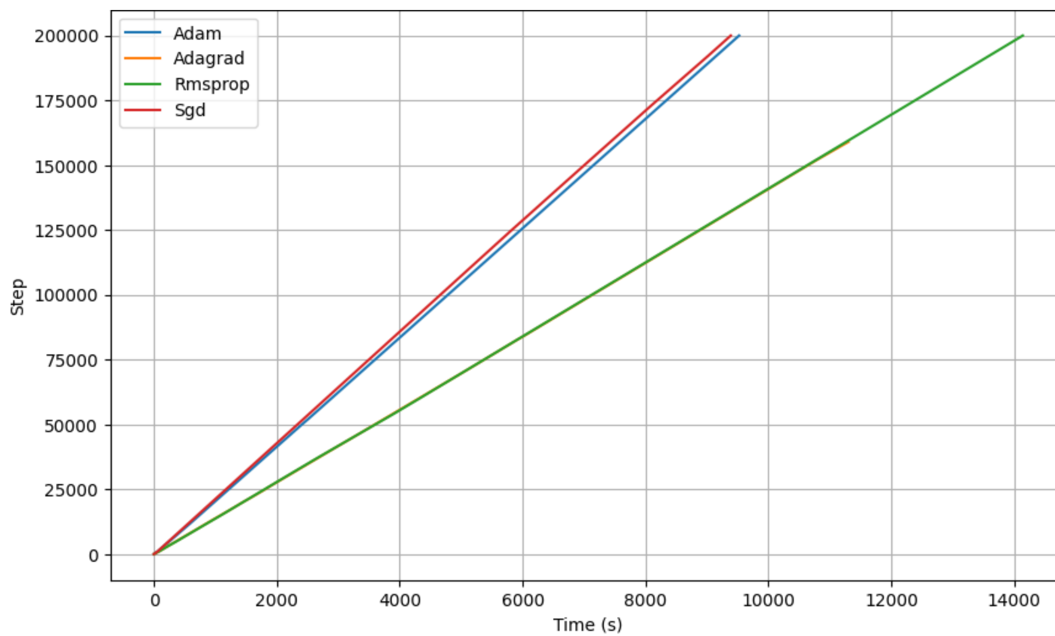


Figure 6.26: *Time per step for different optimizers.*

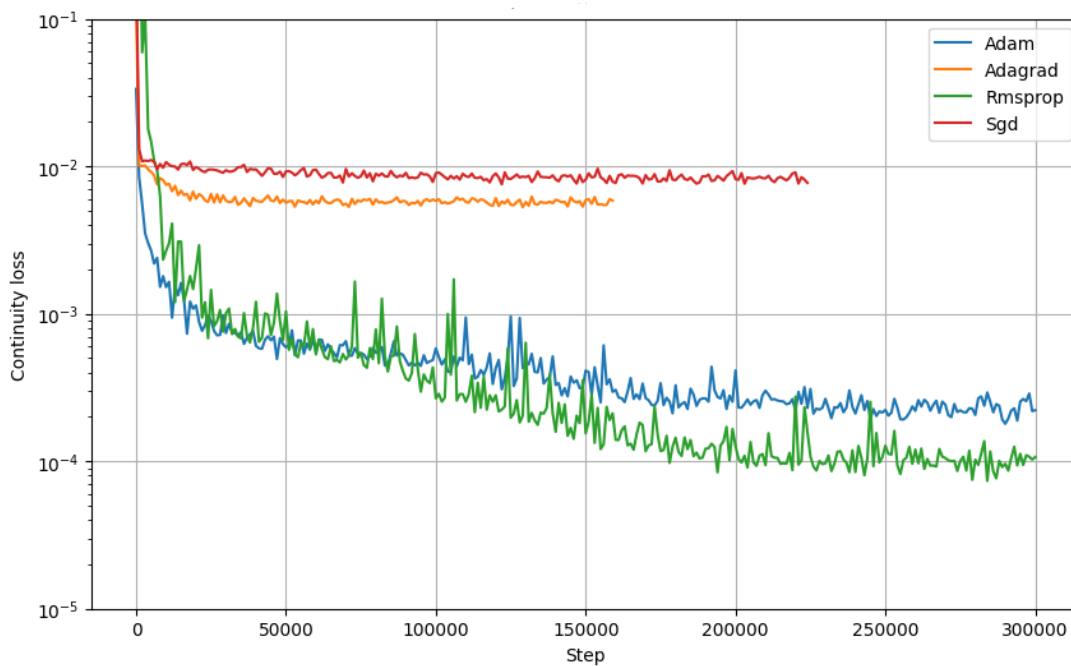


Figure 6.27: *Continuity loss for different optimizers.*

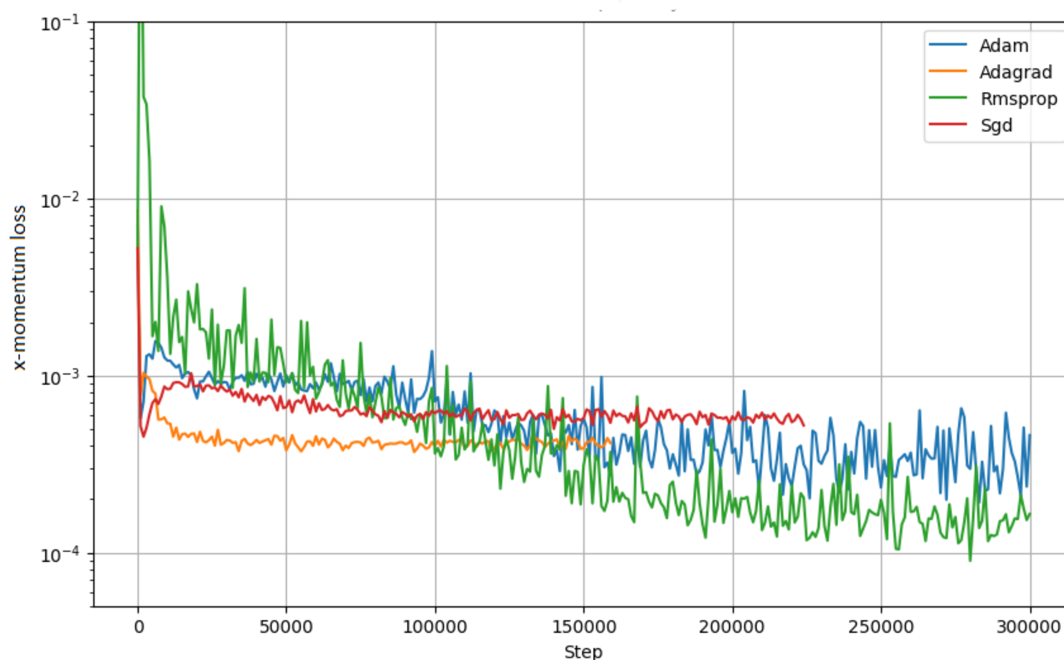


Figure 6.28: *x-momentum loss for different optimizers.*

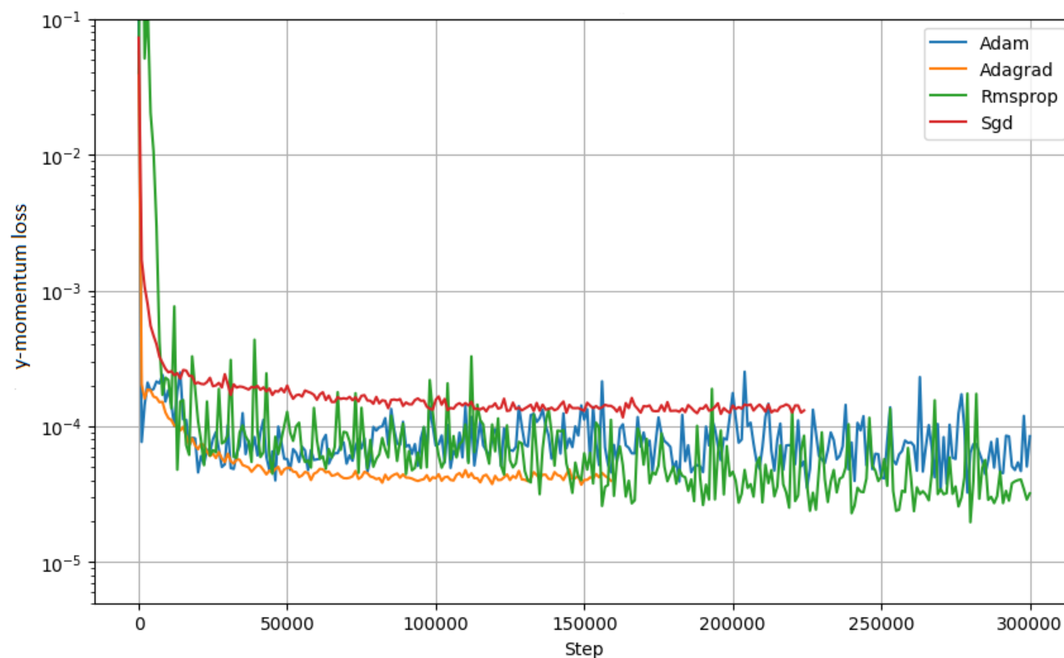


Figure 6.29: *y-momentum loss for different optimizers.*

It is noticeable that SGD and AdaGrad have almost no fluctuations in their loss values compared to the other 2 optimizers.

6.5 Adaptive activation functions

To investigate how adaptive activation functions influence the accuracy and the convergence, we tested the performance of 2 neural networks architectures with and without adaptive activation functions. The architectures selected are one network with 4 hidden layers and 256 neurons per layer and one network with 6 hidden layers and 512 neurons per layer. All networks are trained with Adam optimizer and use SiLU as the activation function.

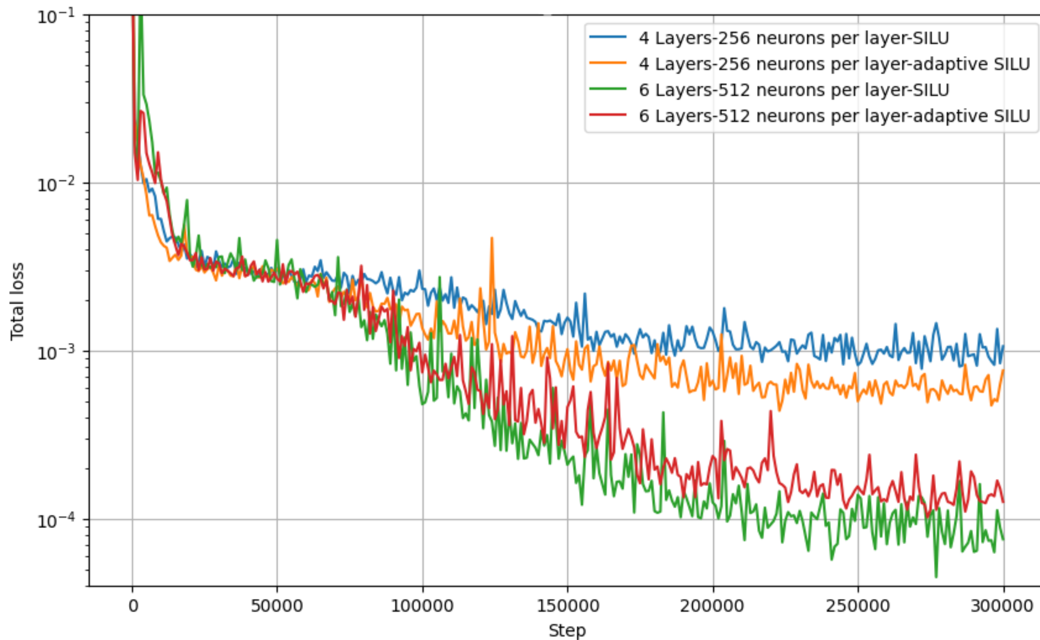


Figure 6.30: *Influence of adaptive activation functions.*

It is evident that using adaptive activation functions does not always reduce the total loss. To be more precise, enabling adaptive activation functions effectively reduces the total training loss for the network with 4 layers and 256 neurons per layer. However, in the larger network with 6 layers and 512 neurons per layer, it not only fails to reduce the loss but actually increases it.

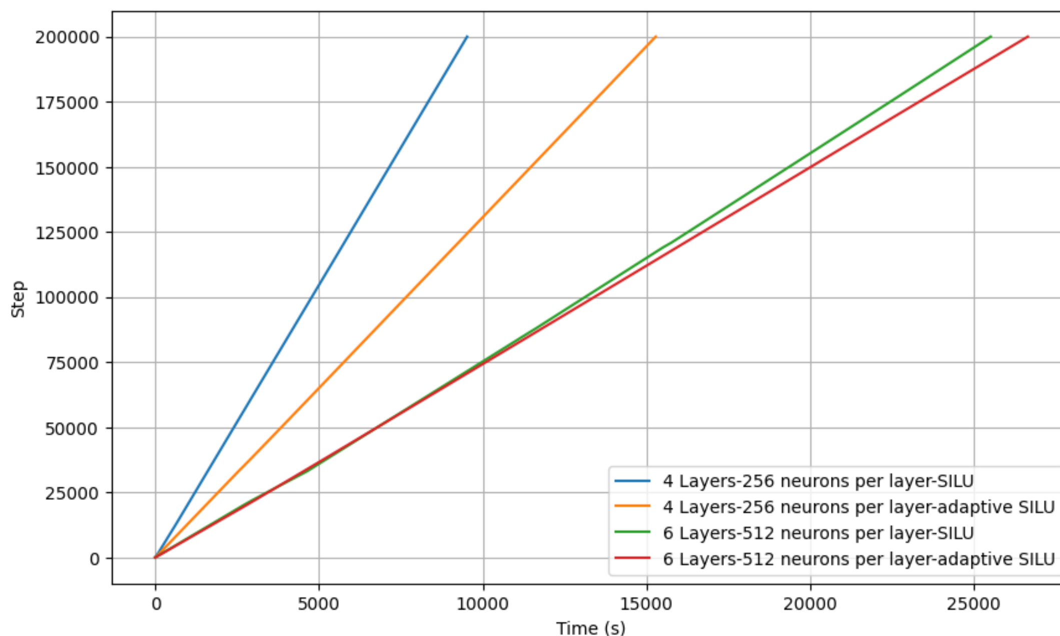


Figure 6.31: *Influence of adaptive activation functions on time per step.*

As it was expected the utilization of adaptive activation functions leads to an increase in the duration of each step. However the small network experiences much bigger rise in time per step, in comparison to the bigger network that the increase in time per step is only noticeable after the 100k steps.
For the accuracy test we picked 3 random pairs of fc and u_{max}.

Table 6.3: *The parameters of the 3 case studies for the accuracy evaluation of the adaptive activation functions.*

Study	fc	u_{max}(m/s)	Reynolds
1	0.256	1.111	1462
2	0.212	0.389	512
3	0.162	0.925	1217

Accuracy of small network (4 layers-256 neurons per layer)

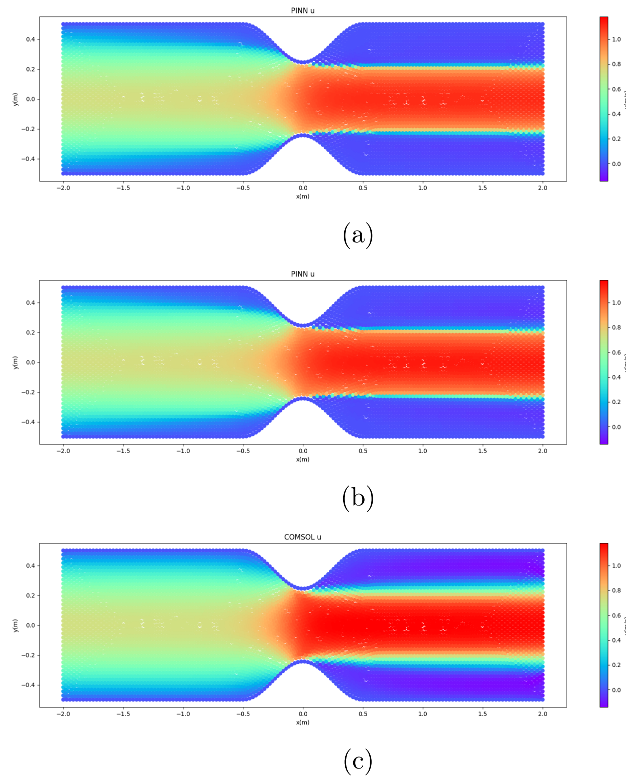


Figure 6.32: *Prediction of u velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

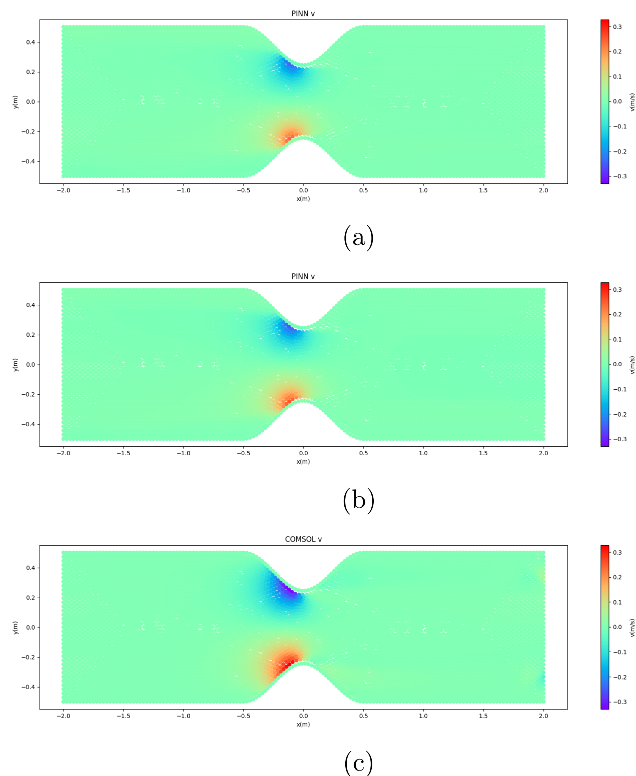


Figure 6.33: *Prediction of v velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

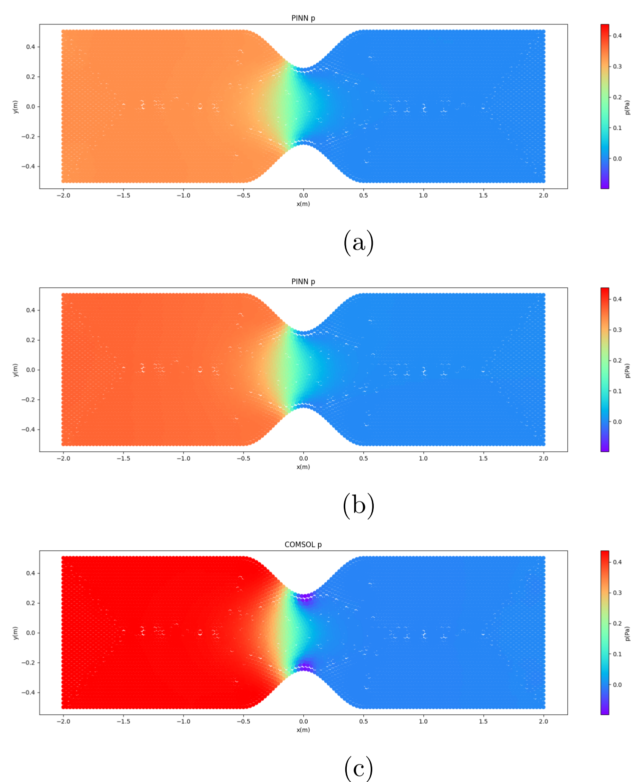


Figure 6.34: *Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

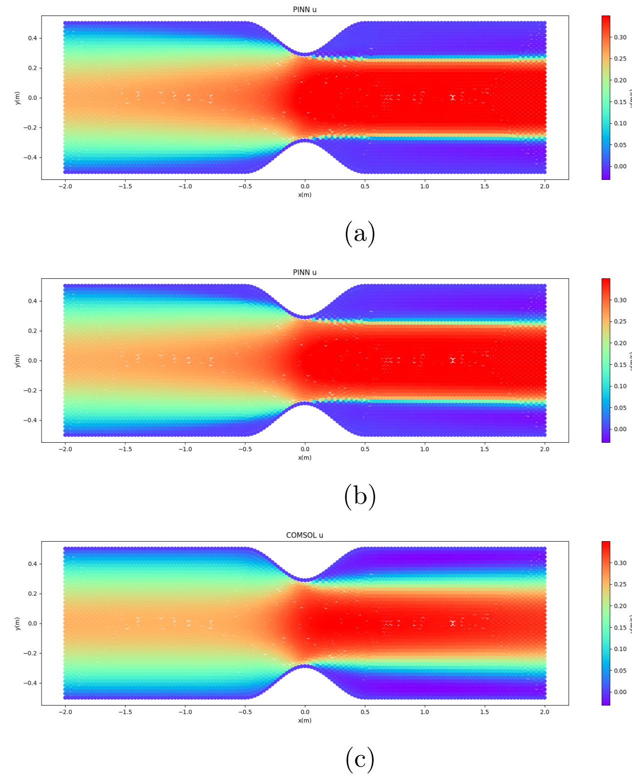


Figure 6.35: *Prediction of u velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

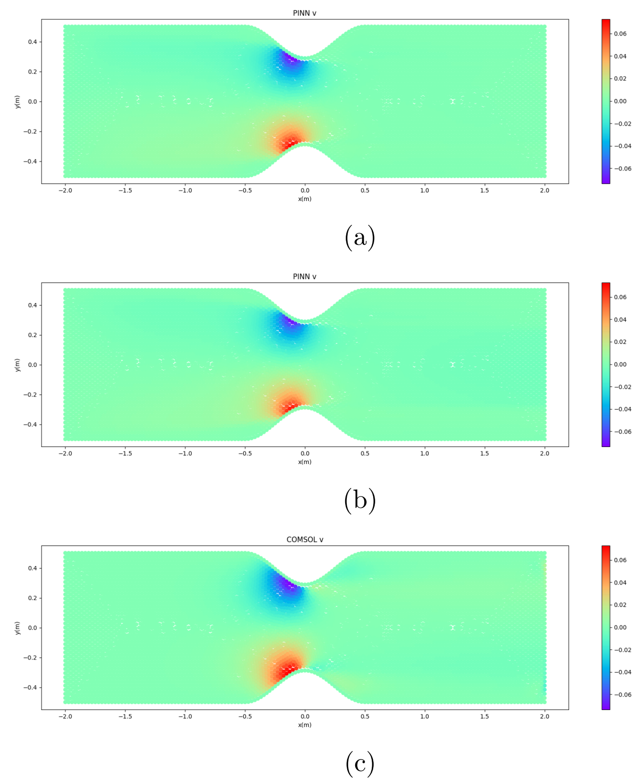


Figure 6.36: *Prediction of v velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

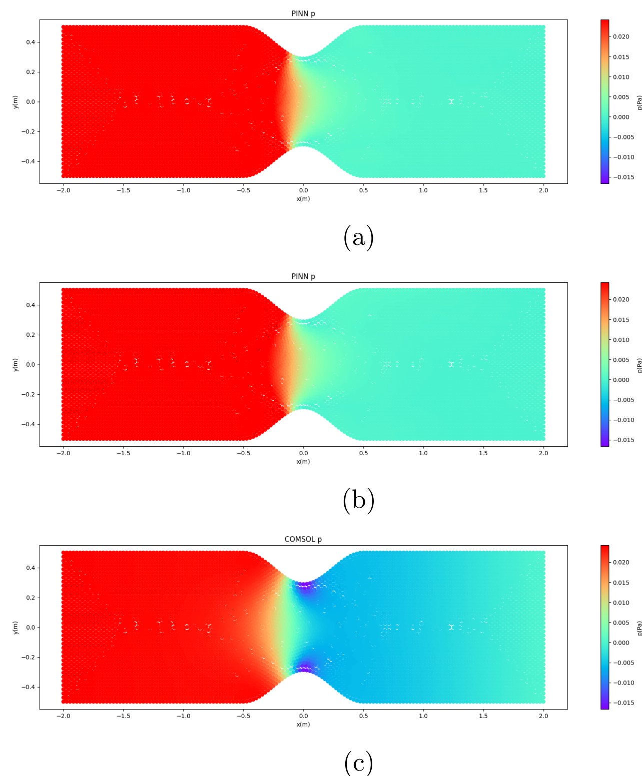


Figure 6.37: *Prediction of pressure in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

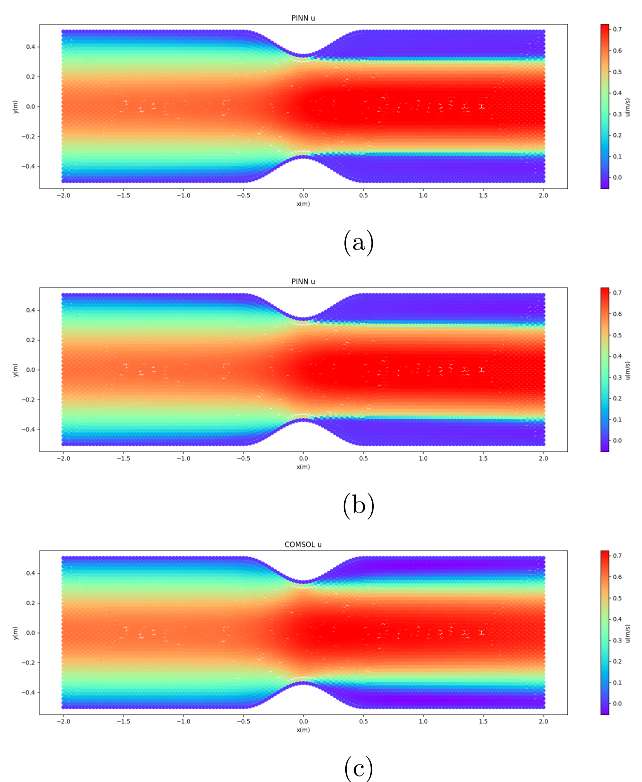


Figure 6.38: *Prediction of u velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

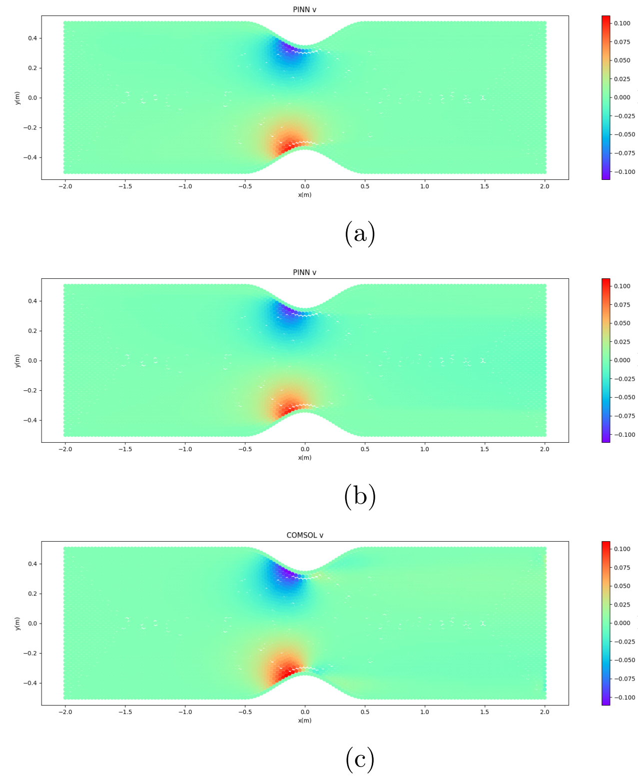


Figure 6.39: *Prediction of v velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

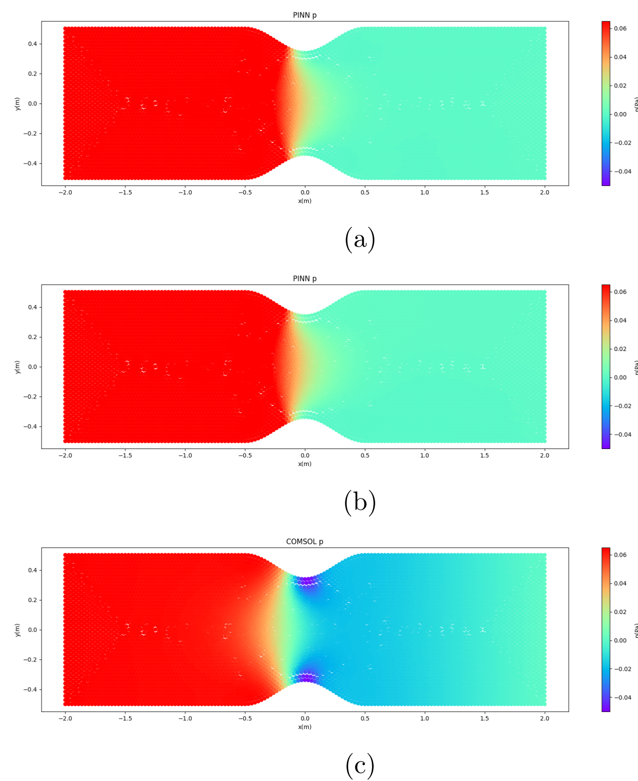
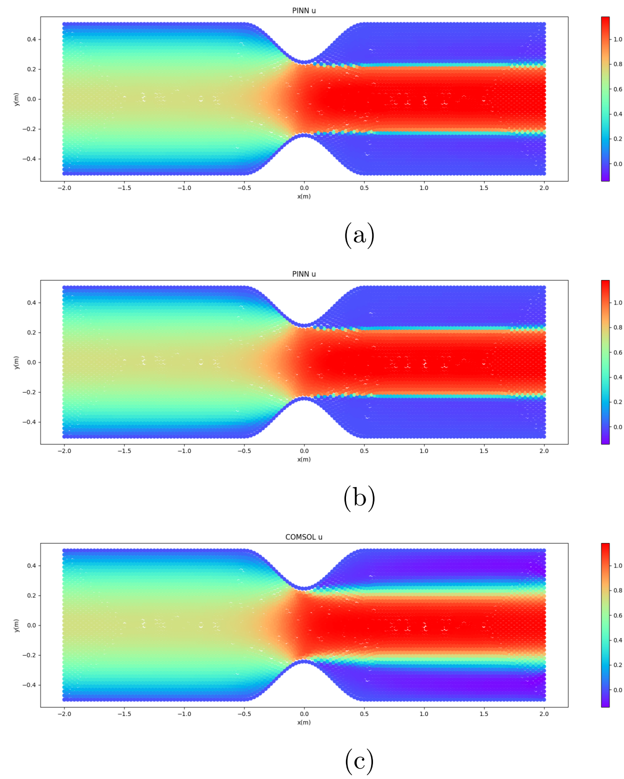


Figure 6.40: *Prediction of pressure in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

Accuracy of big network (6 layers-512 neurons per layer)



(a)

(b)

(c)

Figure 6.41: *Prediction of u velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

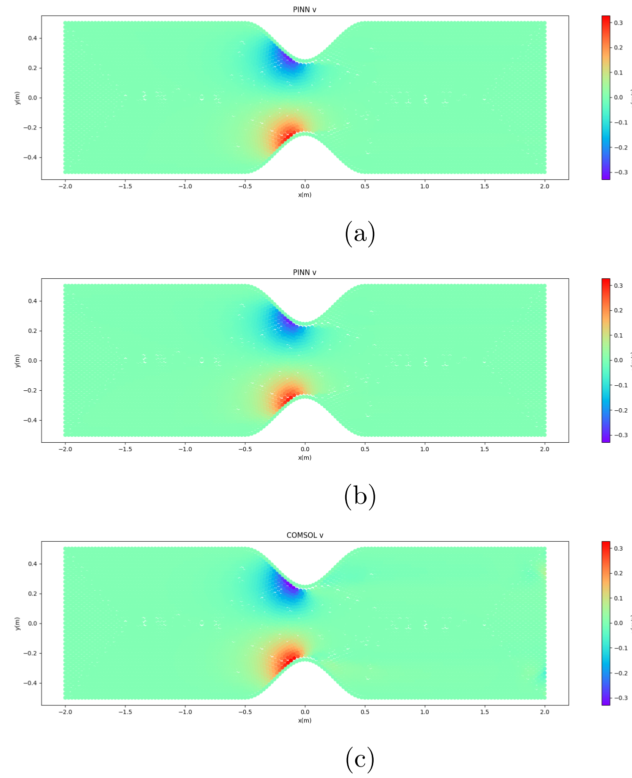


Figure 6.42: *Prediction of v velocity in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL..*

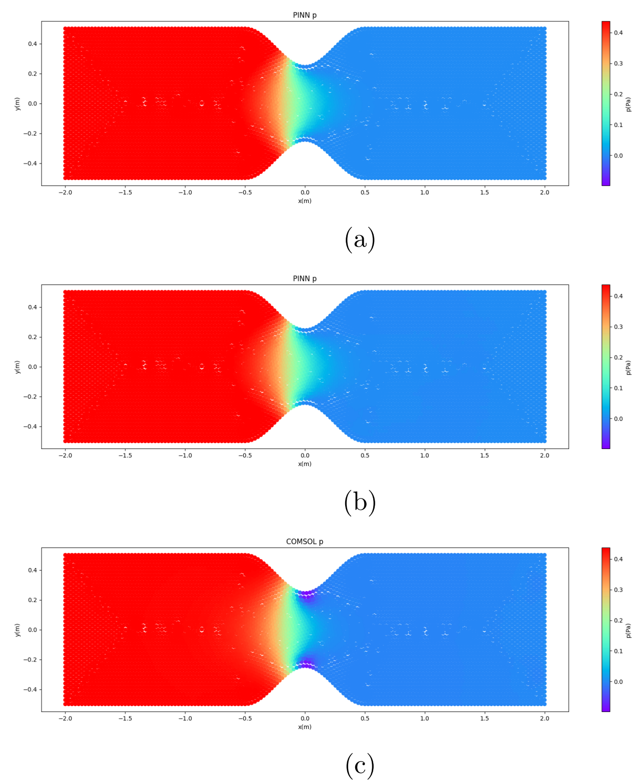
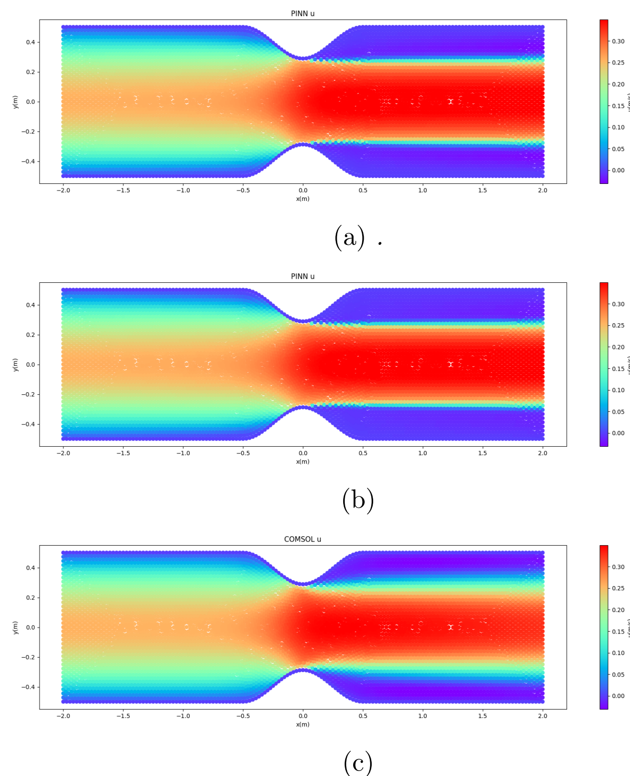


Figure 6.43: *Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

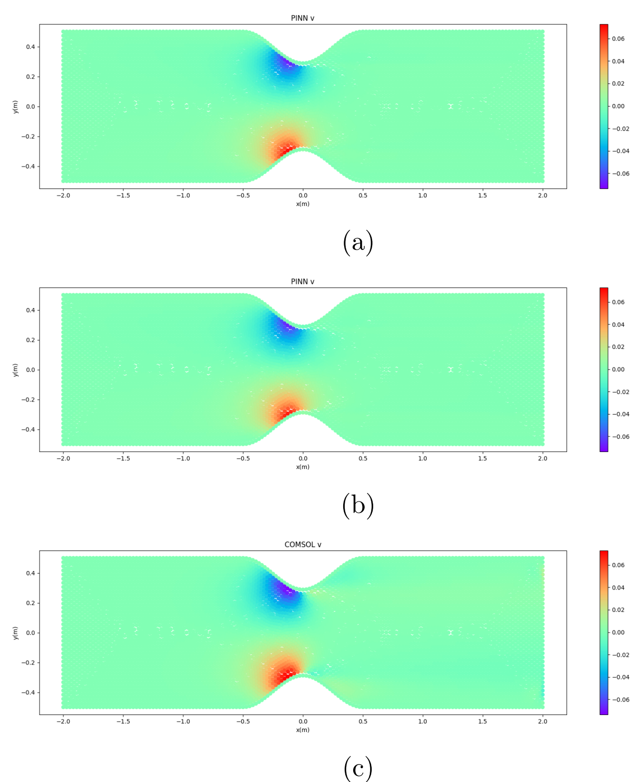


(a) .

(b)

(c)

Figure 6.44: *Prediction of u velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

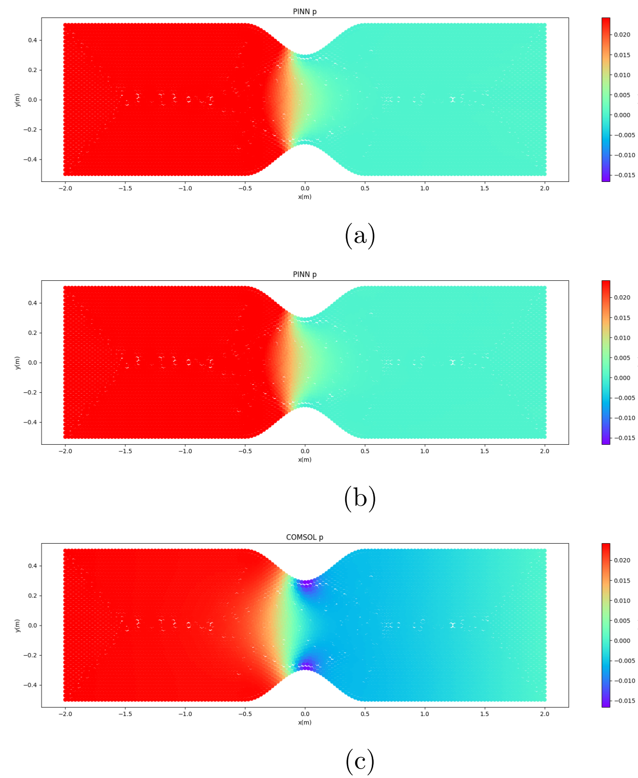


(a)

(b)

(c)

Figure 6.45: *Prediction of v velocity in Study 2 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

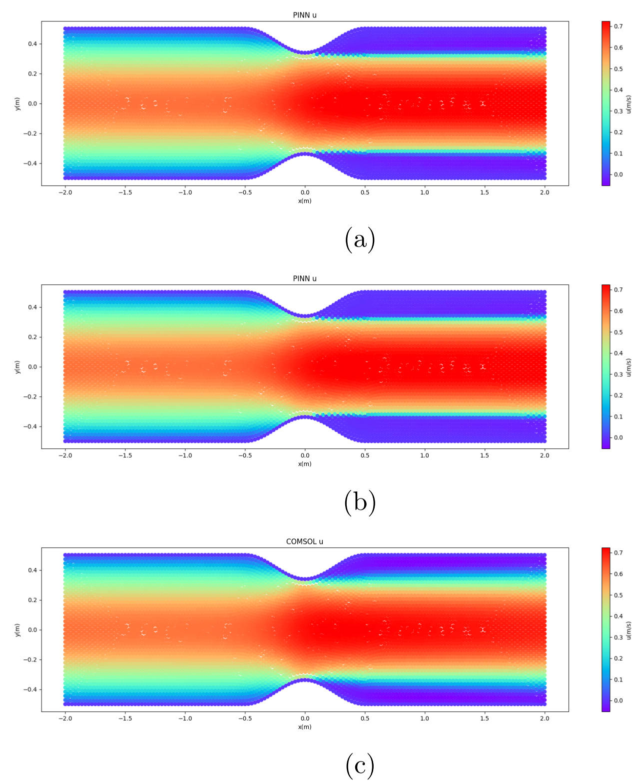


(a)

(b)

(c)

Figure 6.46: *Prediction of pressure in Study 1 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*



(a)

(b)

(c)

Figure 6.47: *Prediction of u velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

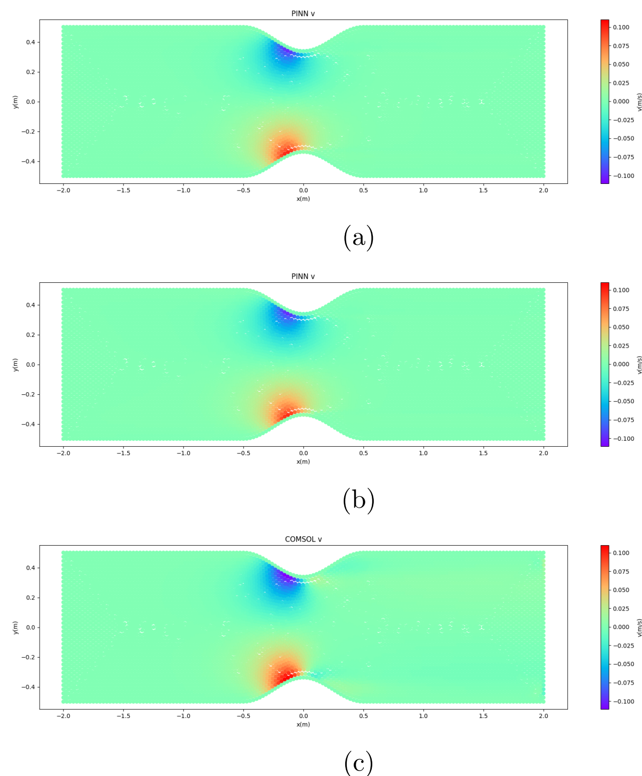


Figure 6.48: *Prediction of v velocity in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

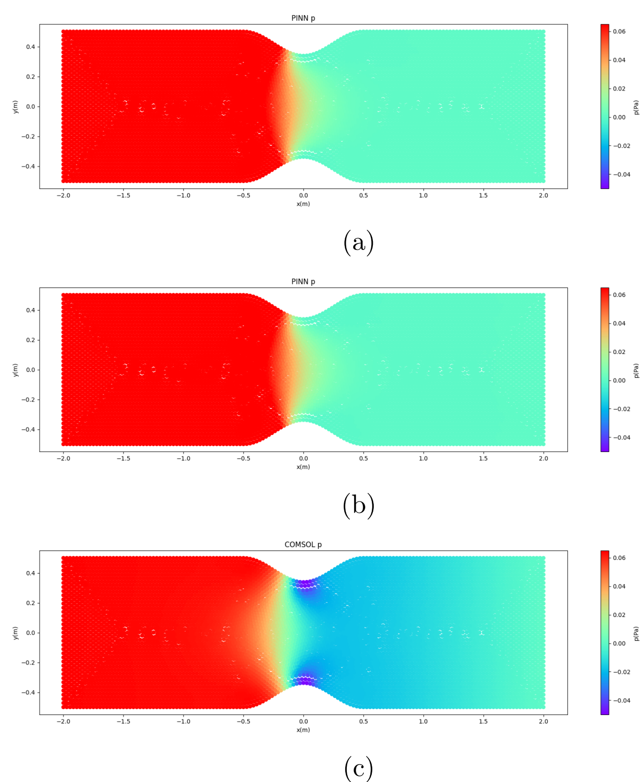


Figure 6.49: *Prediction of pressure in Study 3 (Table 6.3) using: (a) PINN without, (b) PINN with adaptive activation functions, (c) FEM in COMSOL.*

Table 6.4: *Errors for the different networks and studies.*

Study	Network	RMSE u	Norm RMSE u	RMSE v	Norm RMSE v	RMSE p	Norm RMSE p
1	Small	0.0893	0.0676	0.0153	0.0233	0.0666	0.1243
1	Adapt small	0.0799	0.0606	0.0138	0.0212	0.0479	0.0893
1	Big	0.0776	0.0588	0.0064	0.0098	0.0224	0.0418
1	Adapt big	0.0808	0.0612	0.0071	0.0108	0.0197	0.0368
2	Small	0.0374	0.0984	0.0038	0.0260	0.0106	0.2608
2	Adapt small	0.0314	0.0826	0.0038	0.0259	0.0082	0.2007
2	Big	0.0240	0.0630	0.0026	0.0180	0.0061	0.1505
2	Adapt big	0.0246	0.0647	0.0028	0.0192	0.0064	0.1560
3	Small	0.0475	0.0612	0.0048	0.0215	0.0257	0.2238
3	Adapt small	0.0461	0.0594	0.0058	0.0264	0.0248	0.2155
3	Big	0.0382	0.0492	0.0041	0.0189	0.0210	0.1826
3	Adapt big	0.0406	0.0523	0.0042	0.0191	0.0217	0.1885

As summarized in Table 6.4, the big network (6 hidden layers and 512 neurons per layer) with adaptive activation functions is associated with higher errors, while the small network is performing better (lower errors) in all 3 case studies (with parameters defined in Table 6.3) compared to to small network

6.6 Number of sampling points

In this section, the number of sampling points and its impact on the convergence rate and accuracy is studied. We examine if the small network (4 hidden layers with 256 neurons per hidden layer) with enabled adaptive activation functions of the section 6.5 can further reduce its total loss and improve its accuracy by sampling more points. To record the errors, we reuse again the 3 previous case studies (Table 6.6) .

A batch size of 1000 was consistently utilized in all trainings, resulting in a total of 3.84 million spatial points per epoch for sampling 1 and 5.92 million for sampling 2, since in each training iteration 3840 batch points and 5920 were used accordingly.

Table 6.5: *Number of points for the 2 samplings.*

Geometry	**Sampling 1**	**Sampling 2**
Inlet	160	160
Outlet	160	160
Up wall	160	300
Down wall	160	300
Interior	3200	5000

Table 6.6: *The parameters of the 3 case studies for the accuracy evaluation of different samplings.*

Study	fc	u_{max}(m/s)	Reynolds
1	0.256	1.111	1462
2	0.212	0.389	512
3	0.162	0.925	1217

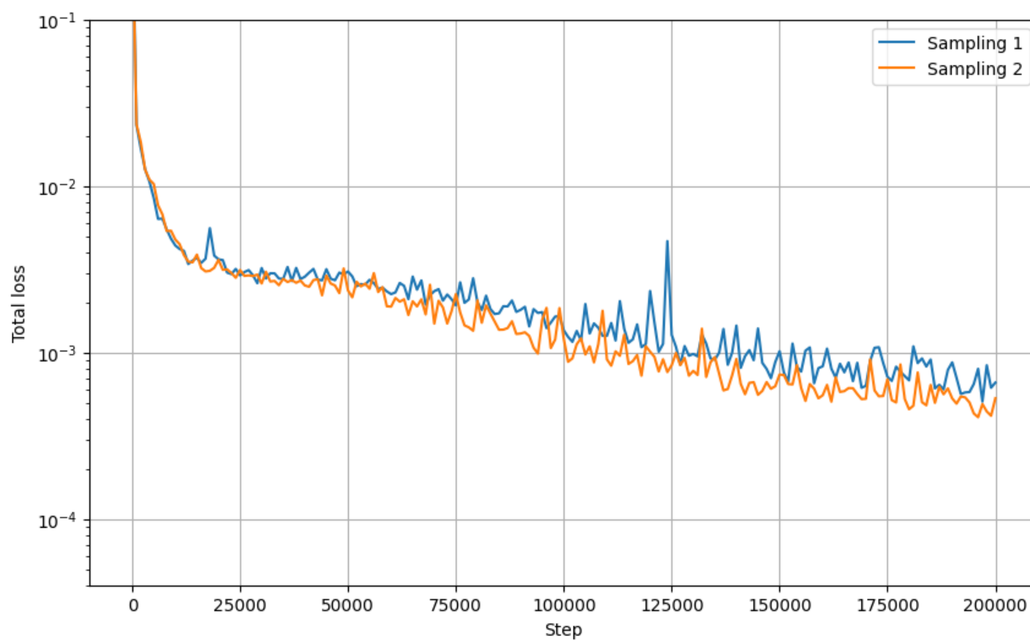


Figure 6.50: *Total loss for the 2 Samplings.*

As expected the total loss is lower on sampling 2 but with slightly highercomputation time per step.

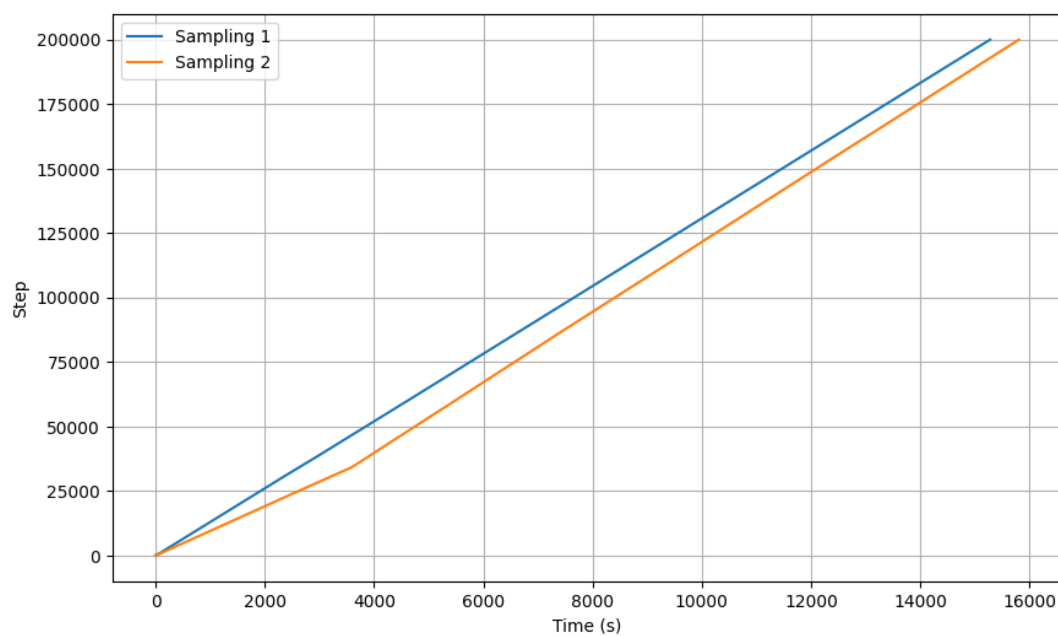


Figure 6.51: *Time per step for the 2 Samplings.*

Table 6.7: *Errors for the 2 samplings.*

Study	Network	RMSE u	Norm RMSE u	RMSE v	Norm RMSE v	RMSE p	Norm RMSE p
1	Sampling 1	0.0799	0.0606	0.0138	0.0212	0.0479	0.0893
1	Sampling 2	0.0798	0.0605	0.0107	0.0162	0.0190	0.0355
2	Sampling 1	0.0314	0.0826	0.0038	0.0259	0.0082	0.2007
2	Sampling 2	0.0314	0.0825	0.0034	0.0232	0.0080	0.1965
3	Sampling 1	0.0461	0.0594	0.0058	0.0264	0.0248	0.2155
3	Sampling 2	0.0467	0.0602	0.0055	0.0249	0.0255	0.2215

In Study 1 with the implementation of sampling 2, all errors decrease with a particularly significant reduction in the pressure error. In Study 2, all errors show a slight decline. Interestingly, in Study 3, only the error in v velocity decreases, while u velocity and pressure errors experience a modest increase.

6.7 Single-case PINN vs multi-case PINN

In this section difference in losses and accuracy between single-case PINN and multi-case PINN is investigated. Both networks consist of 6 hidden layers and 512 neurons per hidden layer, the activation function selected is SiLU and Adam optimizer are used. The single-case PINN is trained for $fc = 0.2$ and Reynolds number approximately 1315, and its errors are compared with the multi-case architecture for the same case.

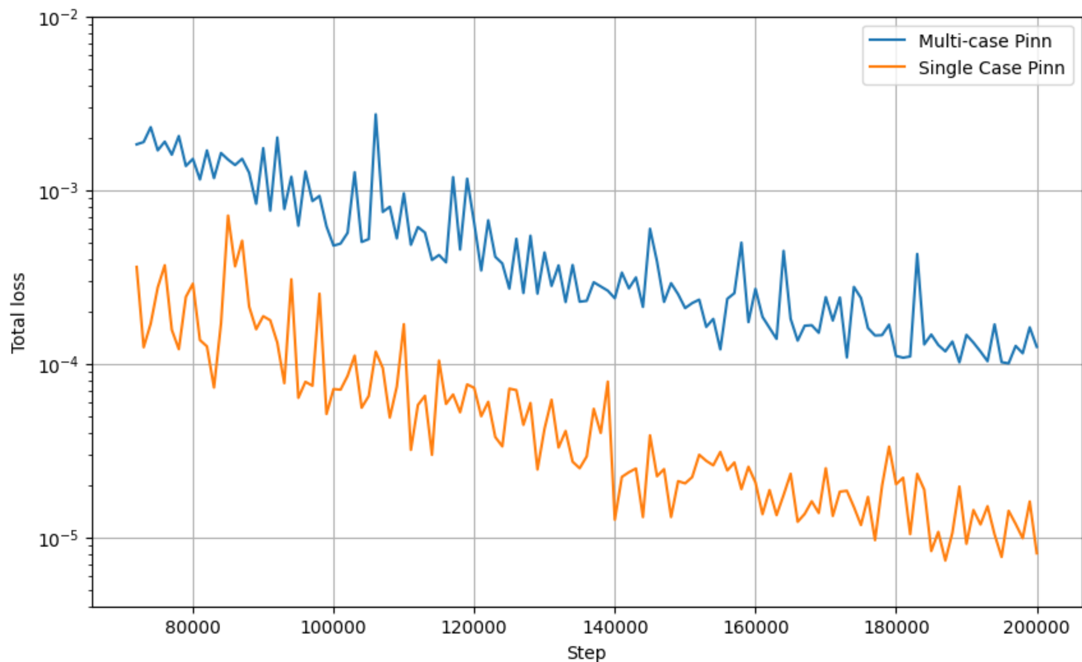


Figure 6.52: *Total loss for the 2 networks.*

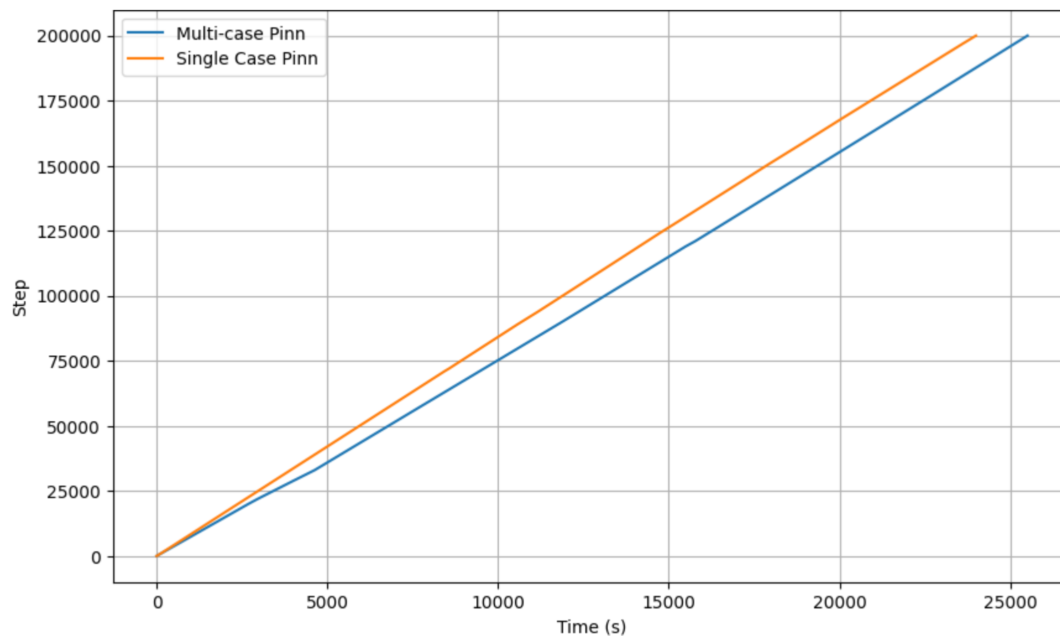
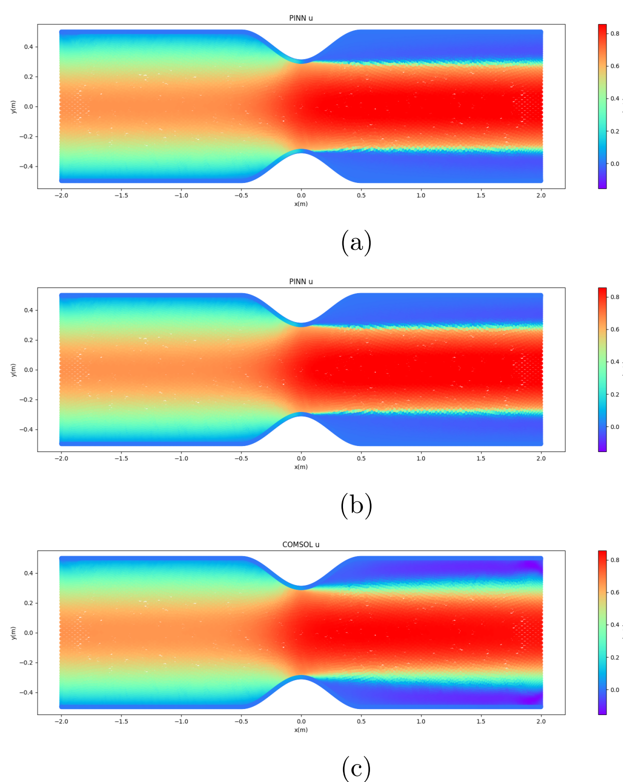


Figure 6.53: *Time per step for the 2 networks.*

As we see, total loss for the single-case network is much lower with smaller time per step. That was totally expected, since the multi-case Pinn has two additional dimensions fc and u_{max}, making it is much more difficult for the network to learn the hidden fluid mechanics.

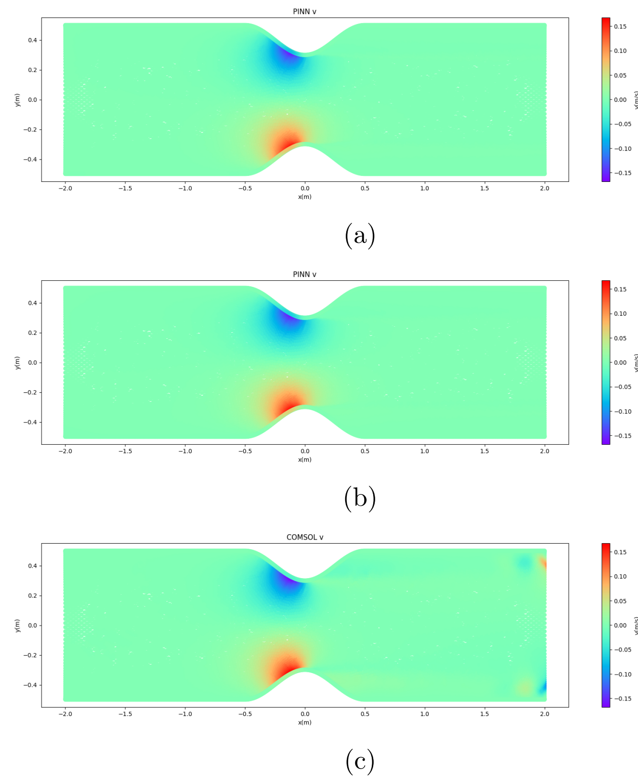


(a)

(b)

(c)

Figure 6.54: *Prediction of u velocity using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL.*

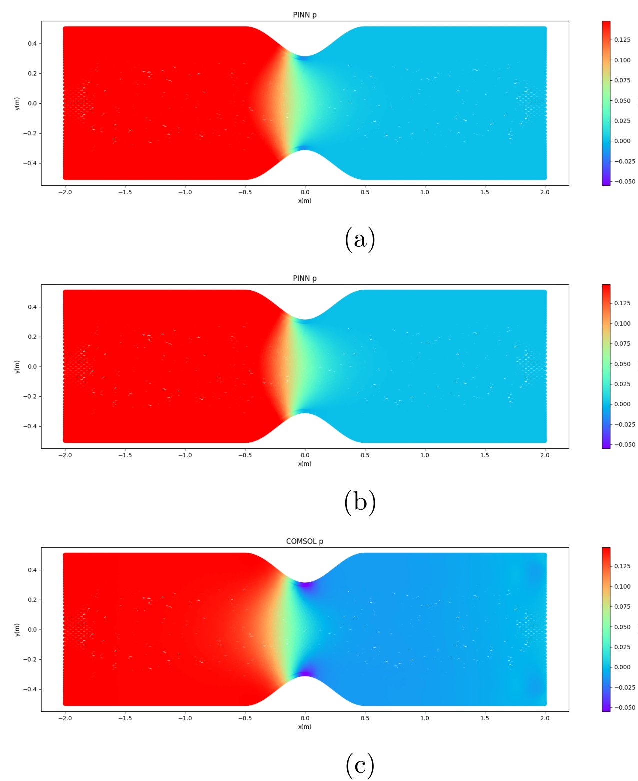


(a)

(b)

(c)

Figure 6.55: *Prediction of v velocity using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL.*



(a)

(b)

(c)

Figure 6.56: *Prediction of pressure using: (a) Single-case PINN, (b) Multi-case PINN, (c) FEM in COMSOL.*

Table 6.8: *Errors for different networks.*

Network	RMSE u	Norm RMSE u	RMSE v	Norm RMSE v	RMSE p	Norm RMSE p
Single-case	0.0468	0.0462	0.0081	0.0240	0.0167	0.0824
Multi-case	0.0483	0.0477	0.0079	0.0236	0.0238	0.1174

Upon observation, it is seen that errors in the u velocity do not reduce significantly in the single-case architecture. However, errors in the v velocity partially increase, while errors in pressure noticeably decrease.

Chapter 7

FEM vs. multi-case PINN

7.1 Introduction

In this chapter, we examine the accuracy of the multi-case PINN vs. FEM ground truth data and investigate how severity of the stenosis and Reynolds number influence the different errors. The PINN used, had 6 hidden layers, 512 neurons per hidden layer, SiLU as the activation function and Adam algorithm as the optimizer.

7.2 Case study 1: Re=500, fc =0.1

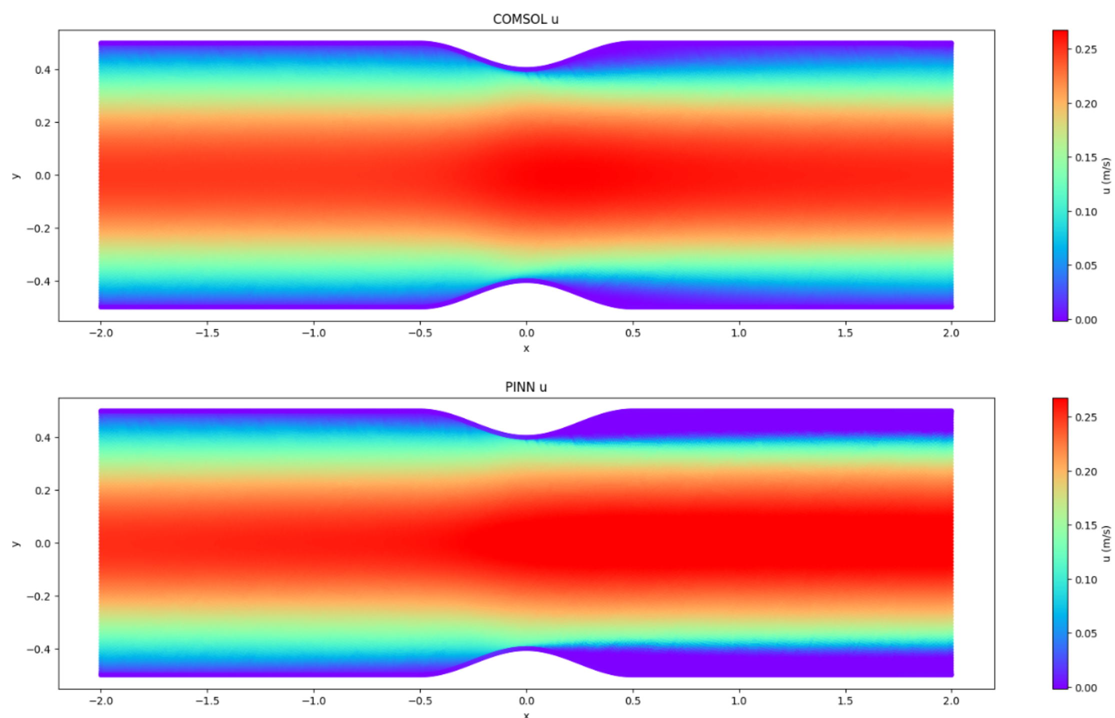


Figure 7.1: *u velocity in multi-case PINN vs FEM for Re=500 and fc=0.1.*

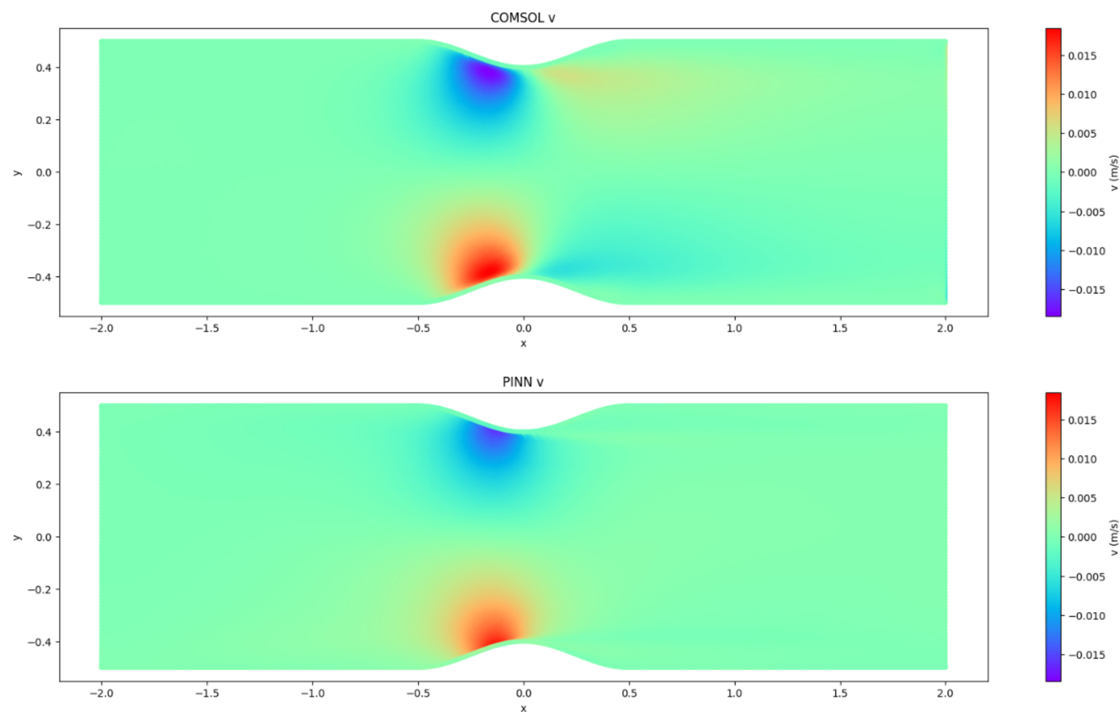


Figure 7.2: *v velocity in multi-case PINN vs FEM for Re=500 and fc=0.3.*

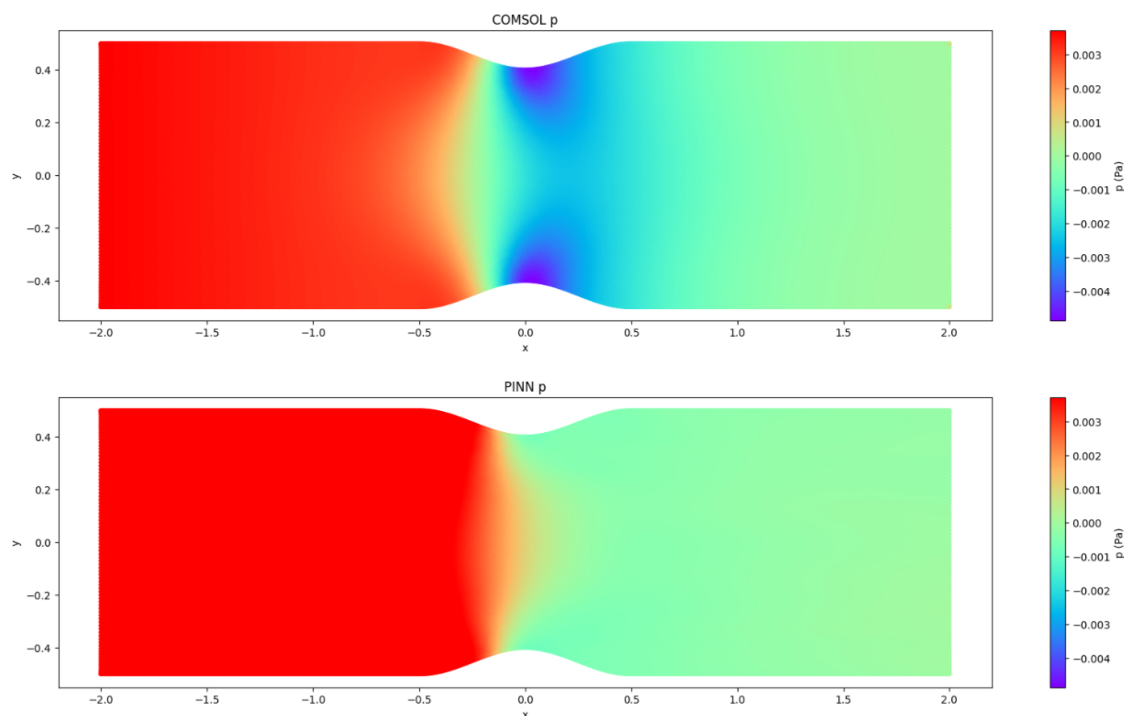


Figure 7.3: *pressure in multi-case PINN vs FEM for Re=500 and fc=0.3.*

As illustrated in Fig.7.1, the PINN manages to accurately represent the u velocity field. However, in Fig.7.2, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.3, the PINN can not display the pressure drop, that is expected at the stenosis, as a result of the energy

loss.

7.3 Case study 2: Re=500, fc =0.2

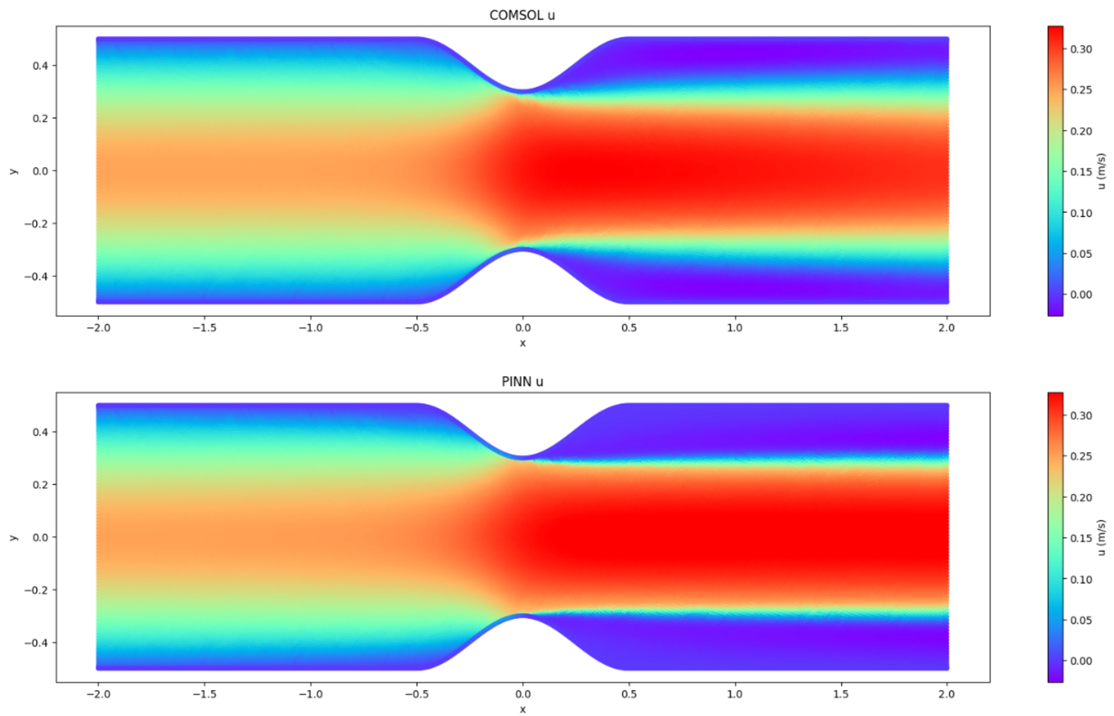


Figure 7.4: *u velocity in multi-case PINN vs FEM for Re=500 and fc=0.2.*

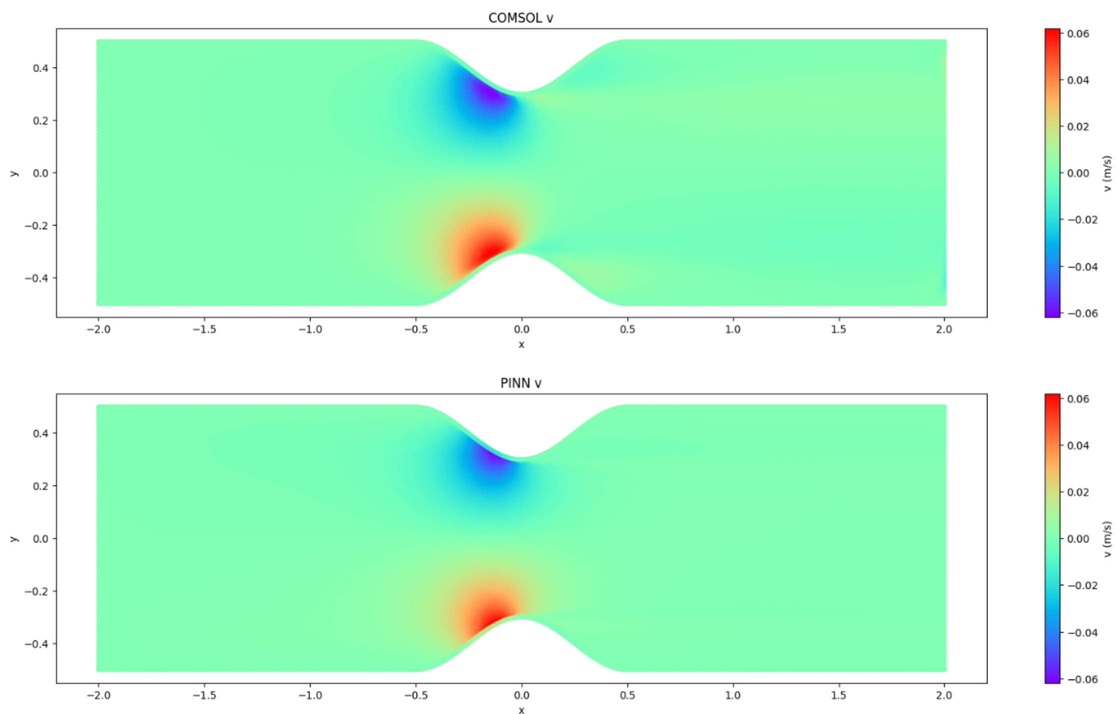


Figure 7.5: *v velocity in multi-case PINN vs FEM for Re=500 and fc=0.2.*

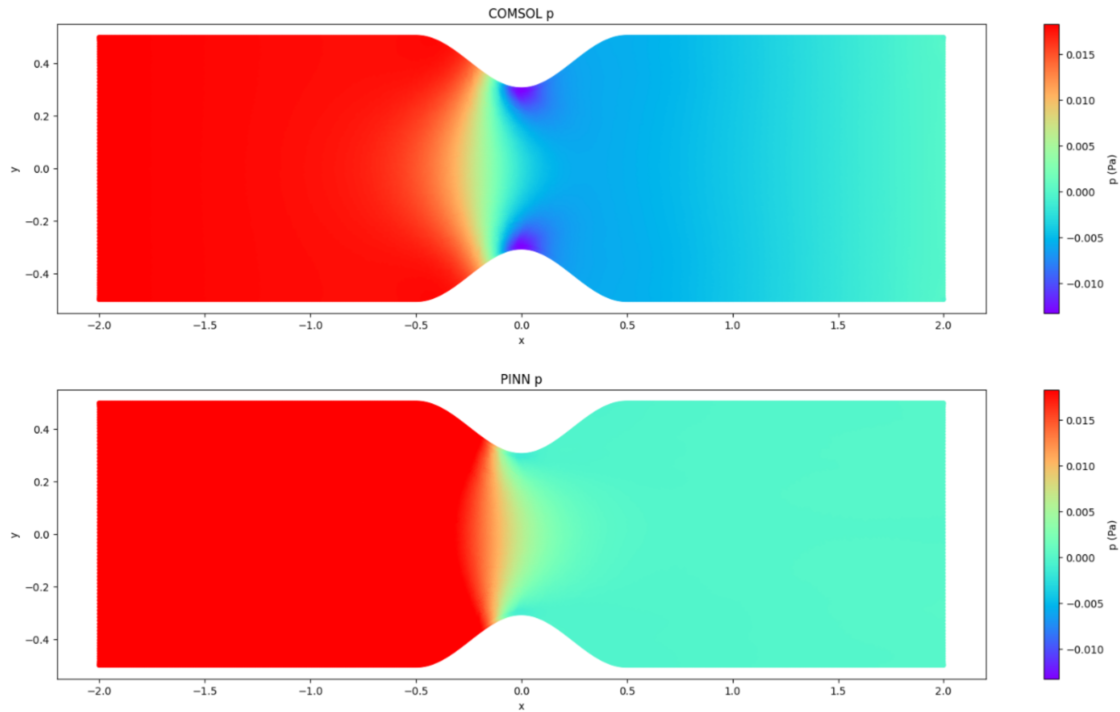


Figure 7.6: *pressure in multi-case PINN vs FEM for Re=500 and fc=0.2.*

As observed in Fig.7.4, deep learning finds it difficult to represent the u velocity gradient after the stenosis. In Fig.7.5, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.6, the PINN can not display the gradual pressure drop.

7.4 Case study 3: Re=500, fc =0.3

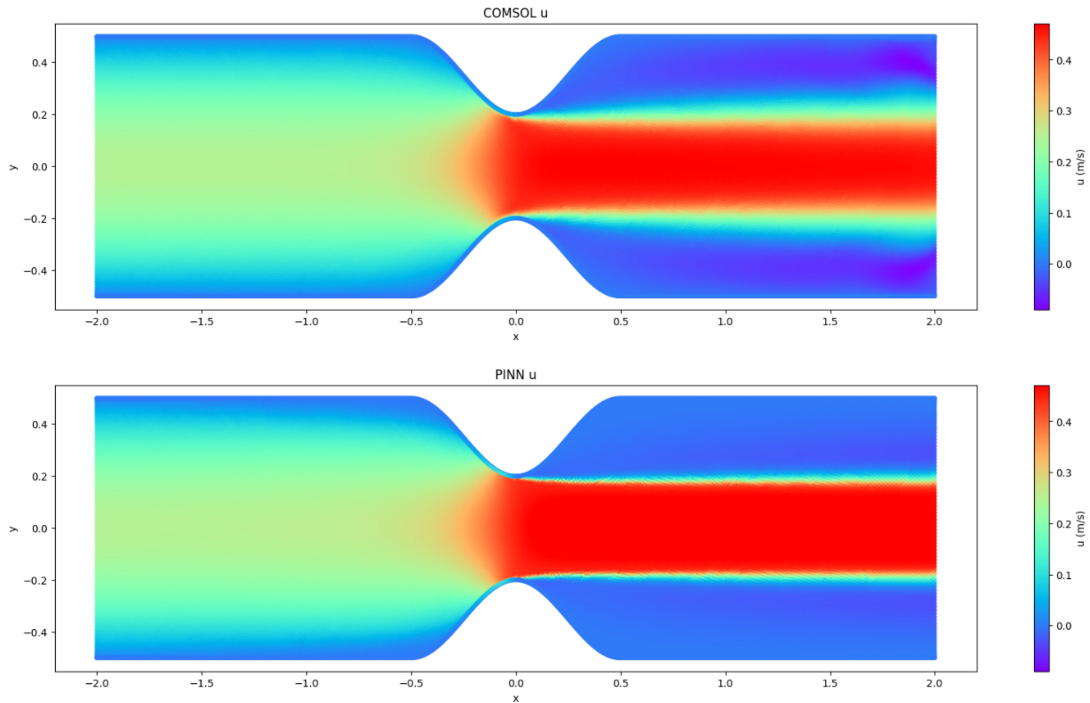


Figure 7.7: *u velocity in multi-case PINN vs FEM for Re=500 and fc=0.3.*

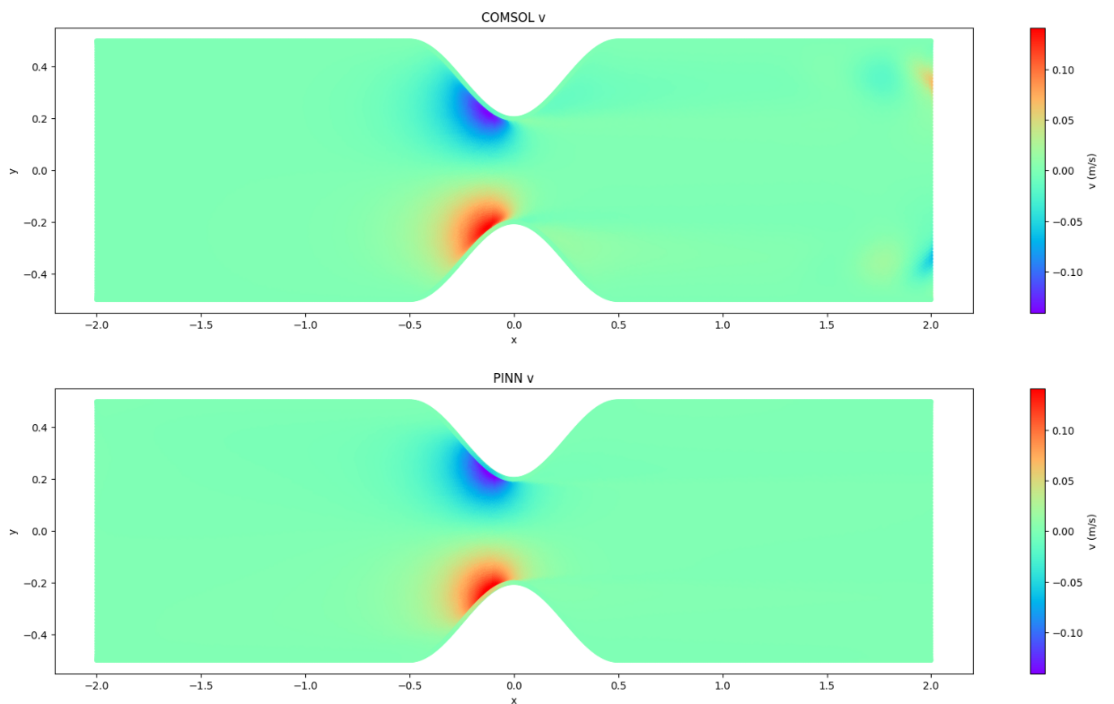


Figure 7.8: *v velocity in multi-case PINN vs FEM for Re=500 and fc=0.3.*

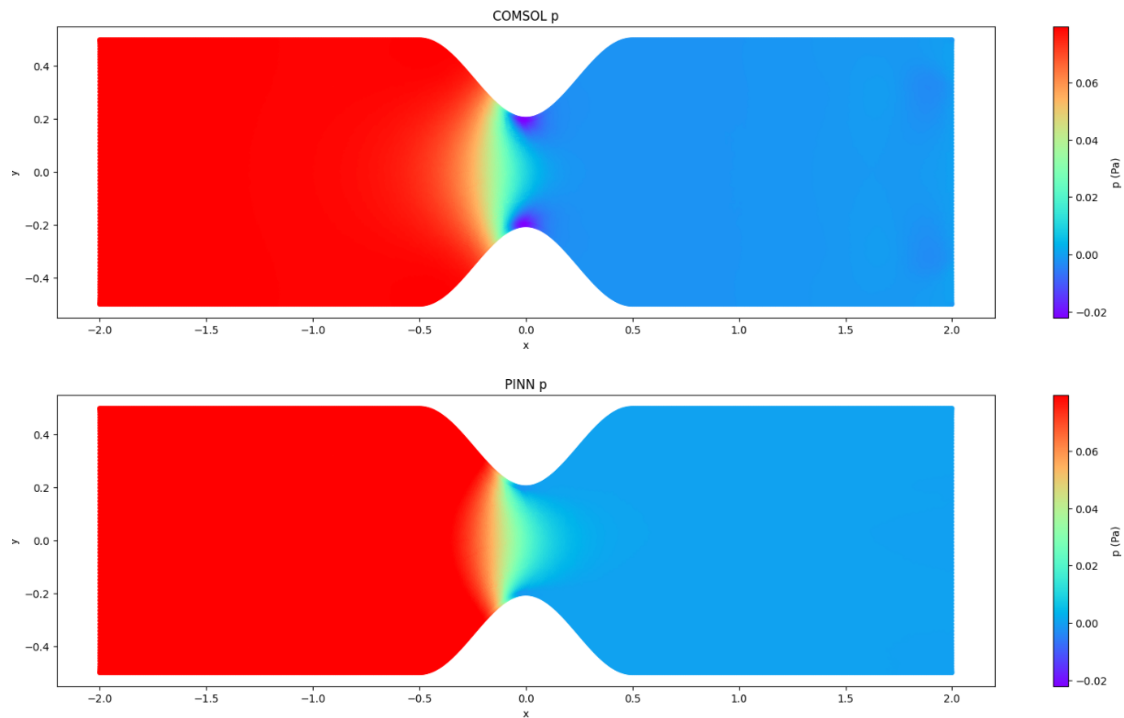


Figure 7.9: *pressure in multi-case PINN vs FEM for Re=500 and fc=0.3.*

As observed in Fig.7.8 and Fig.7.9, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.5 Case study 4: Re=750, fc =0.1

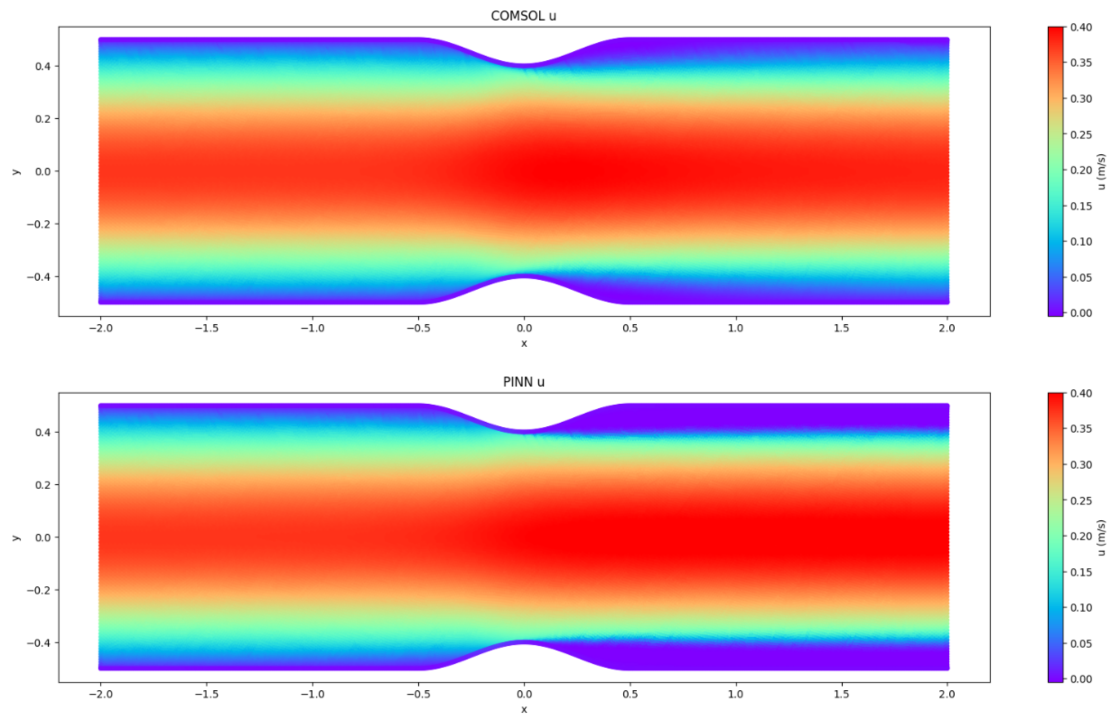


Figure 7.10: *u velocity in multi-case PINN vs FEM for Re=750 and fc=0.1.*

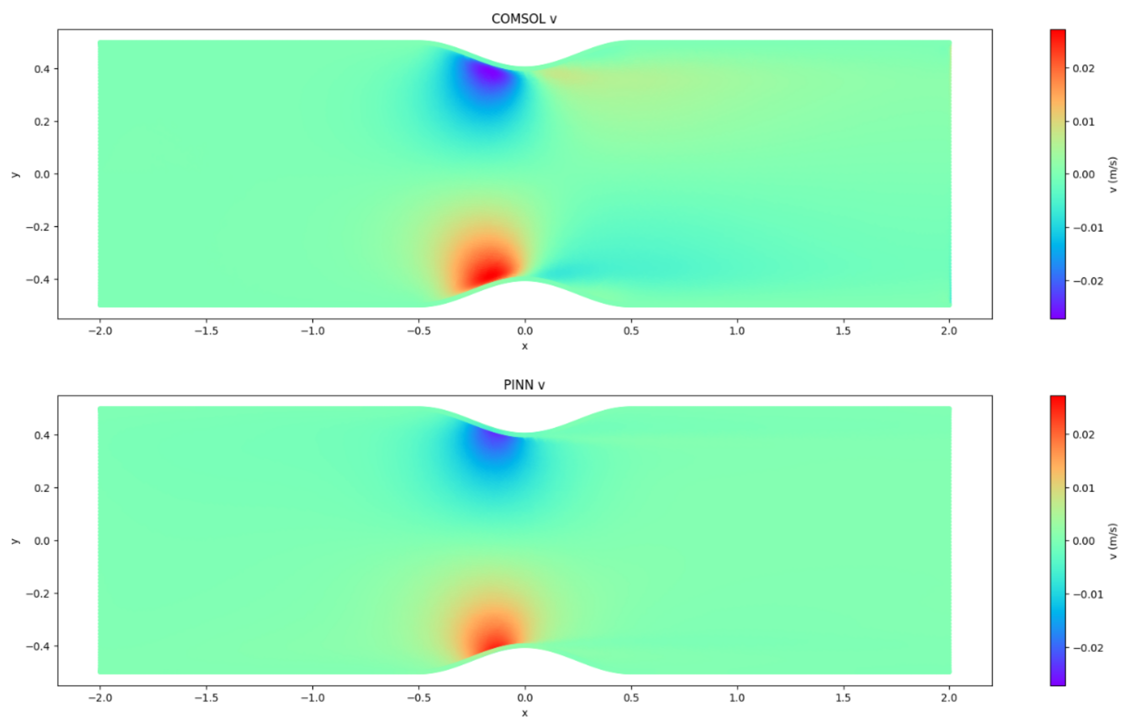


Figure 7.11: *v velocity in multi-case PINN vs FEM for Re=750 and fc=0.1.*

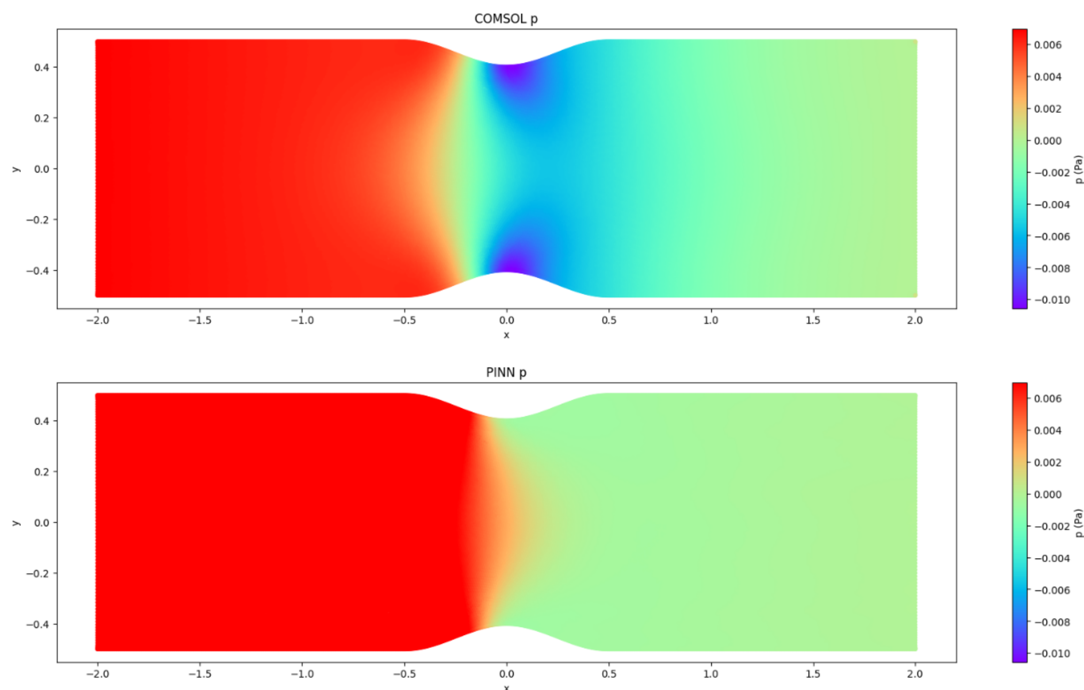


Figure 7.12: *pressure in multi-case PINN vs FEM for Re=750 and fc=0.1.*

As displayed in Fig.7.10, the PINN manages to accurately represent the u velocity field. However, in Fig. 7.11, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.12, the PINN can not depict the pressure drop, that is expected at the stenosis.

7.6 Case study 5: Re=750, fc =0.2

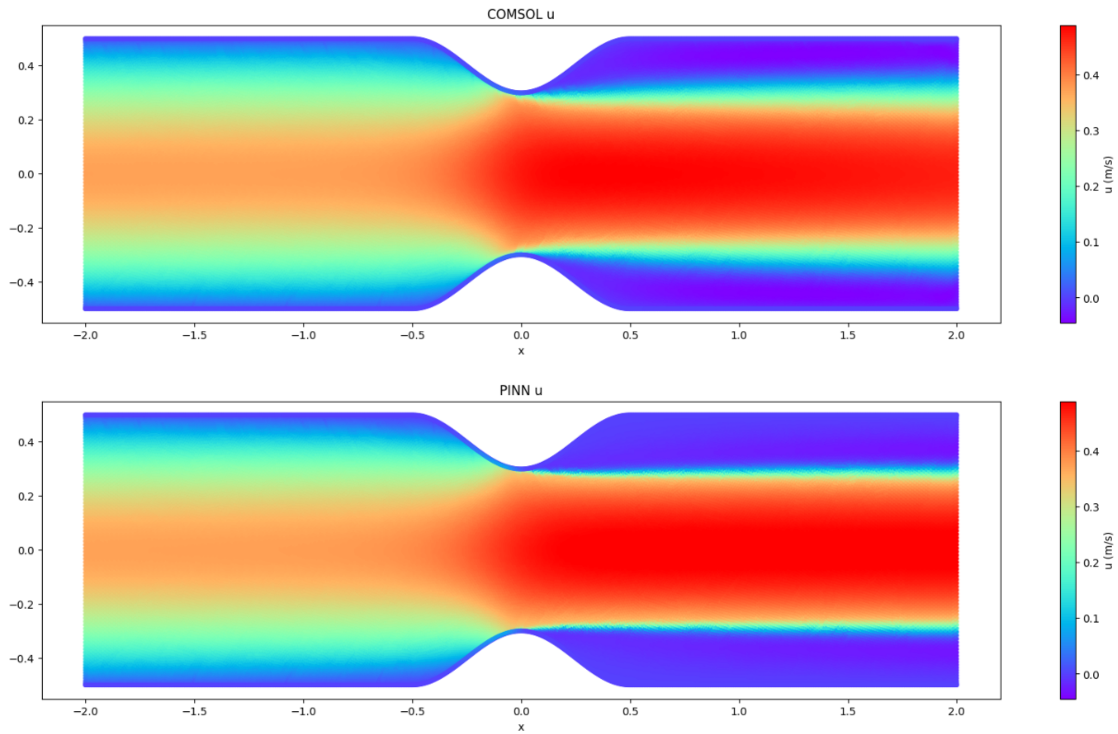


Figure 7.13: *u velocity in multi-case PINN vs FEM for Re=750 and fc=0.2.*

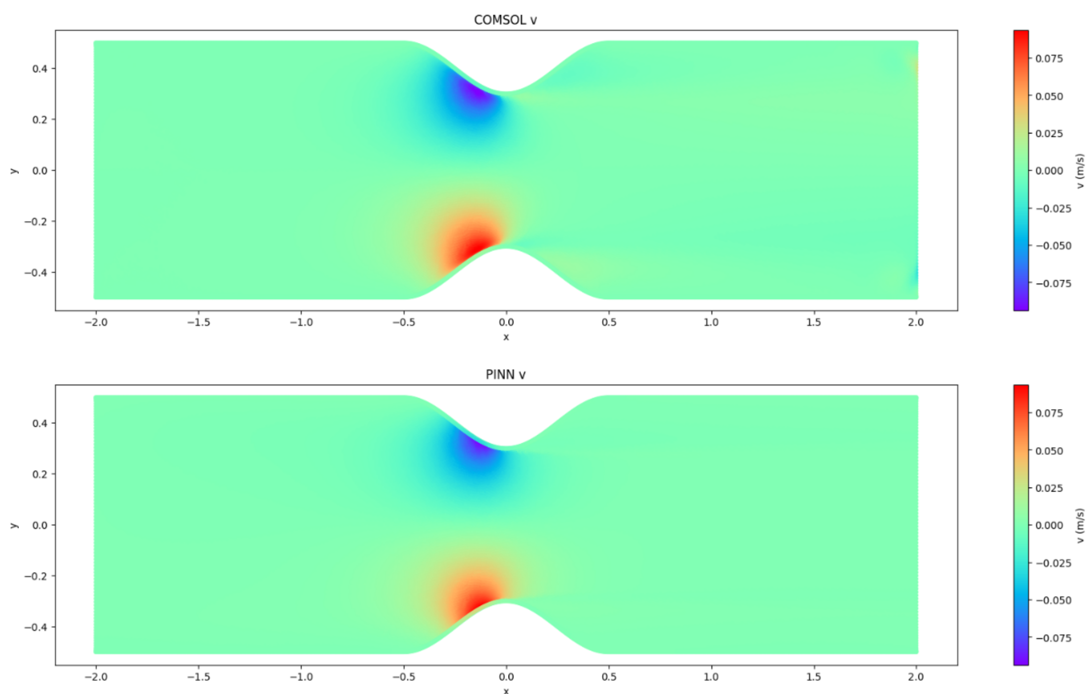


Figure 7.14: *v velocity in multi-case PINN vs FEM for Re=750 and fc=0.2.*

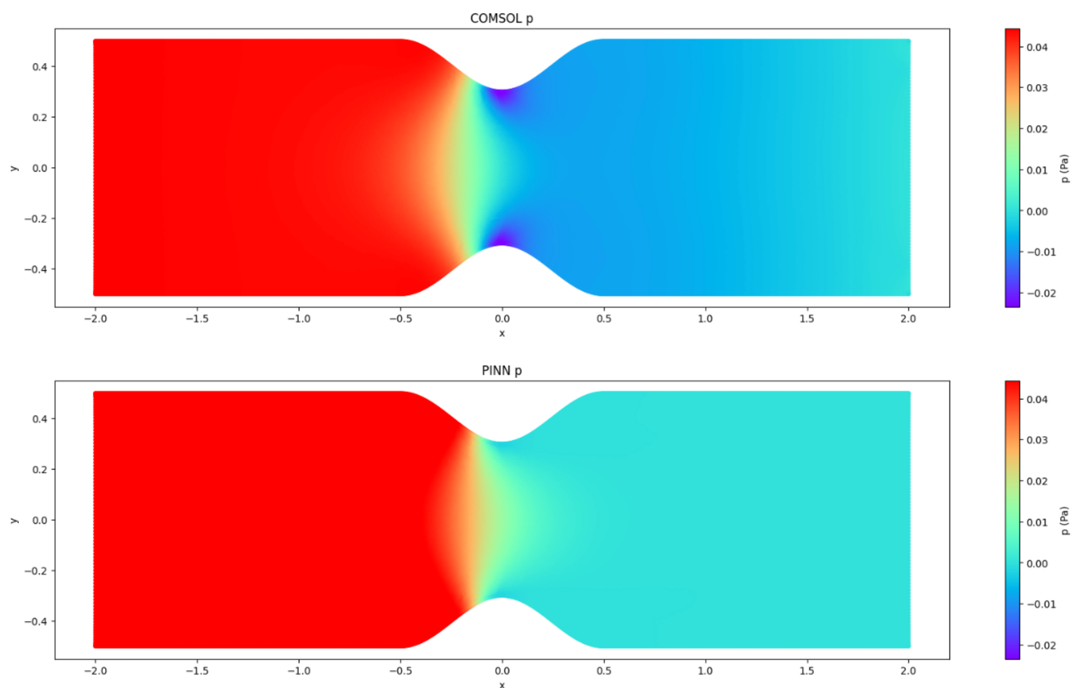


Figure 7.15: *pressure in multi-case PINN vs FEM for Re=750 and fc=0.2.*

As observed in Fig.7.13, deep learning can not accurately represent the u velocity gradient after the stenosis. In Fig.7.14, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.15, the PINN can not display the gradual pressure drop.

7.7 Case study 6: Re=750, fc =0.3

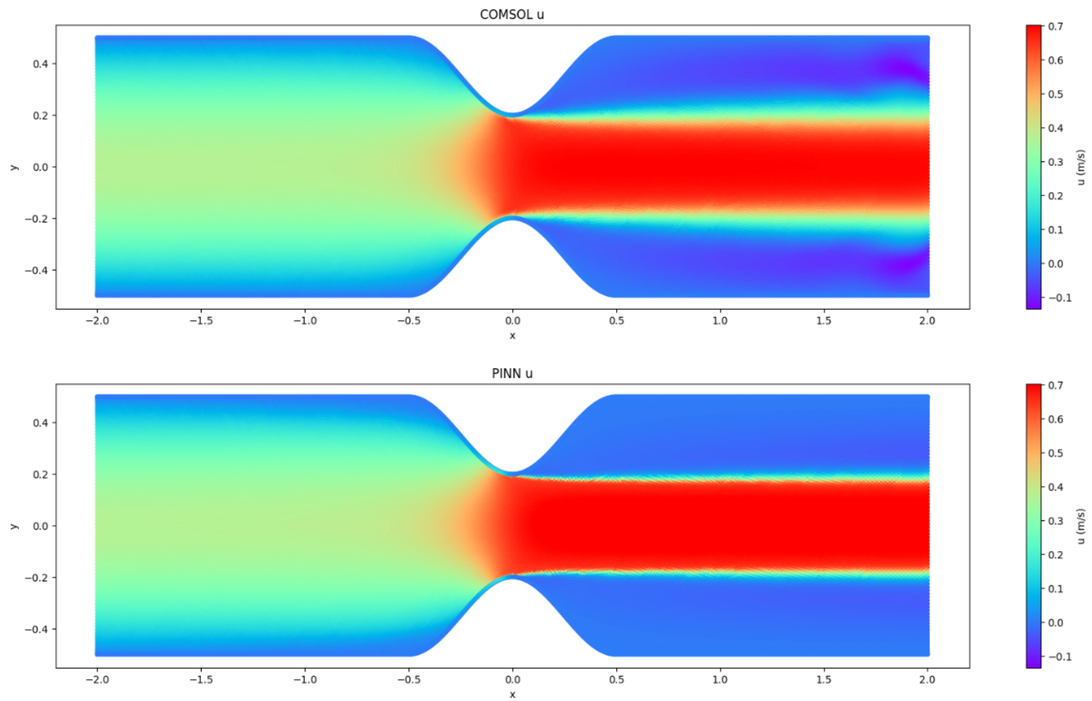


Figure 7.16: *u velocity in multi-case PINN vs FEM for Re=750 and fc=0.3.*

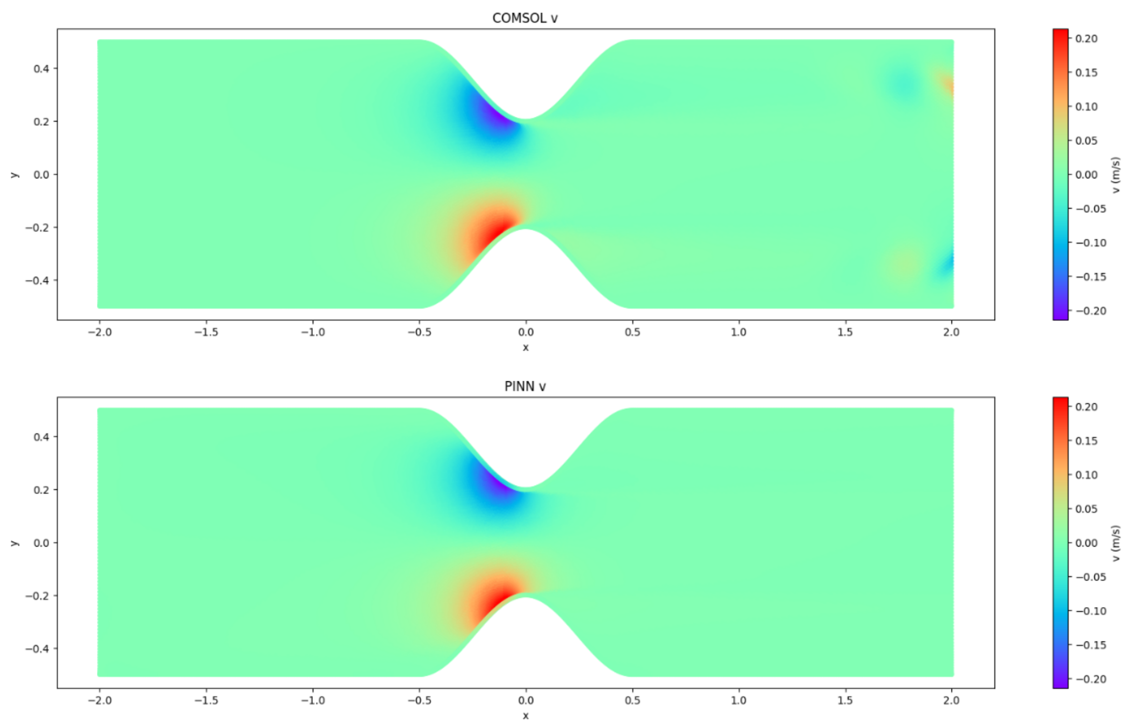


Figure 7.17: *v velocity in multi-case PINN vs FEM for Re=750 and fc=0.3.*

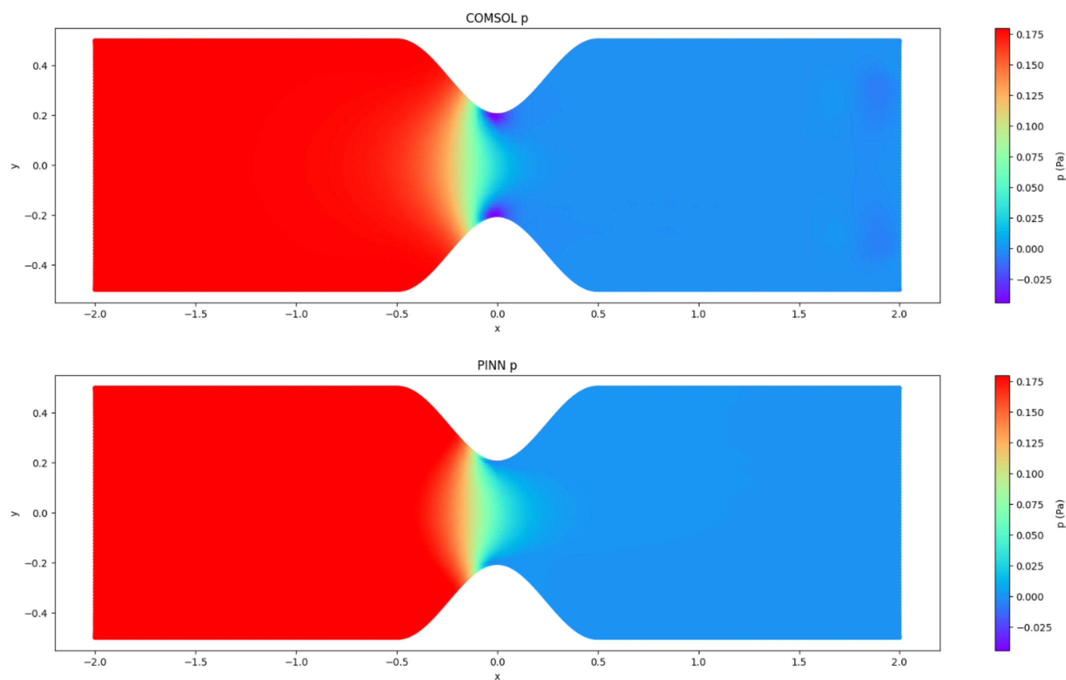


Figure 7.18: *pressure in multi-case PINN vs FEM for Re=750 and fc=0.3.*

As illustrated in Fig.7.17 and Fig.7.18, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.8 Case study 7: Re=1000, fc =0.1

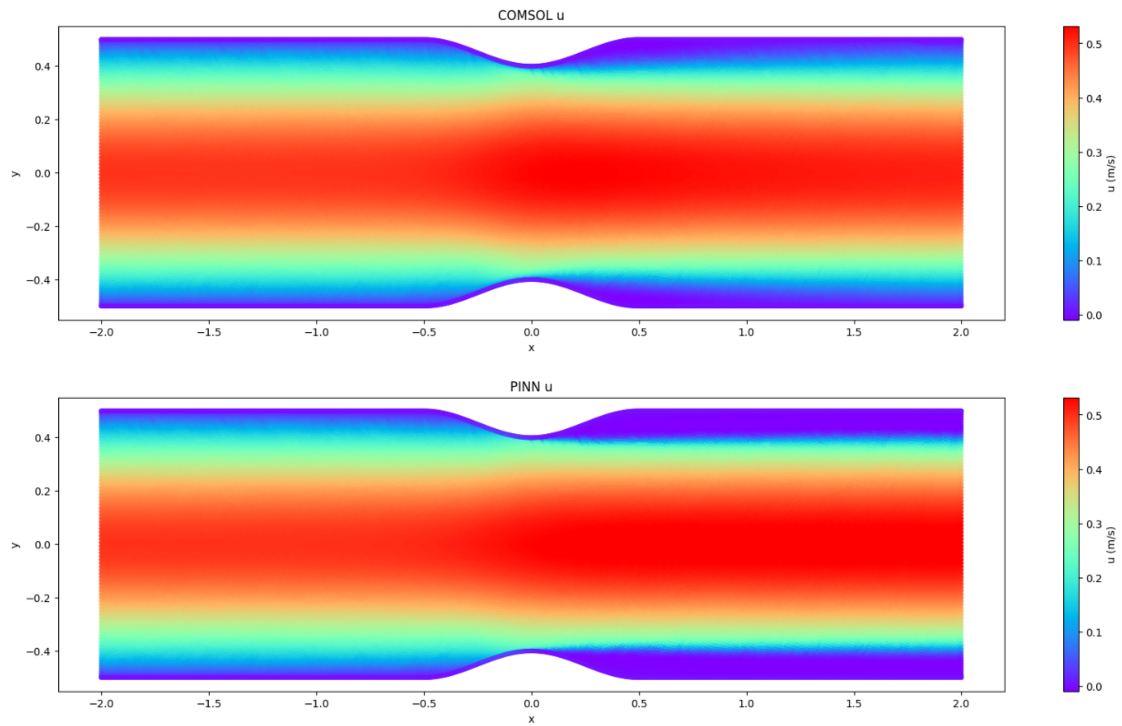


Figure 7.19: *u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.1.*

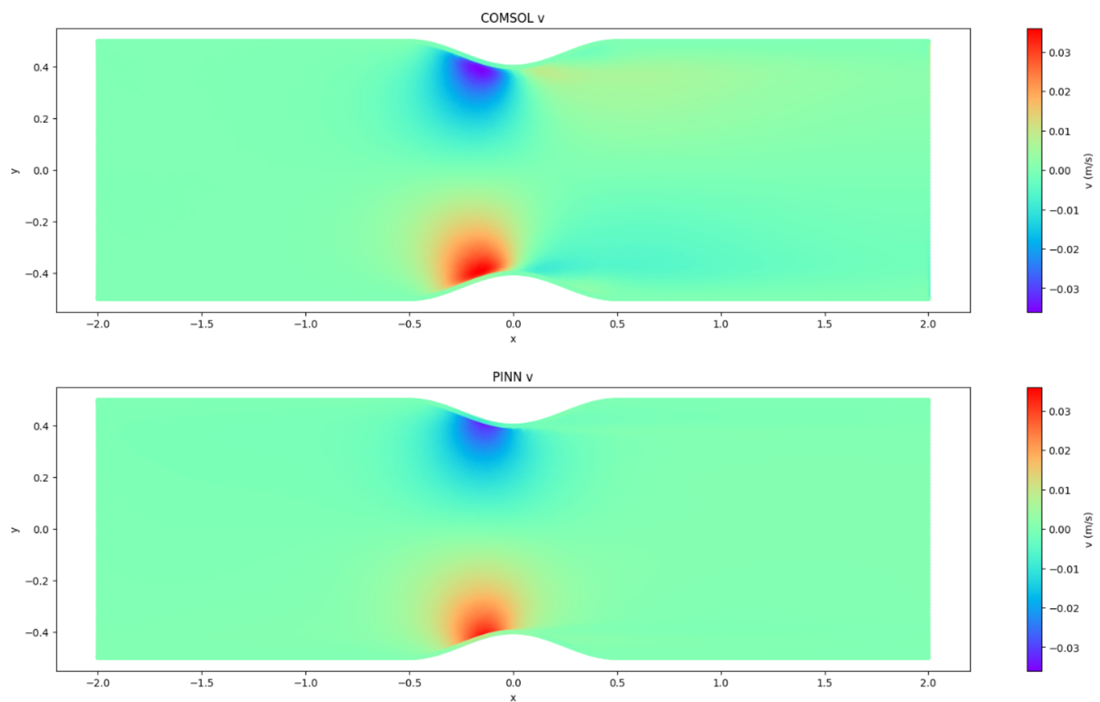


Figure 7.20: *v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.1.*

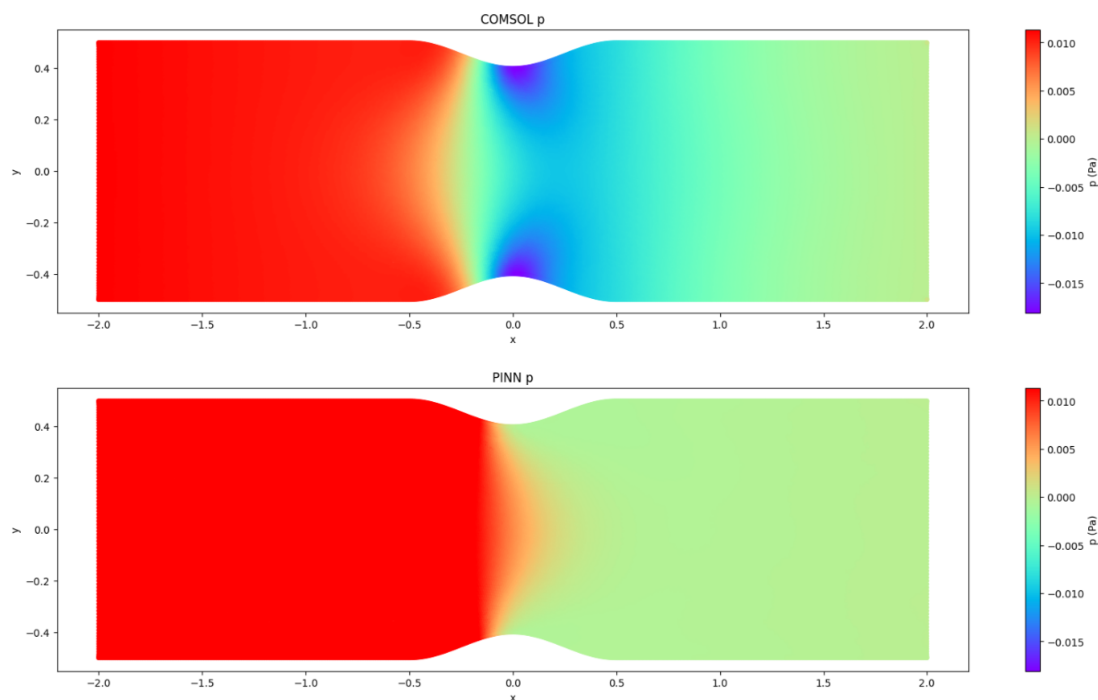


Figure 7.21: *pressure in multi-case PINN vs FEM for Re=1000 and fc=0.1.*

As observed in Fig.7.19, the PINN manages to accurately represent the u velocity field. However, in Fig.7.20, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.21, the PINN can not display the pressure drop, that is expected at the stenosis, as a result of the energy loss.

7.9 Case study 8: Re=1000, fc =0.2

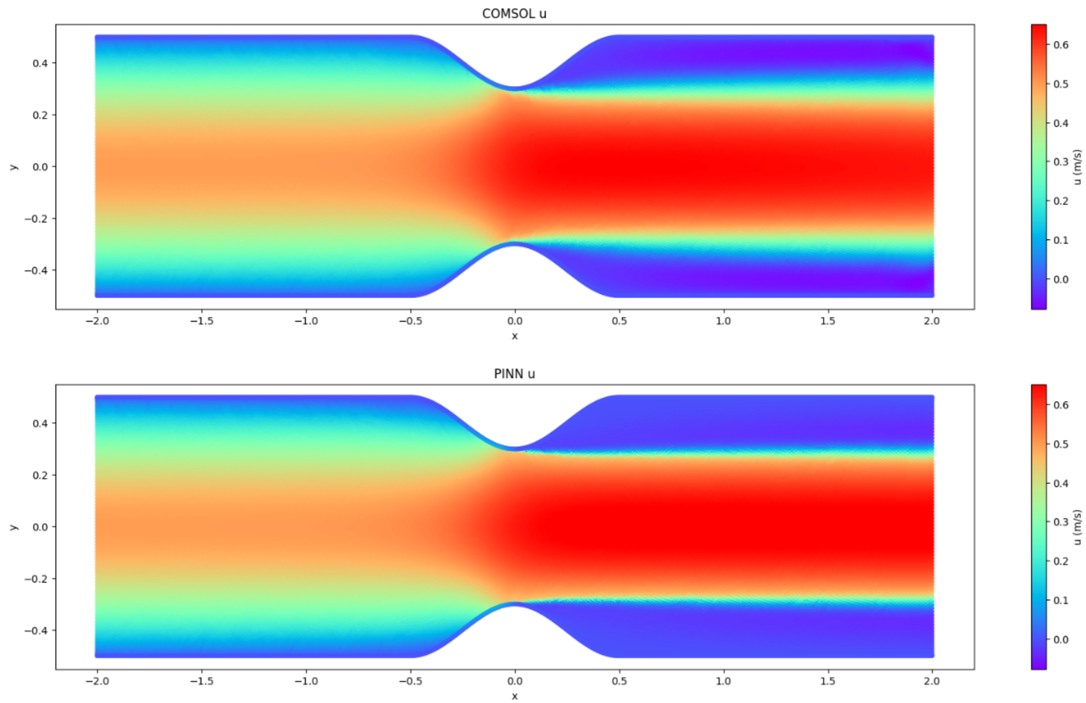


Figure 7.22: *u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.2.*

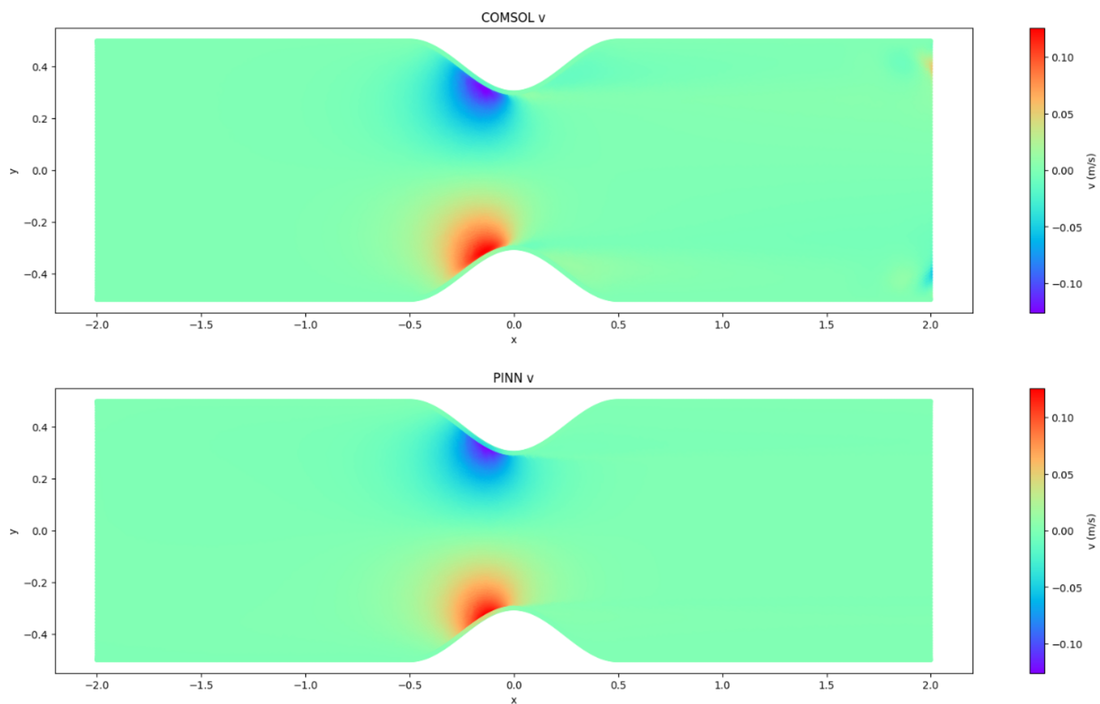


Figure 7.23: *v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.2.*

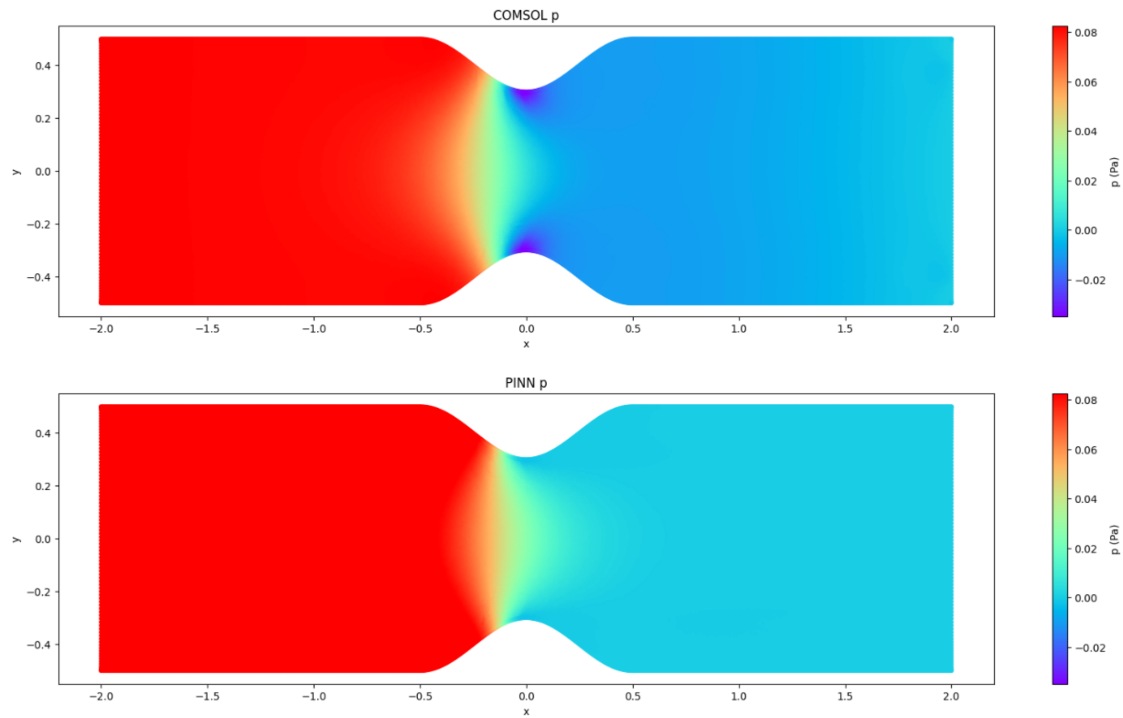


Figure 7.24: *pressure in multi-case PINN vs FEM for Re=1000 and fc=0.2.*

As illustrated in Fig.7.22, deep learning finds it difficult to represent the u velocity gradient after the stenosis. In Fig.7.23, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.24, the PINN can not display the gradual pressure drop.

7.10 Case study 9: Re=1000, fc =0.3

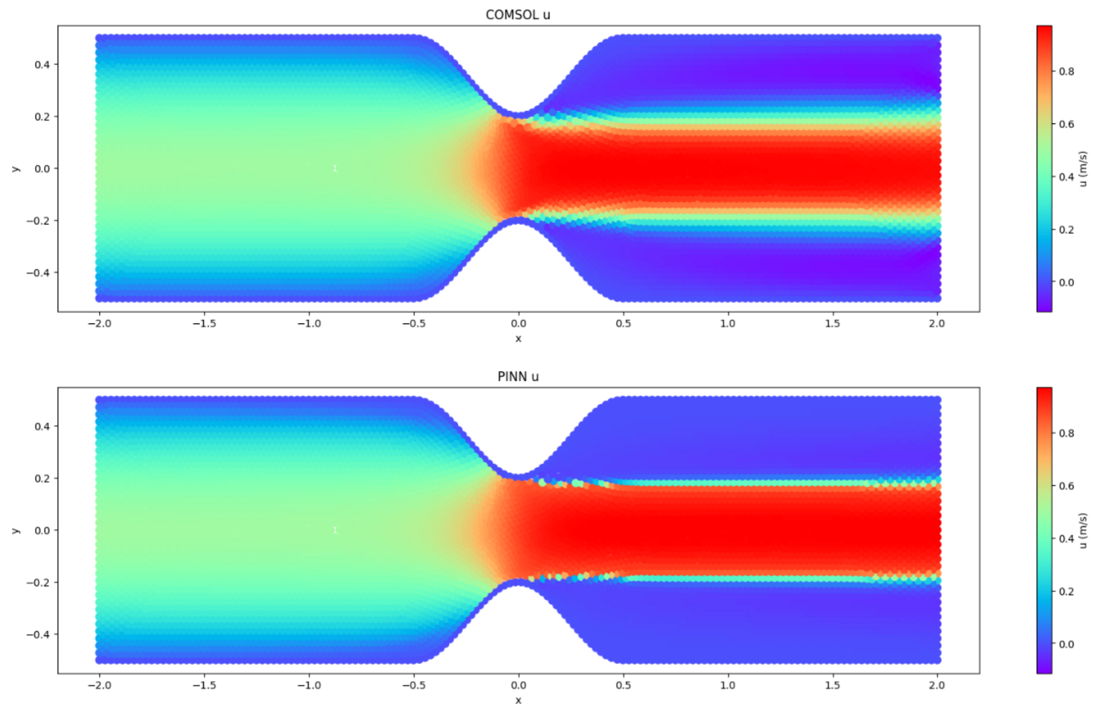


Figure 7.25: *u velocity in multi-case PINN vs FEM for Re=1000 and fc=0.3.*

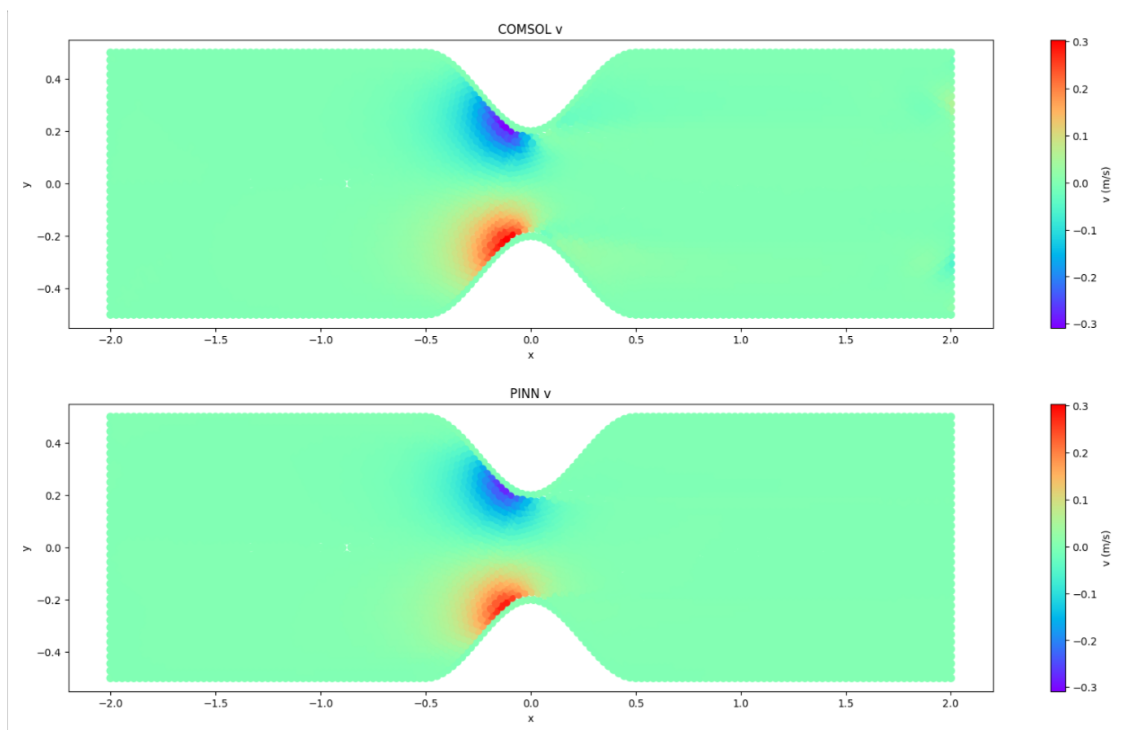


Figure 7.26: *v velocity in multi-case PINN vs FEM for Re=1000 and fc=0.3.*

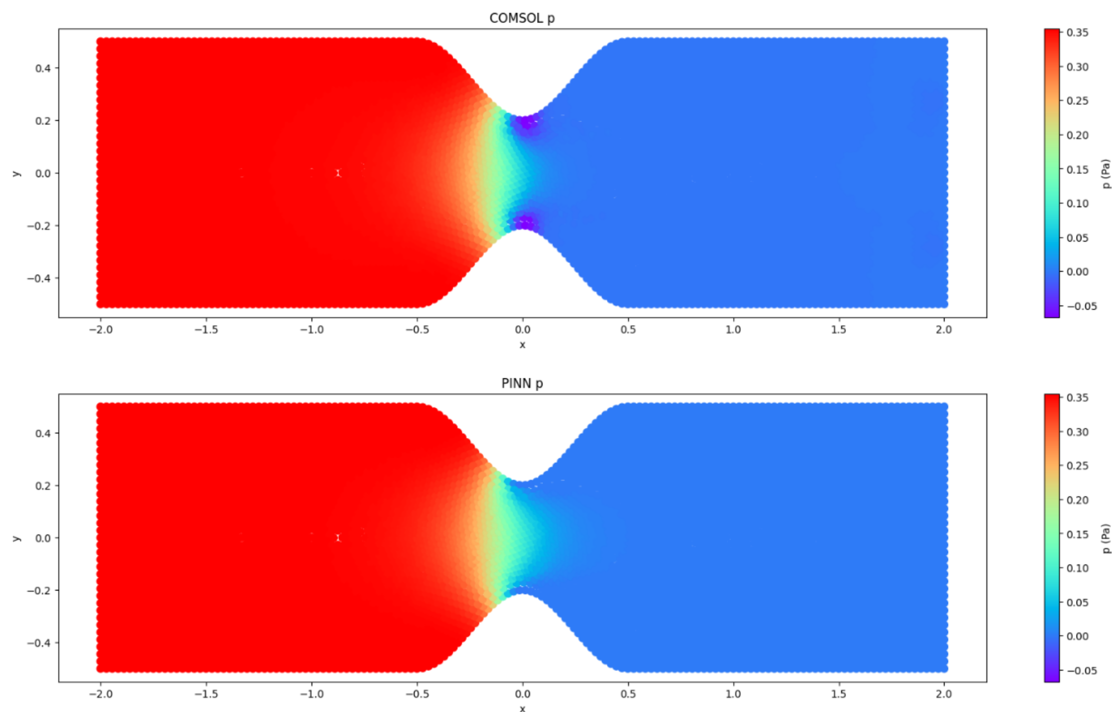


Figure 7.27: *pressure in multi-case PINN vs FEM for Re=1000 and fc=0.3.*

As observed in Fig.7.26 and Fig.7.27, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.11 Case study 10: Re=1250, fc =0.1

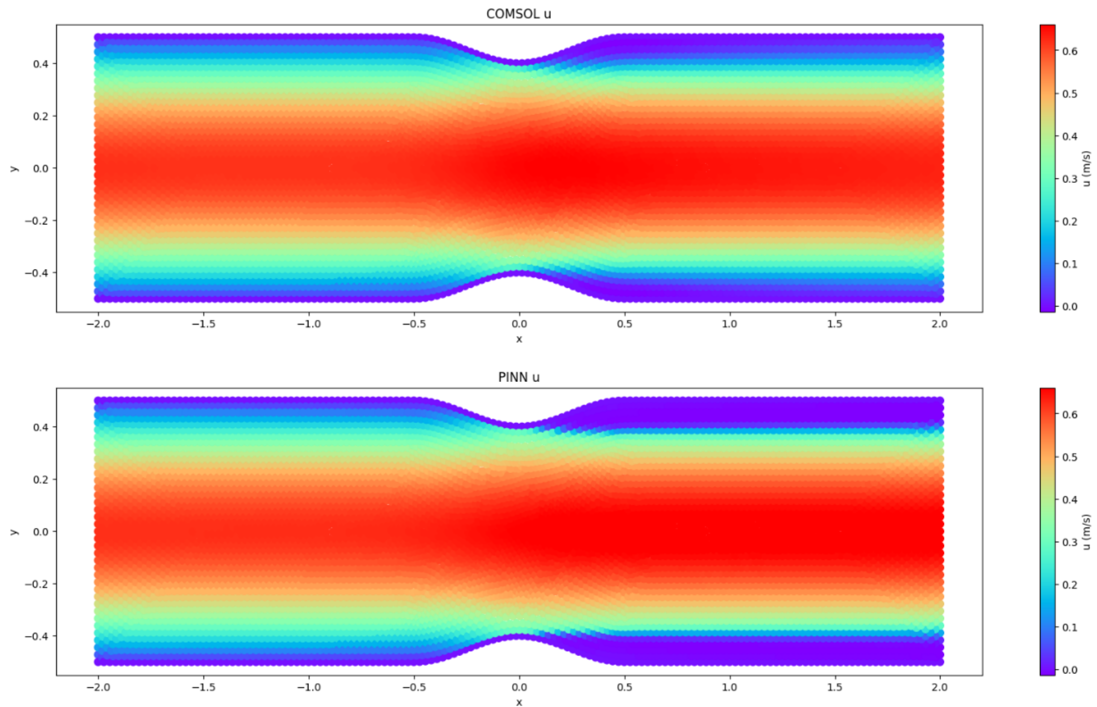


Figure 7.28: *u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.1.*

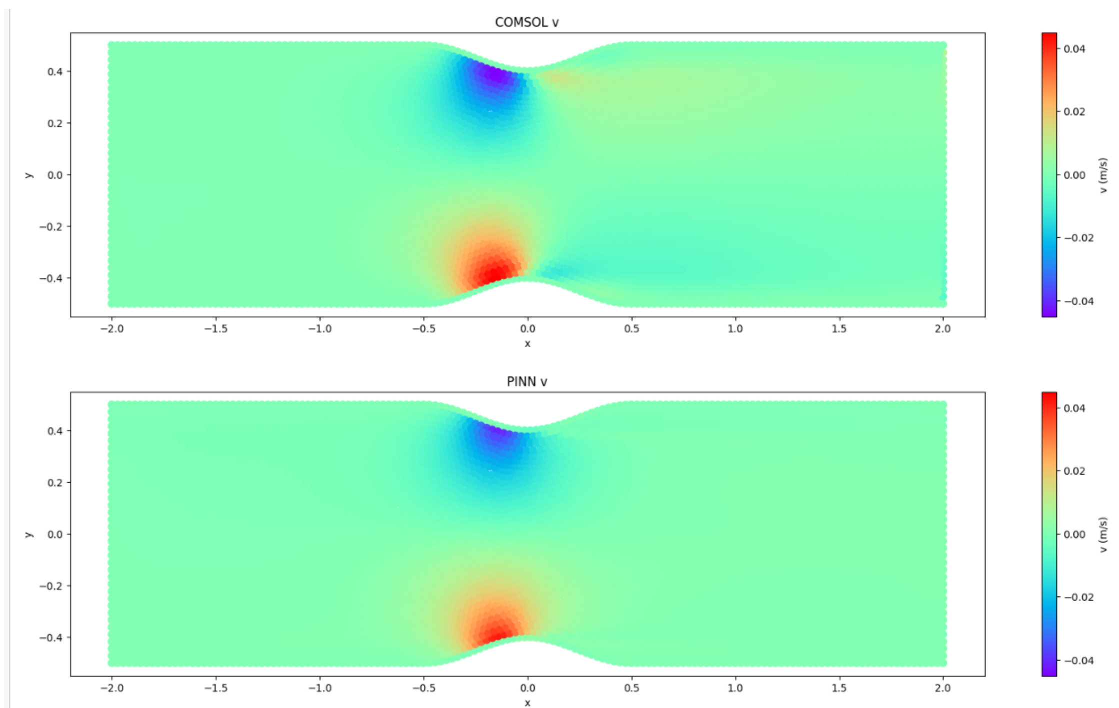


Figure 7.29: *v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.1.*

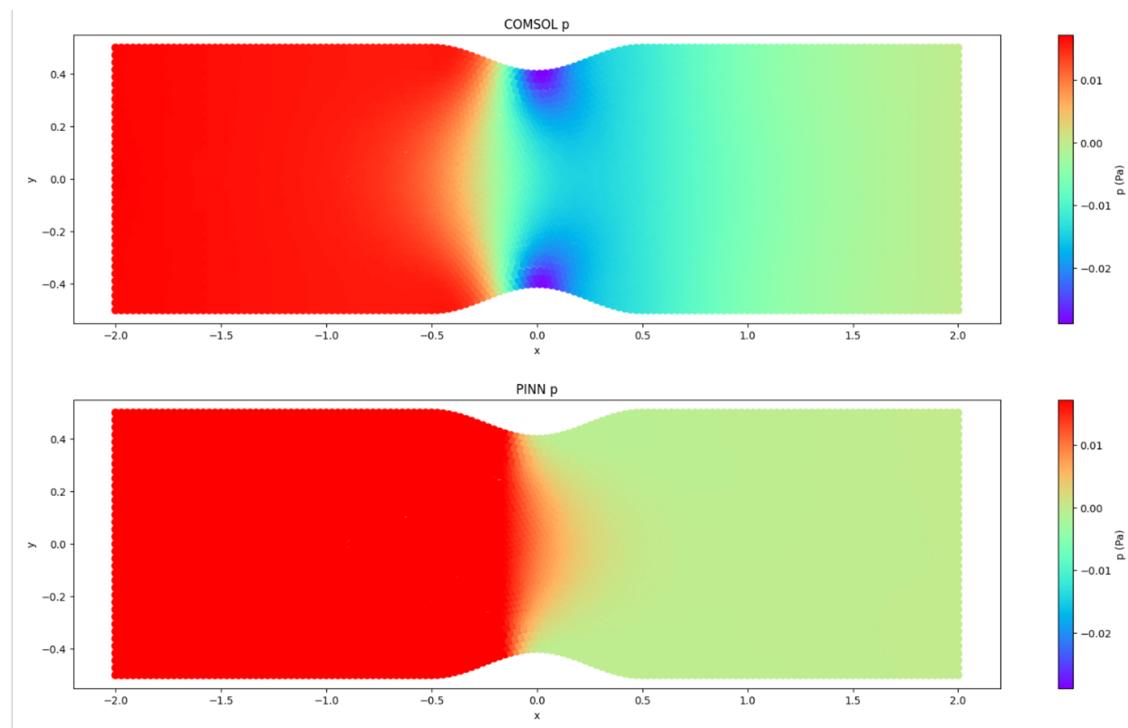


Figure 7.30: *pressure in multi-case PINN vs FEM for Re=1250 and fc=0.1.*

As illustrated in Fig.7.28, the PINN manages to accurately represent the u velocity field. However, in Fig.7.29, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.30, the PINN can not display the pressure drop, that is expected at the stenosis, as a result of the energy loss.

7.12 Case study 11: Re=1250, fc =0.2

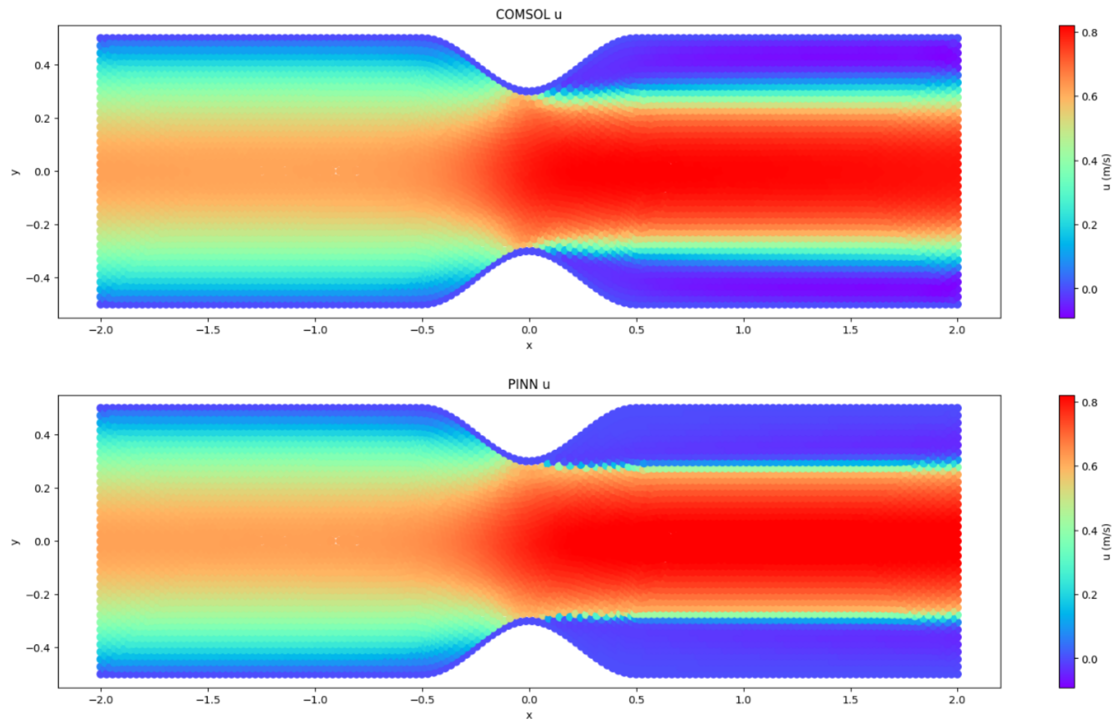


Figure 7.31: *u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.2.*

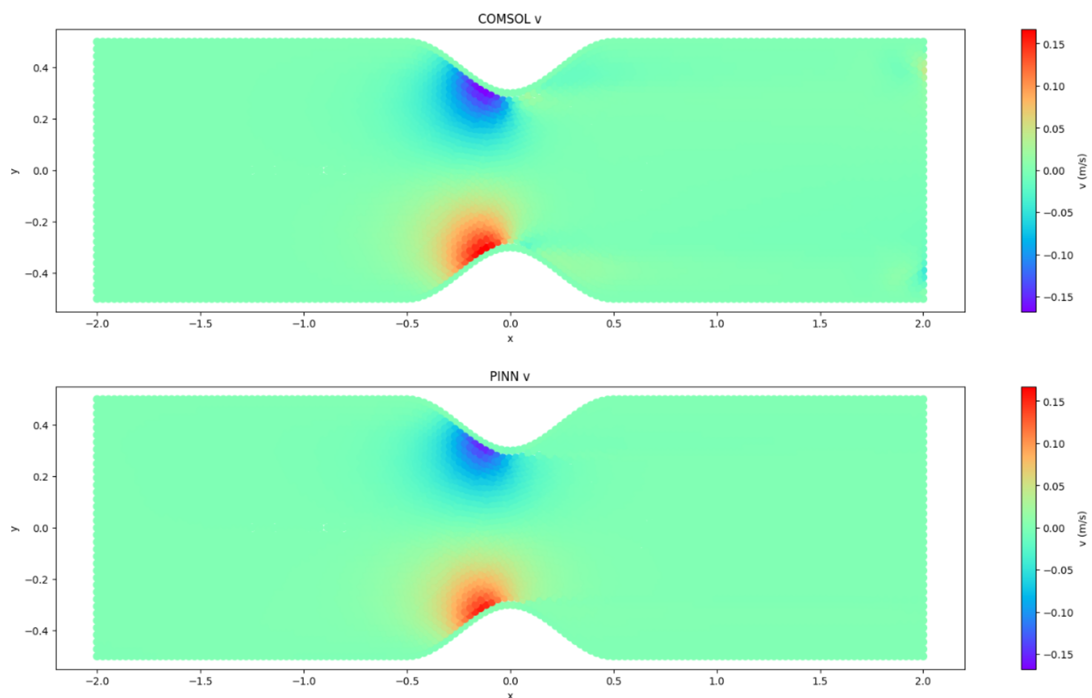


Figure 7.32: *v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.2.*

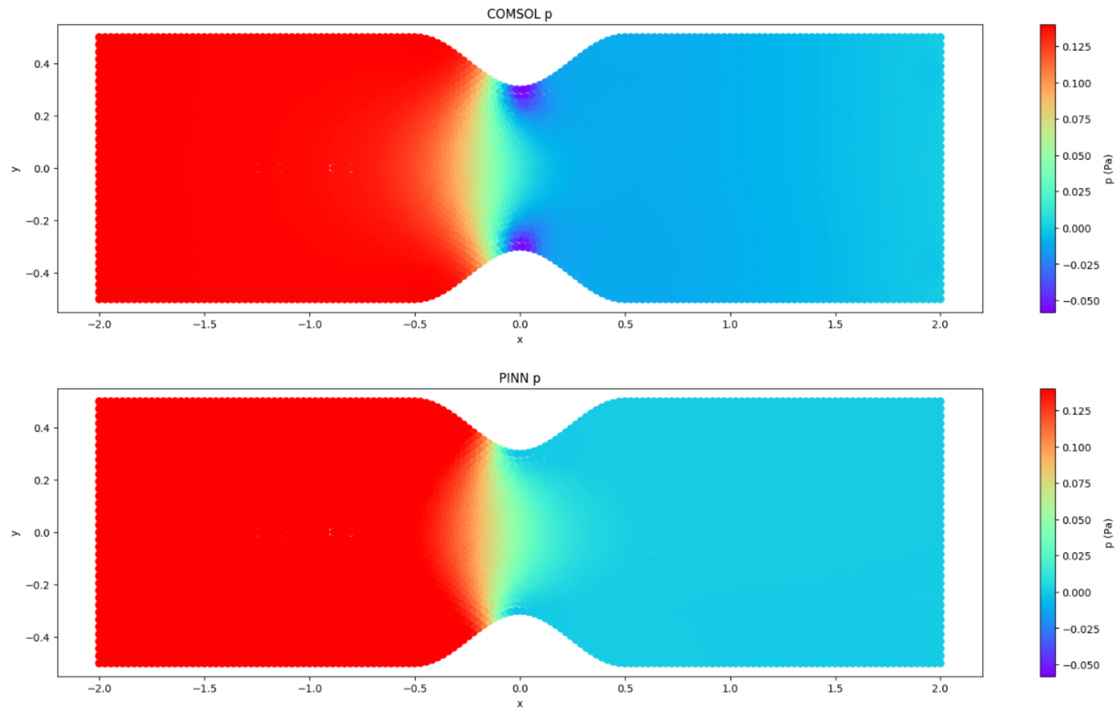


Figure 7.33: *pressure in multi-case PINN vs FEM for Re=1250 and fc=0.2.*

As observed in Fig.7.31, deep learning finds it difficult to represent the u velocity gradient after the stenosis. In Fig.7.32, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.33, the PINN displays the gradual pressure drop more accurately compared to the other case studies with the same fc.

7.13 Case study 12: Re=1250, fc =0.3

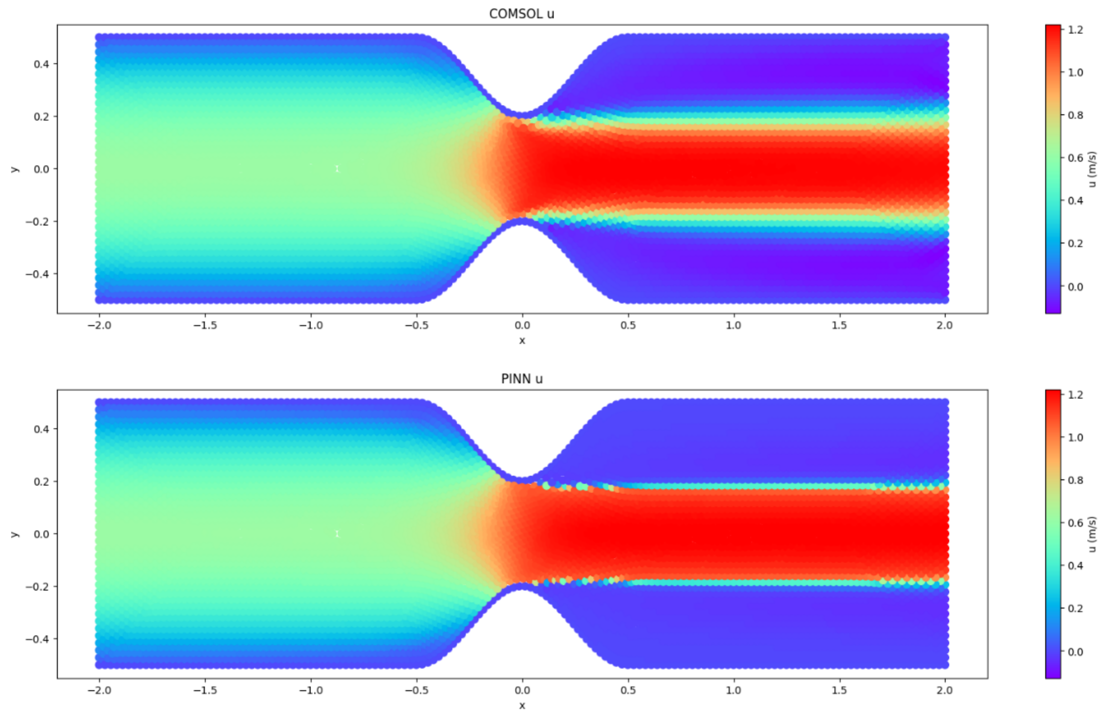


Figure 7.34: *u velocity in multi-case PINN vs FEM for Re=1250 and fc=0.3.*

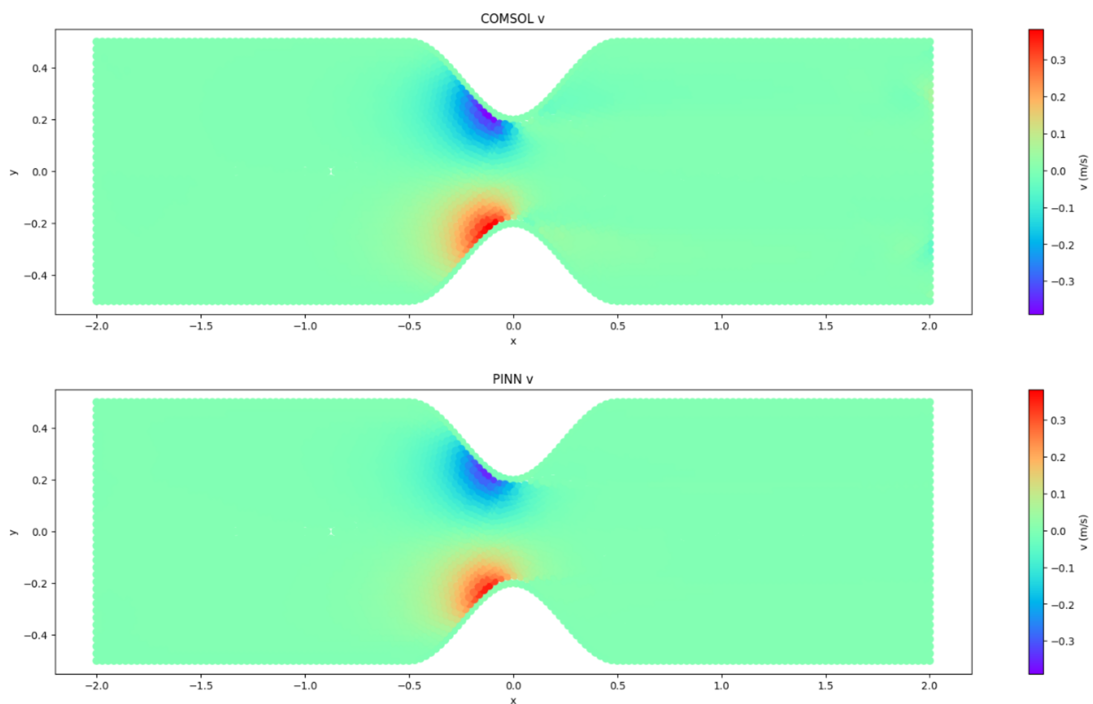


Figure 7.35: *v velocity in multi-case PINN vs FEM for Re=1250 and fc=0.3.*

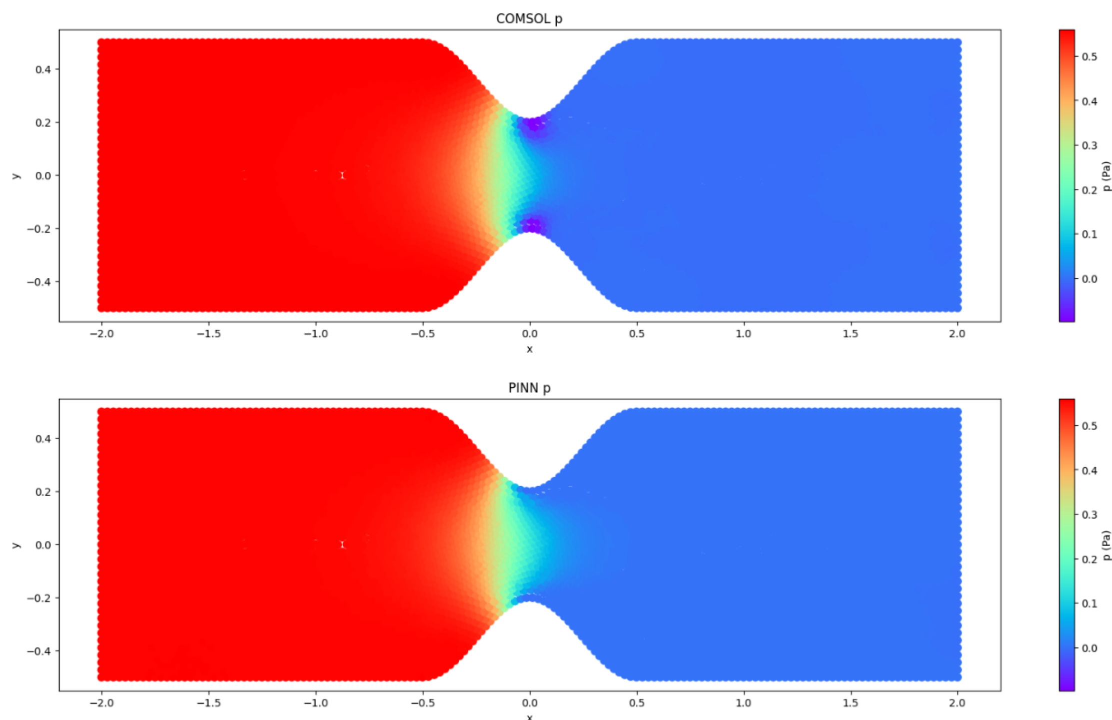


Figure 7.36: *pressure in multi-case PINN vs FEM for Re=1250 and fc=0.3.*

As displayed in Fig.7.35 and Fig.7.36, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.14 Case study 13: Re=1500, fc =0.1

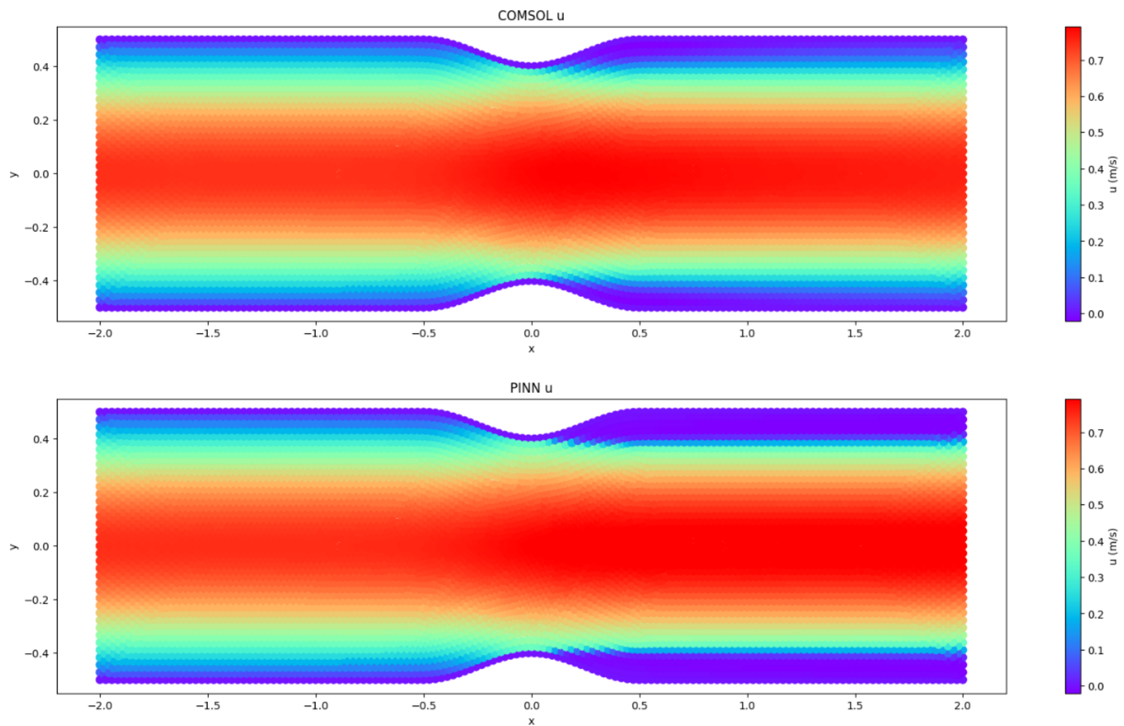


Figure 7.37: *u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.1.*

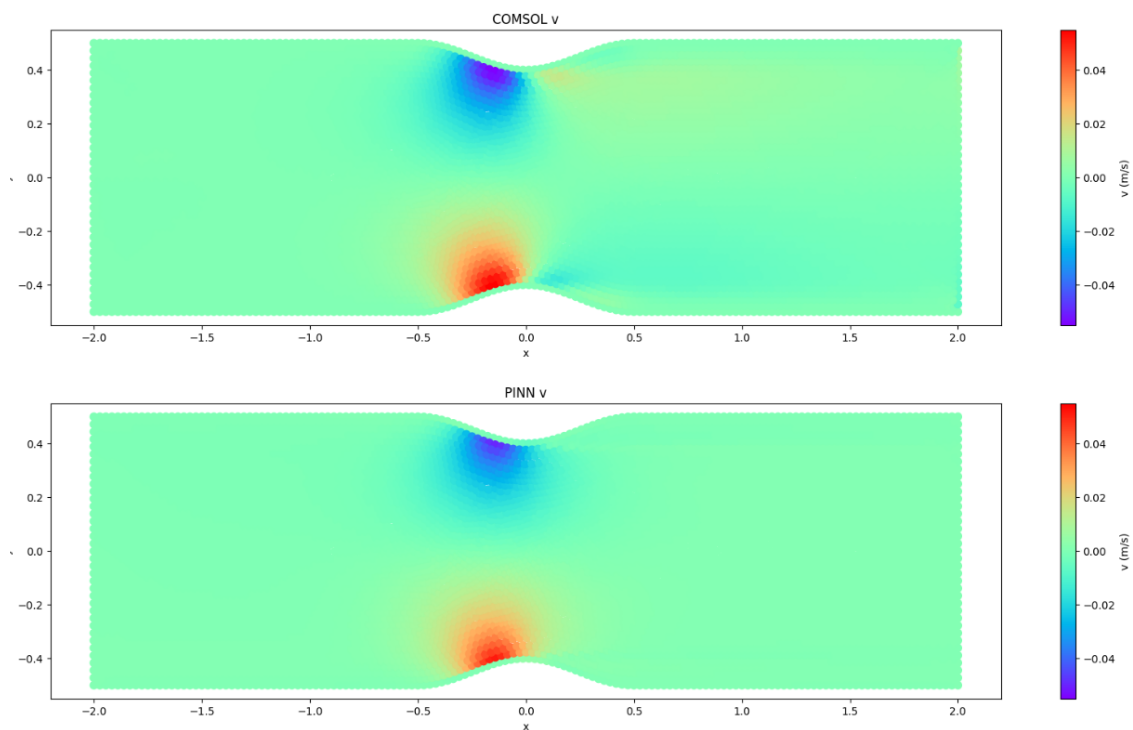


Figure 7.38: *v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.1.*

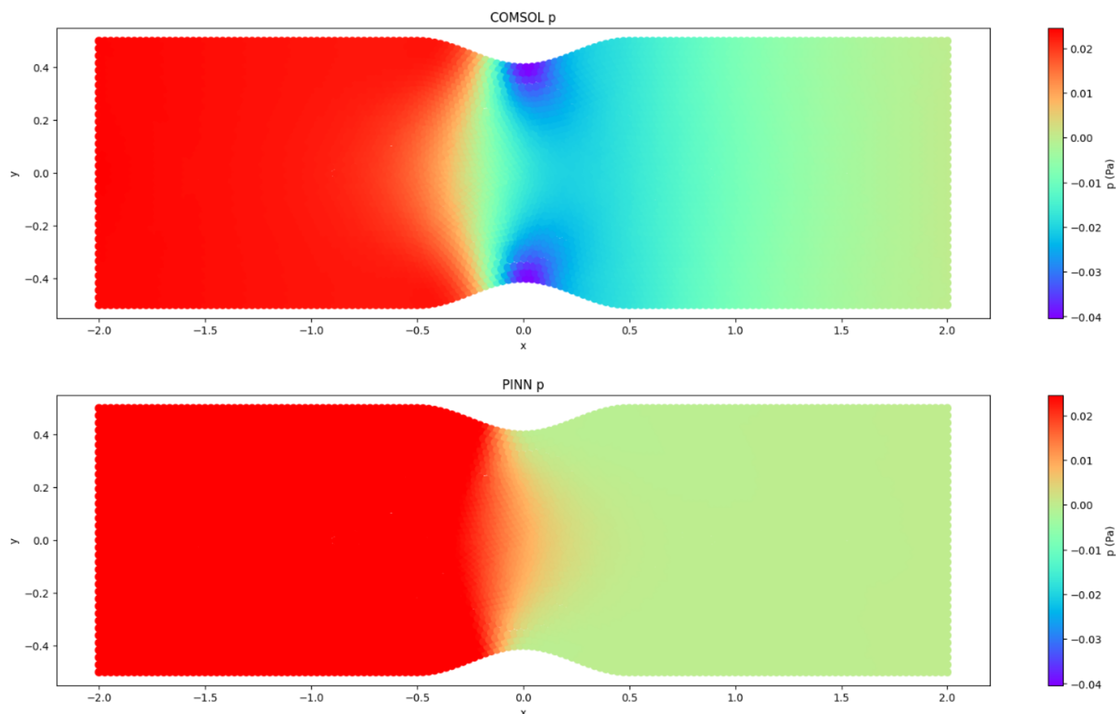


Figure 7.39: *pressure in multi-case PINN vs FEM for Re=1500 and fc=0.1.*

As illustrated in Fig.7.37, the PINN manages to accurately represent the u velocity field. However, in Fig.7.38, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.39, the PINN can not display the pressure drop, that is expected at the stenosis, as a result of the energy loss.

7.15 Case study 14: Re=1500, fc =0.2

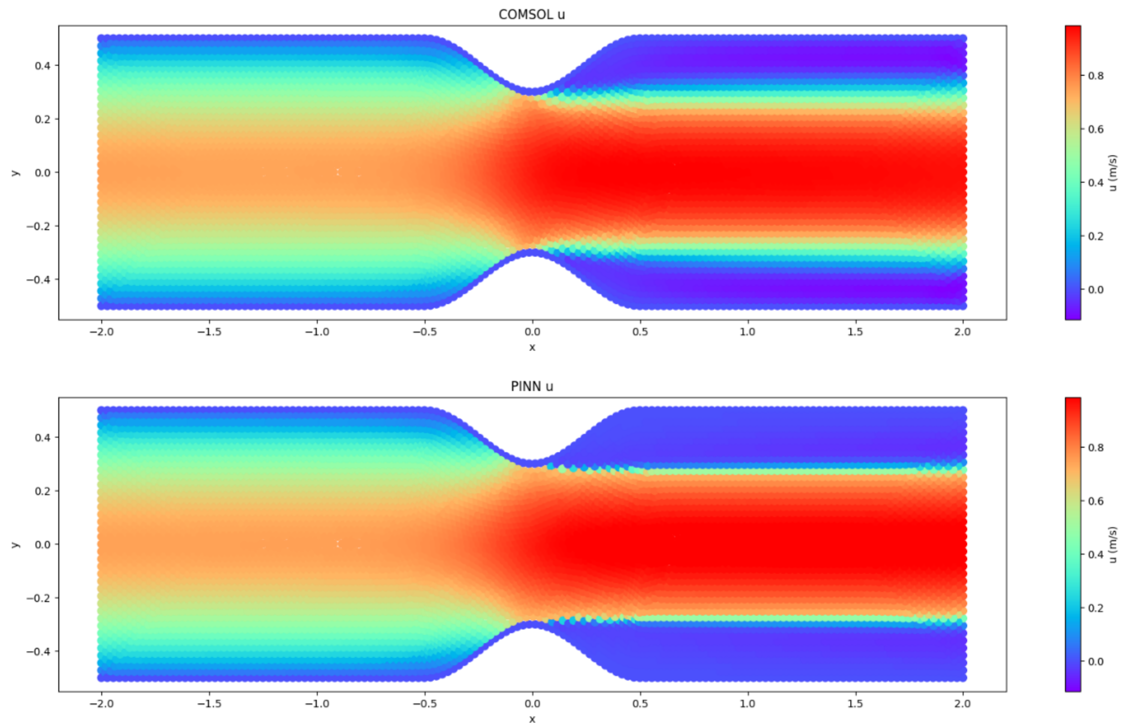


Figure 7.40: *u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.2.*

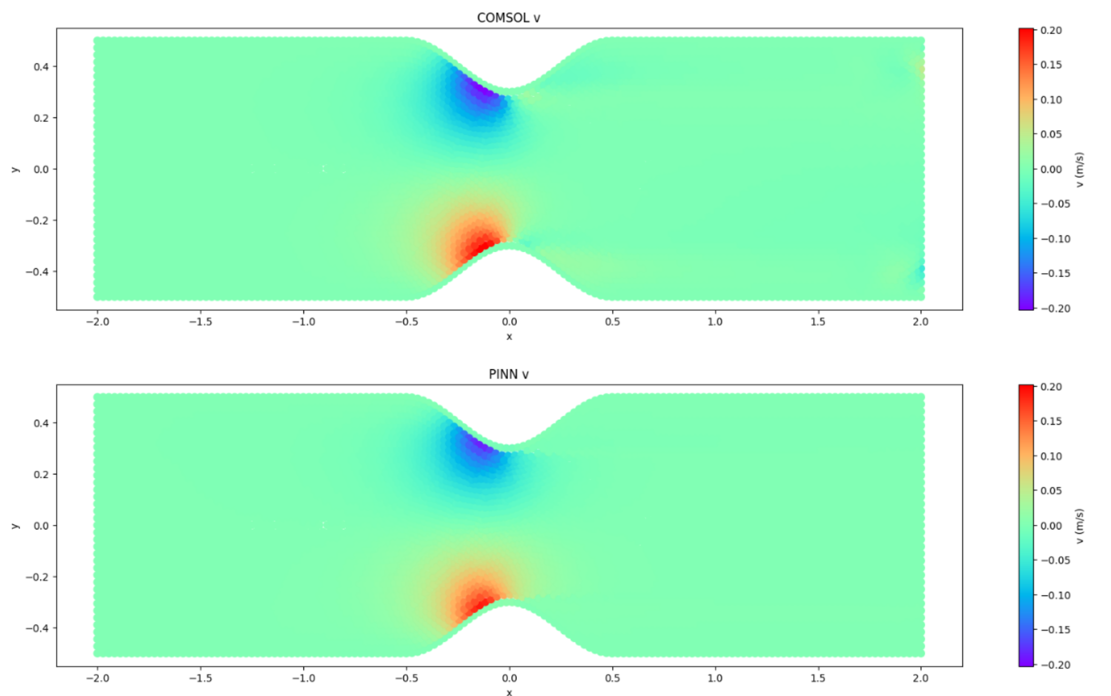


Figure 7.41: *v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.2.*

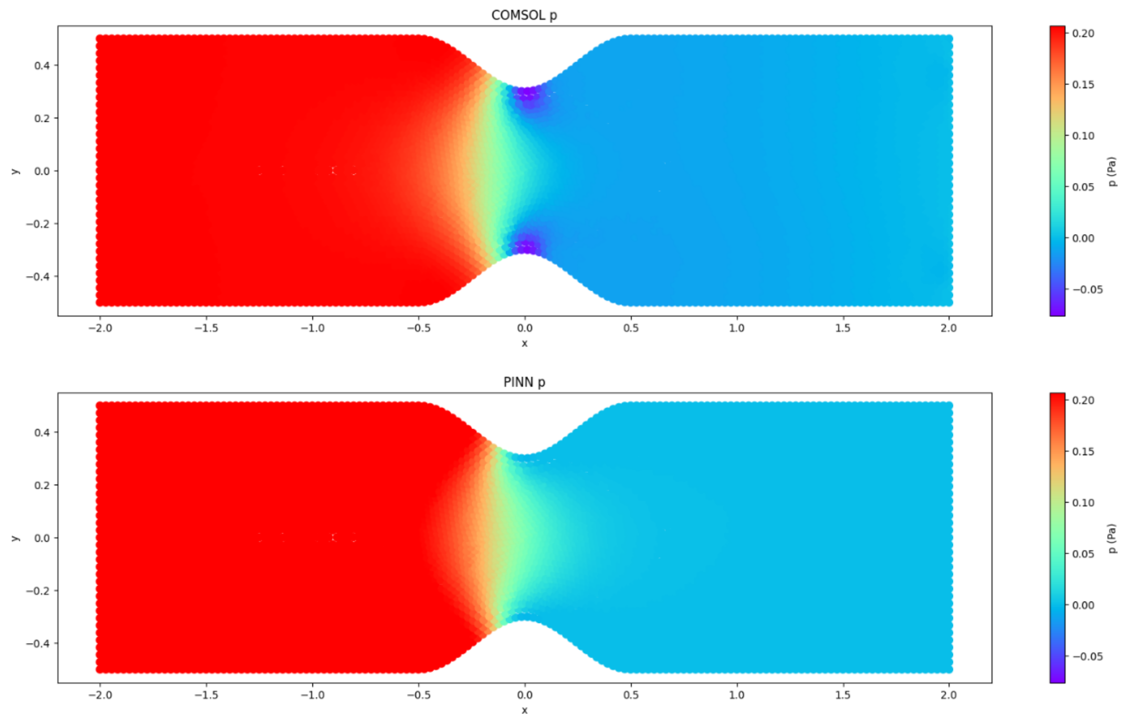


Figure 7.42: *pressure in multi-case PINN vs FEM for Re=1500 and fc=0.2.*

As observed in Fig.7.40, deep learning finds it difficult to represent the u velocity gradient after the stenosis. In Fig.7.41, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.42, the PINN displays the gradual pressure drop accurately.

7.16 Case study 15: Re=1500, fc =0.3

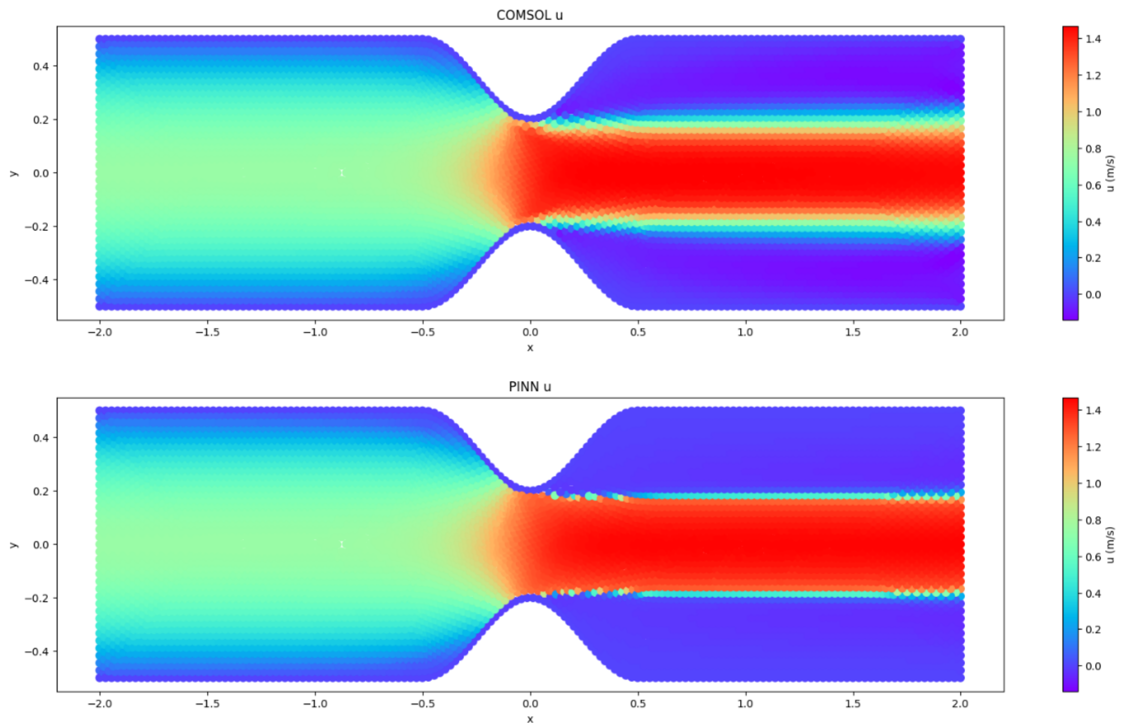


Figure 7.43: *u velocity in multi-case PINN vs FEM for Re=1500 and fc=0.3.*

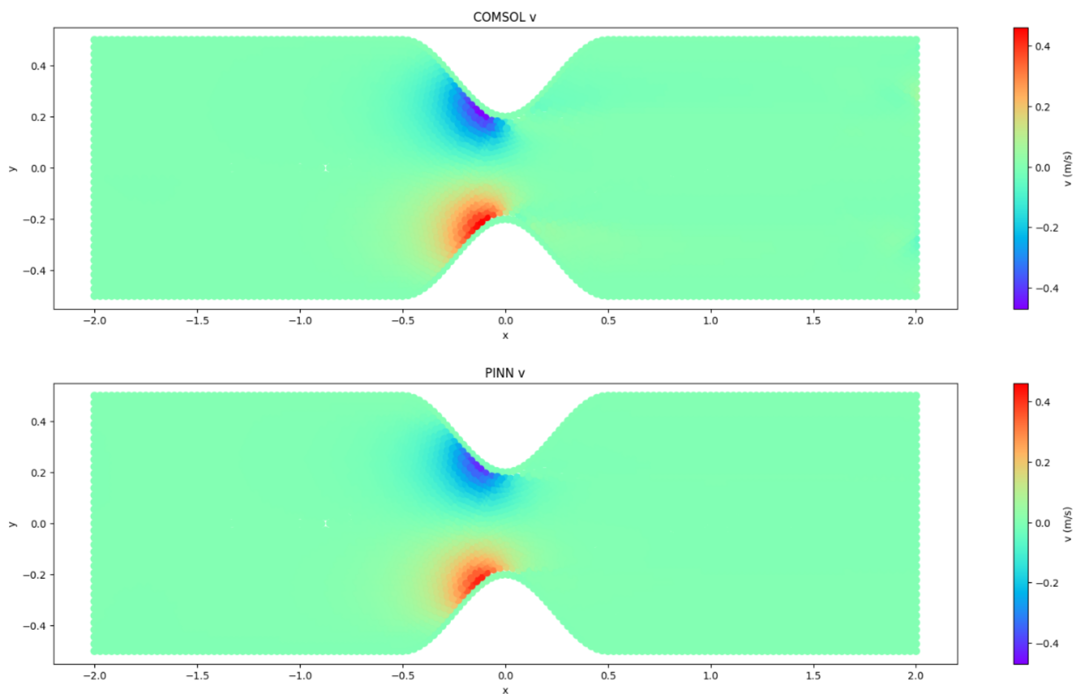


Figure 7.44: *v velocity in multi-case PINN vs FEM for Re=1500 and fc=0.3.*

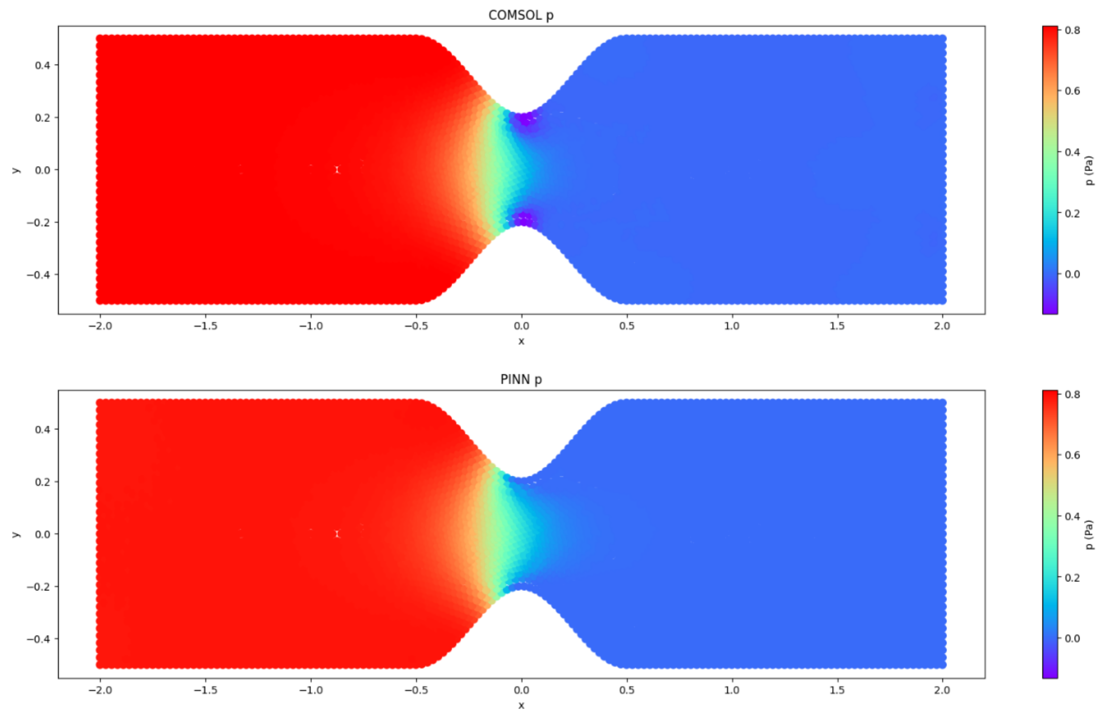


Figure 7.45: *pressure in multi-case PINN vs FEM for Re=1500 and fc=0.3.*

As displayed in Fig.7.44 and Fig.7.45, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.17 Case study 16: Re=1750, fc =0.1

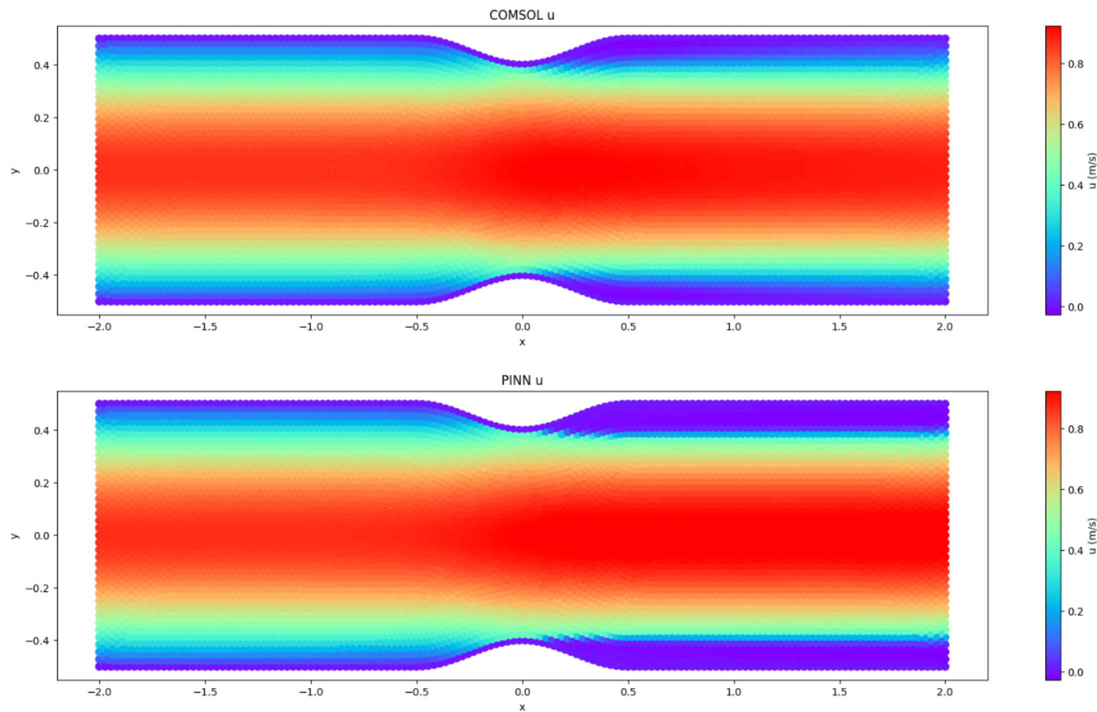


Figure 7.46: *u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.1.*

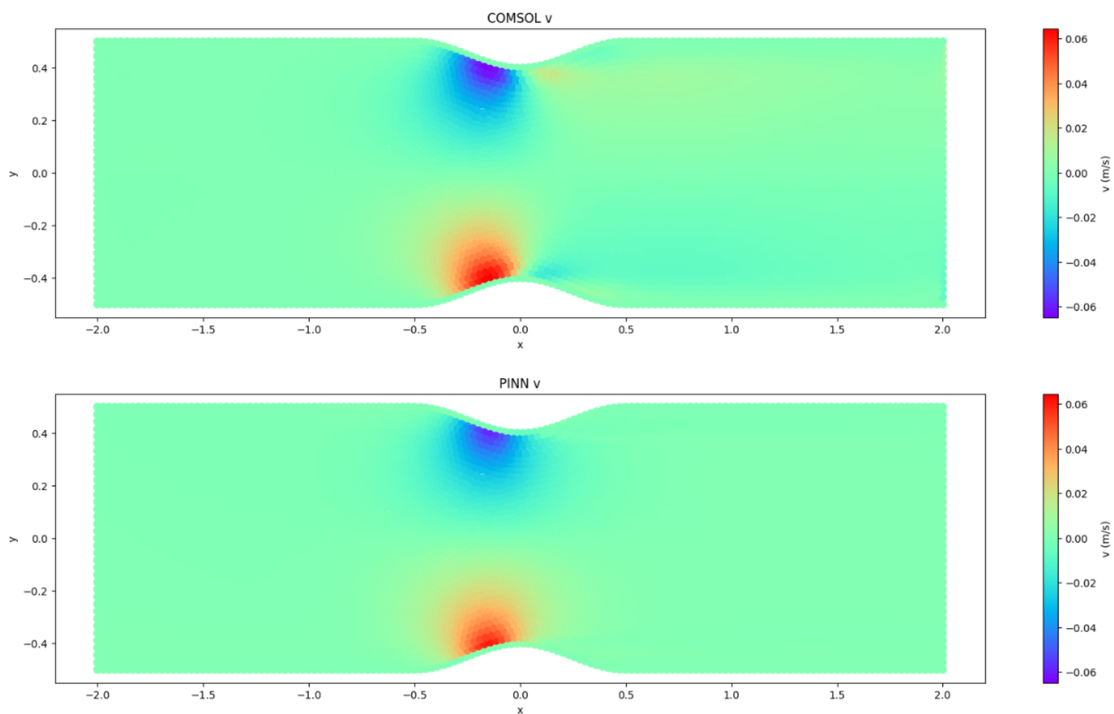


Figure 7.47: *v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.1.*

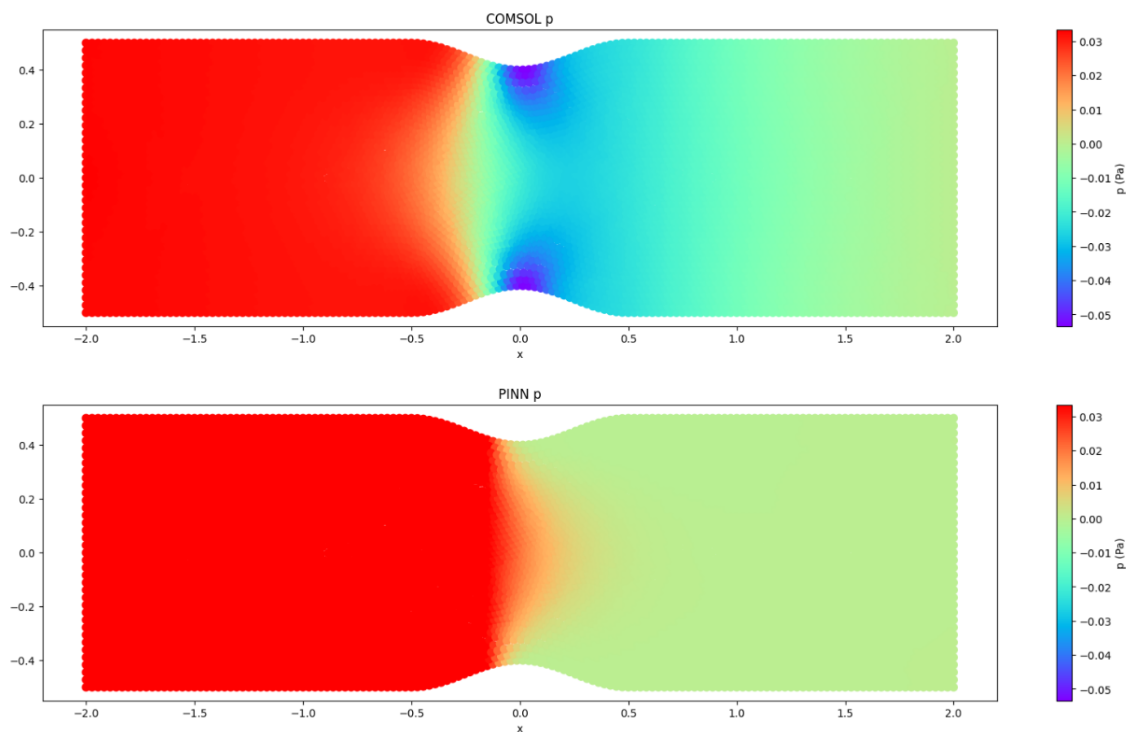


Figure 7.48: *pressure in multi-case PINN vs FEM for Re=1750 and fc=0.1.*

As illustrated in Fig.7.46, the PINN manages to accurately represent the u velocity field. However, in Fig.7.47, it is noticeable that deep learning is not able to capture the gradient of the v velocity after the stenosis. When it comes to pressure field Fig.7.48, the PINN can not display the pressure drop, that is expected at the stenosis, as a result of the energy loss.

7.18 Case study 17: Re=1750, fc =0.2

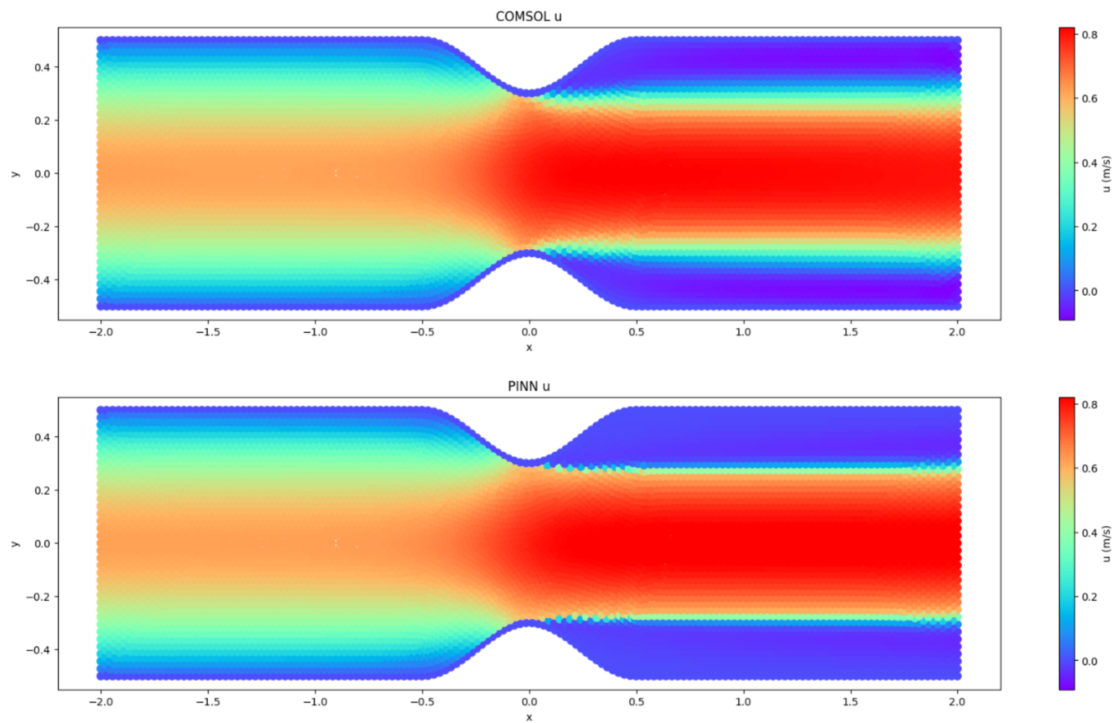


Figure 7.49: *u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.2.*

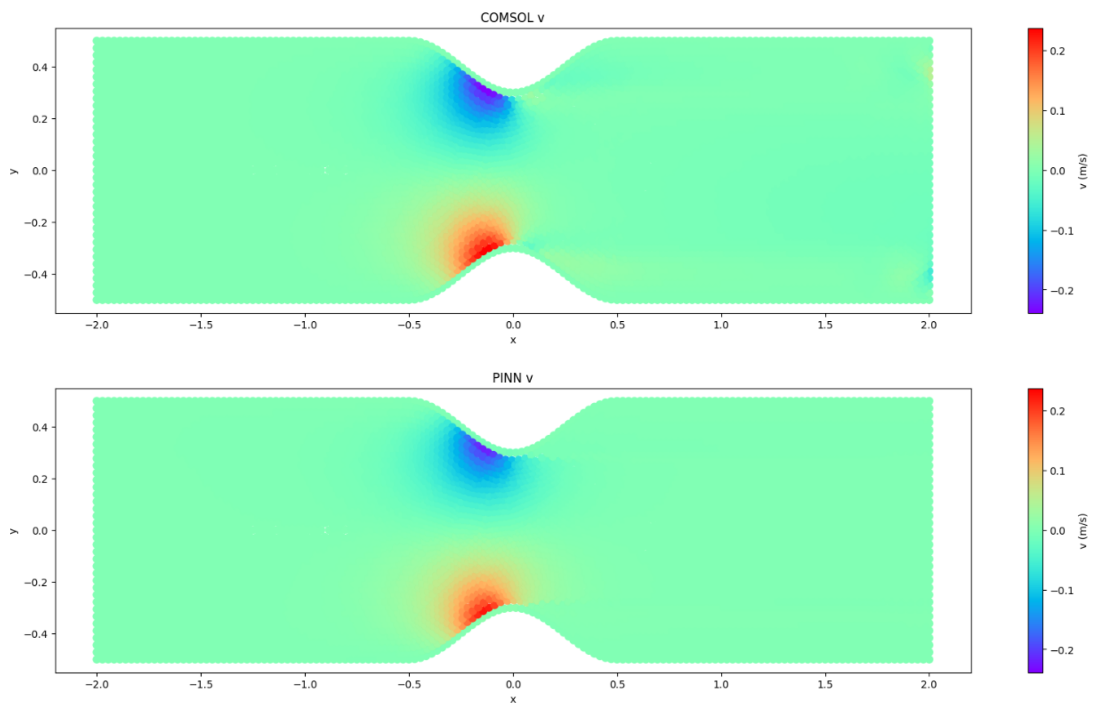


Figure 7.50: *v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.2.*

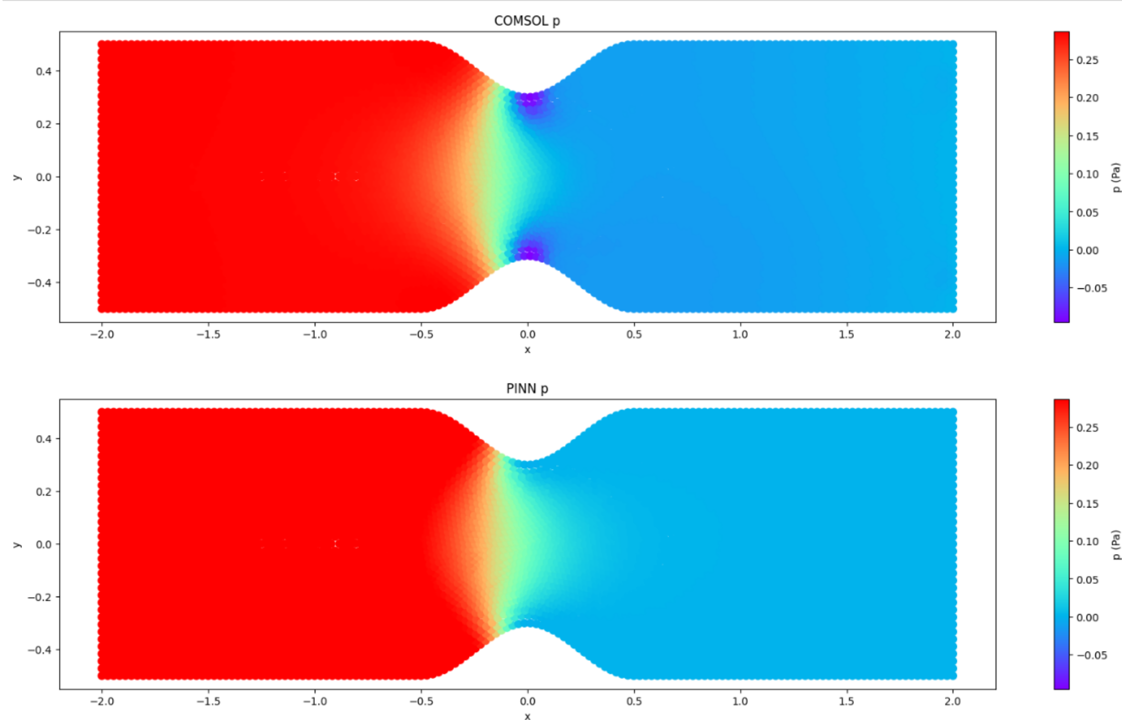


Figure 7.51: *pressure in multi-case PINN vs FEM for Re=1750 and fc=0.2.*

As observed in Fig.7.49, deep learning finds it difficult to represent the u velocity gradient after the stenosis. In Fig.7.50, the PINN displays very low errors at the v velocity field. Regarding the pressure field, Fig.7.51, the PINN can not display the gradual pressure drop.

7.19 Case study 18: Re=1750, fc =0.3

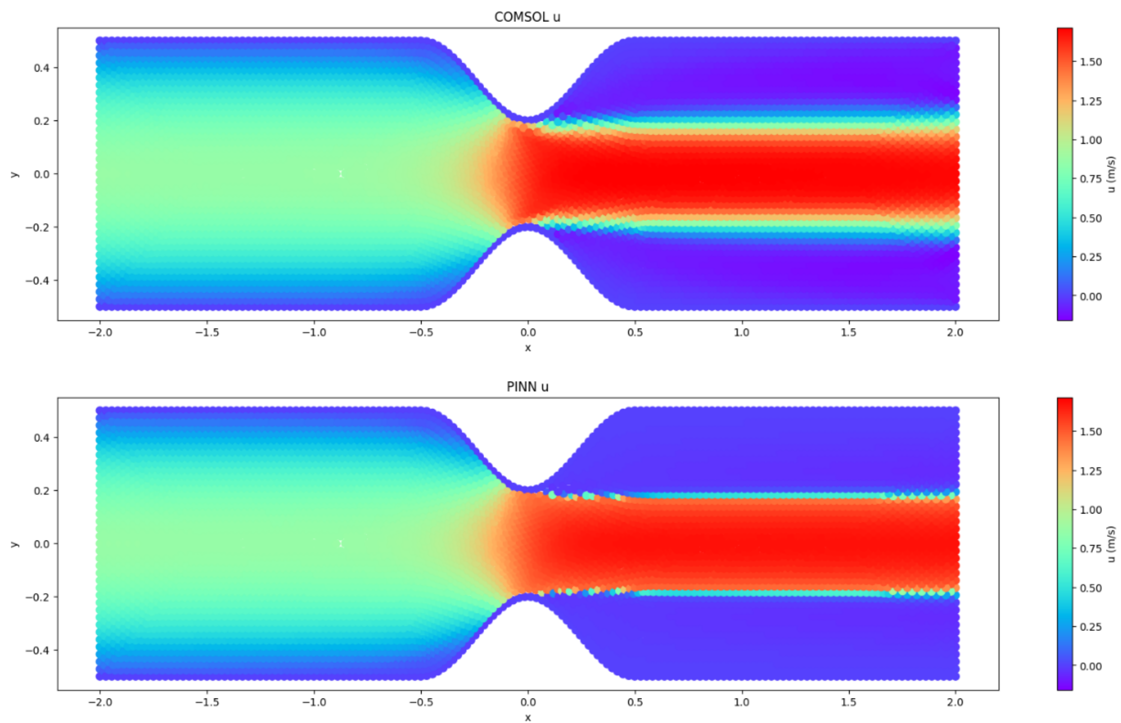


Figure 7.52: *u velocity in multi-case PINN vs FEM for Re=1750 and fc=0.3.*

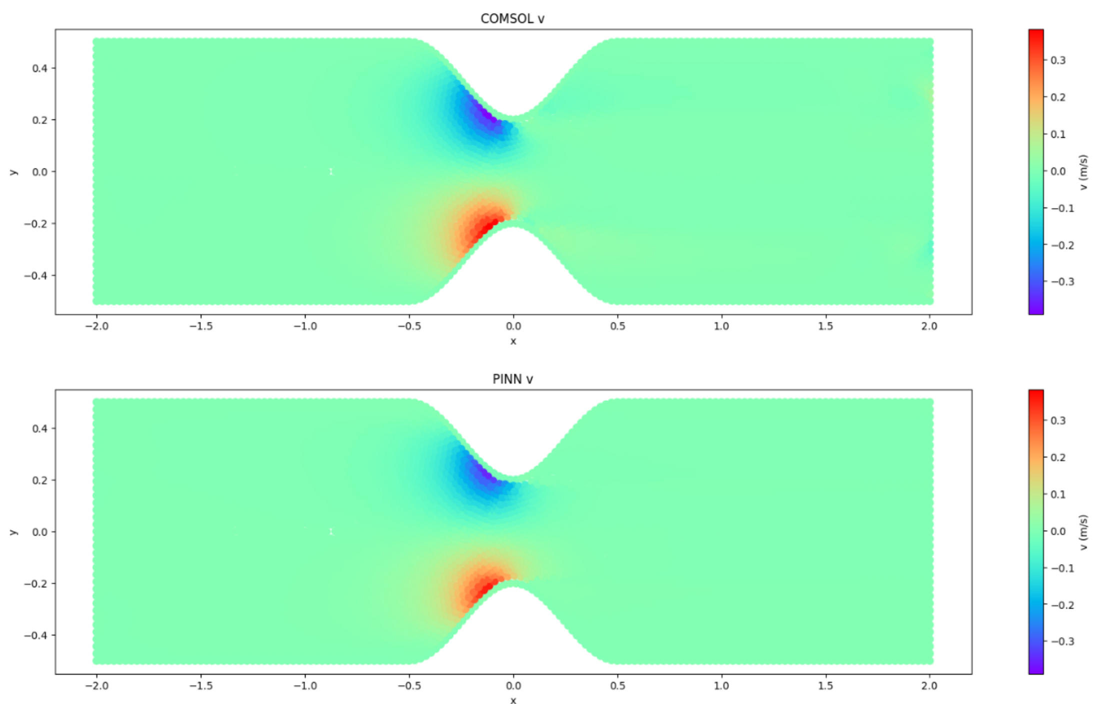


Figure 7.53: *v velocity in multi-case PINN vs FEM for Re=1750 and fc=0.3.*

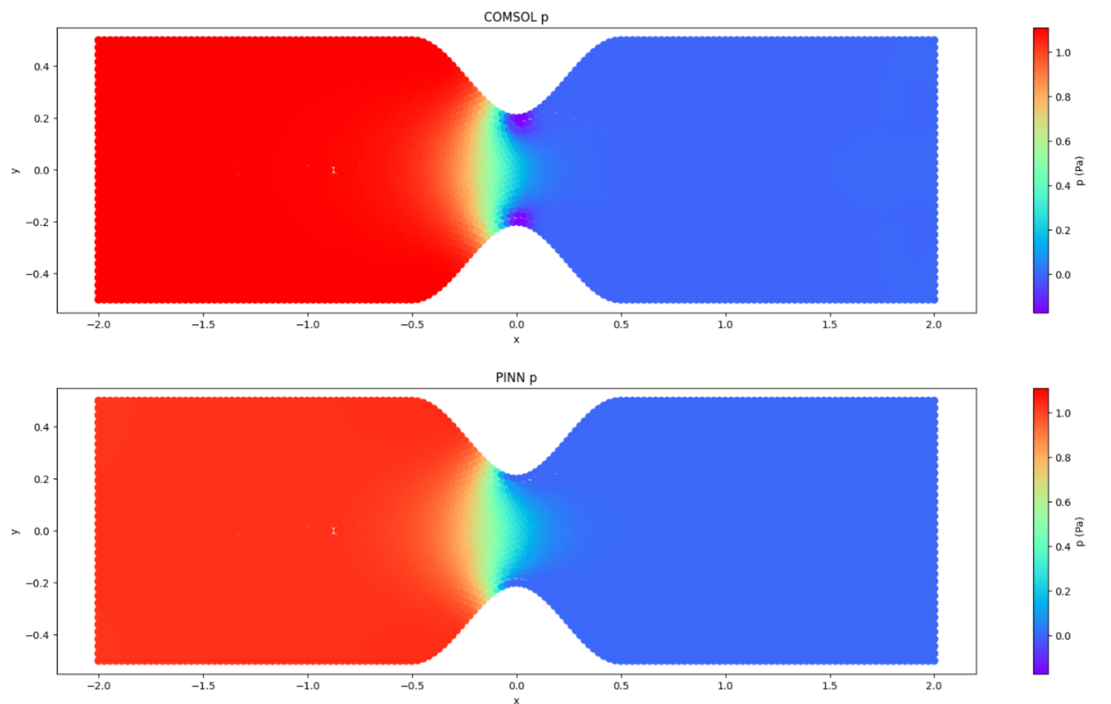


Figure 7.54: *pressure in multi-case PINN vs FEM for Re=1750 and fc=0.3.*

As displayed in Fig.7.53 and Fig.7.54, deep learning manages to capture accurately the v velocity and pressure fields. When it comes to u velocity field, again the errors are very low, yet, the gradient after the stenosis is not depicted correctly.

7.20 Errors PINNs vs FEM

Table 7.1: *Errors for the different studies.*

Study	Re	fc	RMSE u	Norm RMSE u	RMSE v	Norm RMSE v	RMSE p	Norm RMSE p
1	500	0.1	0.0147	0.0548	0.0016	0.0427	0.0019	0.2246
2	500	0.2	0.0203	0.0576	0.0027	0.0222	0.0060	0.1893
3	500	0.3	0.0356	0.0636	0.1893	0.0197	0.0101	0.0997
4	750	0.1	0.0180	0.0444	0.0021	0.0387	0.0050	0.2830
5	750	0.2	0.0266	0.0497	0.0035	0.0189	0.0091	0.1339
6	750	0.3	0.0480	0.0573	0.0073	0.0170	0.0149	0.0667
7	1000	0.1	0.0227	0.0418	0.0026	0.0362	0.0097	0.3299
8	1000	0.2	0.0347	0.0476	0.0044	0.0173	0.0136	0.1153
9	1000	0.3	0.0660	0.0607	0.0062	0.0101	0.0078	0.0184
10	1250	0.1	0.0272	0.0403	0.0031	0.0348	0.0158	0.3424
11	1250	0.2	0.0459	0.0503	0.0046	0.0138	0.0161	0.0812
12	1250	0.3	0.0808	0.0599	0.0072	0.0093	0.0106	0.0161
13	1500	0.1	0.0319	0.0393	0.0035	0.0322	0.0229	0.3531
14	1500	0.2	0.0550	0.0500	0.0055	0.0135	0.0211	0.0746
15	1500	0.3	0.0947	0.0588	0.0085	0.0091	0.0195	0.0206
16	1750	0.1	0.0363	0.0382	0.0040	0.0309	0.0311	0.3581
17	1750	0.2	0.0660	0.0512	0.0069	0.0144	0.0273	0.0714
18	1750	0.3	0.1085	0.0579	0.0103	0.0095	0.0526	0.0410

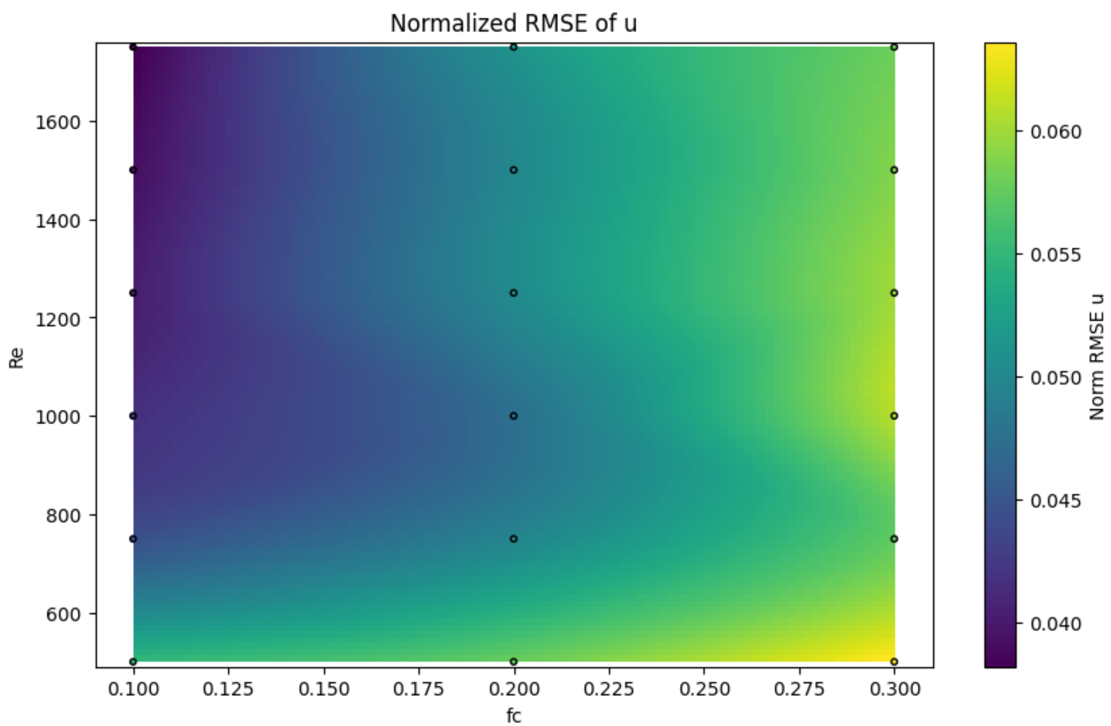


Figure 7.55: *Norm RMSE in u velocity.*

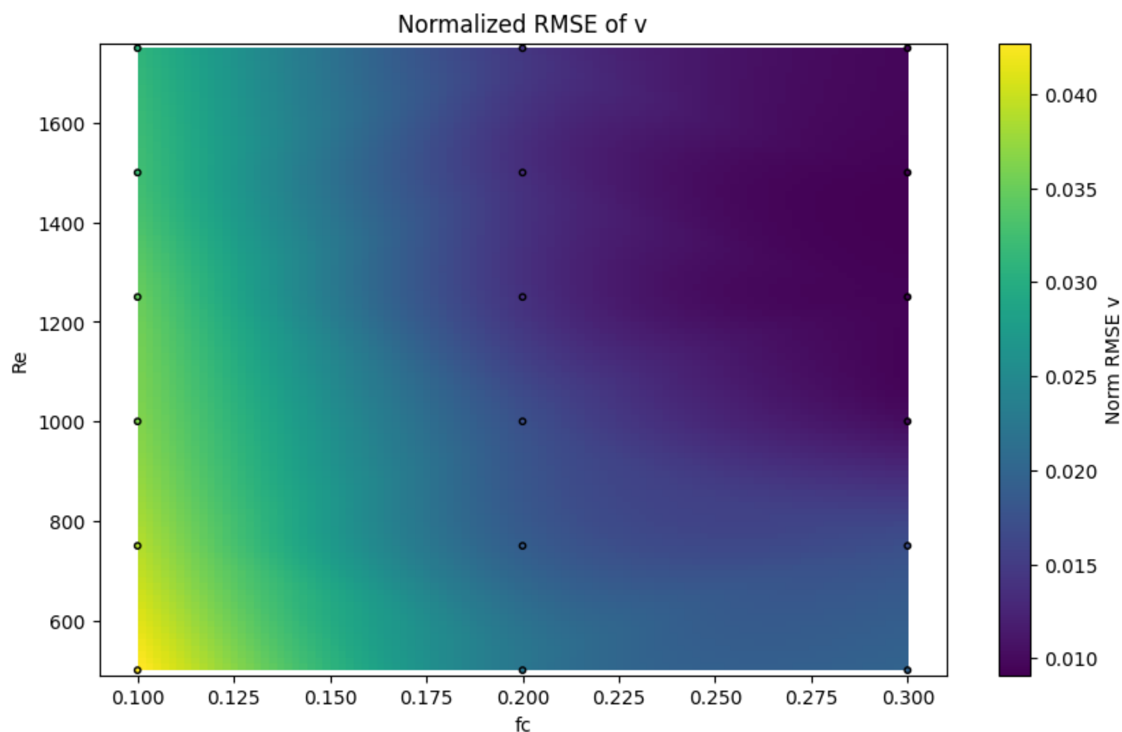


Figure 7.56: *Norm RMSE in v velocity.*

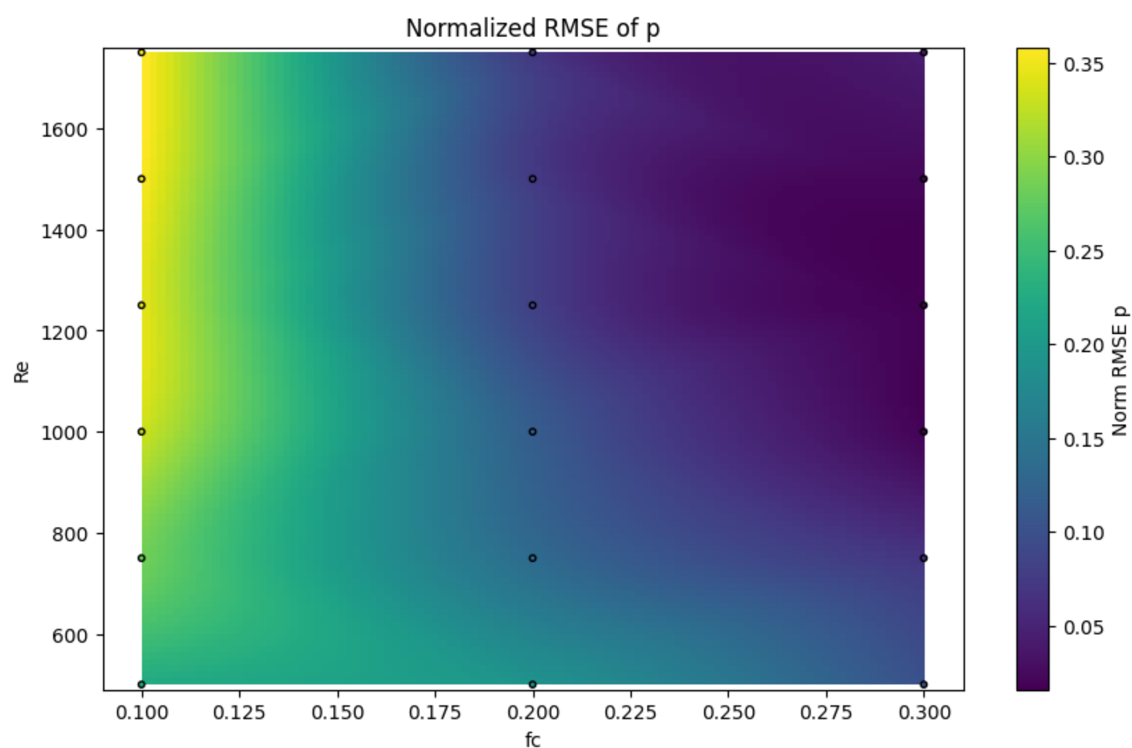


Figure 7.57: *Norm RMSE in p pressure.*

In general, errors in the velocity fields were very low. As it was observed u velocity errors slightly increase at higher Reynolds numbers and degrees of stenosis, while v velocity errors seem to increase at lower Reynolds numbers and degrees of stenosis. Additionally

results indicate that pressure prediction accuracy drops for bigger Reynolds numbers but increases for bigger degrees of stenosis. It should be considered that errors in general were slightly bigger than expected from the bibliography and that is because of the extended range of Reynolds numbers and degrees of stenosis that the network had to learn to predict. Moreover, it should be noted that the inspiration study [12], which exhibits very low errors, investigates a multi-case PINN, that predicts the velocity and pressure fields for a constant low Reynolds number and for gradual stenoses that do not reach high degrees of stenosis, as opposed to our work. It is important to mention that the inaccuracies in velocity and pressure fields seem to be consistent and appear at the same regions. More specifically it is evident that while the degree of stenosis grows, the gradient of u velocity at the height of the stenosis seems to be increasingly ignored after the narrowing. Regarding the pressure it is observable that at low degrees of stenosis the network fails to represent the pressure drop precisely.

Conclusion

Vascular hemodynamics are often studied in similar patient-specific anatomies. Following the conventional methodology, a separate CFD simulation must be performed for every case. Despite being a well-optimized and accurate procedure, it still takes several hours to prepare the geometry, create a mesh, setup and perform the numerical simulation. This Thesis explores a special category of neural networks that integrate PDEs into the training process, known as PINNs. Specifically, we investigated whether PINNs can be generalized, so that they can predict the solutions of a PDE for different boundary conditions and in different geometries, without the need for training data. We employed multi-case PINNs to solve the 2D Navier–Stokes equations in simplified stenotic vessel geometries with varying degrees of stenosis and for different Reynolds numbers, creating surrogate models of blood flow. We analyzed how different activation functions, optimizers, network architectures and point samplings influence the accuracy and performance of the multi-case PINNs. Additionally, we examined how adaptive activation functions and TSPs could enhance these models. The main findings of the Thesis are:

- Implementing TSPs improves accuracy and reduces computation time per step.
- SiLU and tanh produce the lowest overall loss compared to other activation functions.
- The most efficient architecture is a network with 6 hidden layers and 512 neurons per hidden layer, achieving the lowest loss in the shortest time.
- While RMSProp offers better convergence rates, its high computation time makes Adam the most practical optimizer.
- Adaptive activation functions do not consistently improve performance, as their effectiveness depends on the network architecture.
- A higher number of sampling points improves convergence but increases the time per step.
- Single-case PINNs are less effective than CFD simulations due to worse convergence rates.
- Multi-case PINNs, while generally performing worse than CFD simulations, enable real-time predictions.
- Velocity errors are very low overall.
- u velocity errors increase with higher Reynolds numbers and degrees of stenosis.

- v velocity errors increase at lower Reynolds numbers and lower degrees of stenosis.
- Pressure prediction accuracy declines at higher Reynolds numbers but improves with greater degrees of stenosis.

In summary, multi-case PINNs enable a single training process to handle various geometries and flow conditions, offering real-time predictions. Future work should extend these models to 3D dimensions using patient-specific vascular models and explore advanced unsupervised learning techniques that could surpass the performance of the PINNs used in this study.

Appendix

Appendix A

Python code

```python
class Stenosis(torch.nn.Module):
    def __init__(self, r, l):
        super().__init__()
        self.register_buffer("r", torch.tensor(r), persistent=False
            )
        self.register_buffer("l", torch.tensor(l), persistent=False
            )  # length of line

    def forward(self, x):
        x_ref = x[..., 0:1]
        y_ref = x[..., 1:2]
        fc = x[..., 2:3]

        # Transform x_ref to range [-1, 1] for the cosine function
        x_ref_transformed = 2 * (x_ref - self.l/8) / (3*self.l/8 -
            self.l/8) - 1


        # Create masks for the piecewise function
        mask1 = (x_ref >= -self.l/2) & (x_ref < -self.l / 8)
        mask2 = (x_ref >= -self.l / 8) & (x_ref <  self.l / 8)

        # Compute y_case based on the interval
        y_case = torch.where(
            mask1,
            y_ref,
            torch.where(
                mask2,
                y_ref * (1 - fc * (1 + torch.cos(x_ref_transformed
                    * torch.pi))),
                y_ref
            )
        )

        return torch.cat((x_ref, y_case), -1)
```

Listing A.1: Stenosis tranformation class

With the help of the Stenosis class we can define the different stenosis geometries with respect to fc.

```
class GE(torch.nn.Module):
    def __init__(self, r, l):
        super().__init__()
        r = torch.tensor(r)
        self.register_buffer("r", r, persistent=False)
        l = torch.tensor(l)
        self.register_buffer("l", l, persistent=False)  # length of
            line

    def forward(self, x):
        x_case = x[..., 0:1]
        y_case = x[..., 1:2]
        fc = x[..., 2:3]

        # Transform x_ref to range [-1, 1] for the cosine function
        x_ref_transformed = 2 * (x_case - self.l/8) / (3*self.l/8 -
            self.l/8) - 1

        # Create masks for the piecewise function
        mask1 = (x_case >= -self.l/2) & (x_case < -self.l / 8)
        mask2 = (x_case >= -self.l / 8) & (x_case < self.l / 8)

        radius = torch.where(
            mask1,
            y_case / self.r,
            torch.where(
                mask2,
                y_case / (1 - fc * (1 + torch.cos(x_ref_transformed
                    * torch.pi))) /self.r,
                y_case / self.r
            )
        )

        cline = 2. * x_case / self.l



        return torch.cat((cline, radius), -1)

class Dissq(torch.nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self,x):
        return 1.-x[...,0:1]**2.


```

```
class warped(torch.nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self,x):
        return x[...,2:4]-x[...,0:2]


class Newcoord(torch.nn.Module):
    def __init__(self,s):
        super().__init__()
        s=torch.tensor(s)
        self.register_buffer("s", s, persistent=False)
    def forward(self,x):
        return self.s*x[...,0:3]
```

Listing A.2: Classes that produced the tube specific coordinates

By utilizing the above classes we could enter as extra input to our network the tube specific coordinates.

```
class NavierStokes_CoordTransformed(PDE):


    name = "NavierStokes_CoordTransformed"

    def __init__(self, nu,case_coord_strList=None,
        case_param_strList=None, rho=1, dim=3, time=True, mixed_form
        =False):
        # set params
        self.dim = dim
        self.time = time
        self.mixed_form = mixed_form
        if case_param_strList is None:
            case_param_strList={}
        if case_coord_strList is None:
            case_coord_strList=["x_case","y_case","z_case"]
        if (case_coord_strList)==1:
            case_coord_strList=case_coord_strList+["y_case","z_case
            "]
        elif (case_coord_strList)==2:
            case_coord_strList=case_coord_strList+["z_case"]
        # coordinates
        t= Symbol("t")

        x = Symbol(case_coord_strList[0])
        y = Symbol(case_coord_strList[1])
        z = Symbol(case_coord_strList[2])
        input_variables = {case_coord_strList[0]: x,
            case_coord_strList[1]: y, case_coord_strList[2]: z, "t":
            t}
```

```
26          for key in case_param_strList:
27              input_variables[key]=case_param_strList[key]
28          if self.dim == 2:
29              input_variables.pop("z_case")
30          if not self.time:
31              input_variables.pop("t")
32
33          # velocity componets
34          u = Function("u")(*input_variables)
35          v = Function("v")(*input_variables)
36          if self.dim == 3:
37              w = Function("w")(*input_variables)
38          else:
39              w = Number(0)
40
41          # pressure
42          p = Function("p")(*input_variables)
43
44          # kinematic viscosity
45          if isinstance(nu, str):
46              nu = Function(nu)(*input_variables)
47          elif isinstance(nu, (float, int)):
48              nu = Number(nu)
49
50          # density
51          if isinstance(rho, str):
52              rho = Function(rho)(*input_variables)
53          elif isinstance(rho, (float, int)):
54              rho = Number(rho)
55
56          # dynamic viscosity
57          mu = rho * nu
58
59          # set equations
60          self.equations = {}
61          self.equations["continuity"] = (
62              rho.diff(t) + (rho * u).diff(x) + (rho * v).diff(y) + (
                    rho * w).diff(z)
63          )
64
65          if not self.mixed_form:
66              curl = Number(0) if rho.diff(x) == 0 else u.diff(x) + v
                    .diff(y) + w.diff(z)
67              self.equations["momentum_x"] = (
68                  (rho * u).diff(t)
69                  + (
70                      u * ((rho * u).diff(x))
71                      + v * ((rho * u).diff(y))
```

```
                    + w * ((rho * u).diff(z))
                    + rho * u * (curl)
                )
                + p.diff(x)
                - (-2 / 3 * mu * (curl)).diff(x)
                - (mu * u.diff(x)).diff(x)
                - (mu * u.diff(y)).diff(y)
                - (mu * u.diff(z)).diff(z)
                - (mu * (curl).diff(x))
            )
            self.equations["momentum_y"] = (
                (rho * v).diff(t)
                + (
                    u * ((rho * v).diff(x))
                    + v * ((rho * v).diff(y))
                    + w * ((rho * v).diff(z))
                    + rho * v * (curl)
                )
                + p.diff(y)
                - (-2 / 3 * mu * (curl)).diff(y)
                - (mu * v.diff(x)).diff(x)
                - (mu * v.diff(y)).diff(y)
                - (mu * v.diff(z)).diff(z)
                - (mu * (curl).diff(y))
            )
            self.equations["momentum_z"] = (
                (rho * w).diff(t)
                + (
                    u * ((rho * w).diff(x))
                    + v * ((rho * w).diff(y))
                    + w * ((rho * w).diff(z))
                    + rho * w * (curl)
                )
                + p.diff(z)
                - (-2 / 3 * mu * (curl)).diff(z)
                - (mu * w.diff(x)).diff(x)
                - (mu * w.diff(y)).diff(y)
                - (mu * w.diff(z)).diff(z)
                - (mu * (curl).diff(z))
            )

            if self.dim == 2:
                self.equations.pop("momentum_z")

        elif self.mixed_form:
            u_x = Function("u_x")(*input_variables)
            u_y = Function("u_y")(*input_variables)
            u_z = Function("u_z")(*input_variables)
```

```
120             v_x = Function("v_x")(*input_variables)
121             v_y = Function("v_y")(*input_variables)
122             v_z = Function("v_z")(*input_variables)
123 
124             if self.dim == 3:
125                 w_x = Function("w_x")(*input_variables)
126                 w_y = Function("w_y")(*input_variables)
127                 w_z = Function("w_z")(*input_variables)
128             else:
129                 w_x = Number(0)
130                 w_y = Number(0)
131                 w_z = Number(0)
132                 u_z = Number(0)
133                 v_z = Number(0)
134 
135             curl = Number(0) if rho.diff(x) == 0 else u_x + v_y +
                w_z
136             self.equations["momentum_x"] = (
137                 (rho * u).diff(t)
138                 + (
139                     u * ((rho * u.diff(x)))
140                     + v * ((rho * u.diff(y)))
141                     + w * ((rho * u.diff(z)))
142                     + rho * u * (curl)
143                 )
144                 + p.diff(x)
145                 - (-2 / 3 * mu * (curl)).diff(x)
146                 - (mu * u_x).diff(x)
147                 - (mu * u_y).diff(y)
148                 - (mu * u_z).diff(z)
149                 - (mu * (curl).diff(x))
150             )
151             self.equations["momentum_y"] = (
152                 (rho * v).diff(t)
153                 + (
154                     u * ((rho * v.diff(x)))
155                     + v * ((rho * v.diff(y)))
156                     + w * ((rho * v.diff(z)))
157                     + rho * v * (curl)
158                 )
159                 + p.diff(y)
160                 - (-2 / 3 * mu * (curl)).diff(y)
161                 - (mu * v_x).diff(x)
162                 - (mu * v_y).diff(y)
163                 - (mu * v_z).diff(z)
164                 - (mu * (curl).diff(y))
165             )
166             self.equations["momentum_z"] = (
```

```
167                 (rho * w).diff(t)
168                 + (
169                     u * ((rho * w.diff(x)))
170                     + v * ((rho * w.diff(y)))
171                     + w * ((rho * w.diff(z)))
172                     + rho * w * (curl)
173                 )
174                 + p.diff(z)
175                 - (-2 / 3 * mu * (curl)).diff(z)
176                 - (mu * w_x).diff(x)
177                 - (mu * w_y).diff(y)
178                 - (mu * w_z).diff(z)
179                 - (mu * (curl).diff(z))
180             )
181             self.equations["compatibility_u_x"] = u.diff(x) - u_x
182             self.equations["compatibility_u_y"] = u.diff(y) - u_y
183             self.equations["compatibility_u_z"] = u.diff(z) - u_z
184             self.equations["compatibility_v_x"] = v.diff(x) - v_x
185             self.equations["compatibility_v_y"] = v.diff(y) - v_y
186             self.equations["compatibility_v_z"] = v.diff(z) - v_z
187             self.equations["compatibility_w_x"] = w.diff(x) - w_x
188             self.equations["compatibility_w_y"] = w.diff(y) - w_y
189             self.equations["compatibility_w_z"] = w.diff(z) - w_z
190             self.equations["compatibility_u_xy"] = u_x.diff(y) -
                    u_y.diff(x)
191             self.equations["compatibility_u_xz"] = u_x.diff(z) -
                    u_z.diff(x)
192             self.equations["compatibility_u_yz"] = u_y.diff(z) -
                    u_z.diff(y)
193             self.equations["compatibility_v_xy"] = v_x.diff(y) -
                    v_y.diff(x)
194             self.equations["compatibility_v_xz"] = v_x.diff(z) -
                    v_z.diff(x)
195             self.equations["compatibility_v_yz"] = v_y.diff(z) -
                    v_z.diff(y)
196             self.equations["compatibility_w_xy"] = w_x.diff(y) -
                    w_y.diff(x)
197             self.equations["compatibility_w_xz"] = w_x.diff(z) -
                    w_z.diff(x)
198             self.equations["compatibility_w_yz"] = w_y.diff(z) -
                    w_z.diff(y)
199
200             if self.dim == 2:
201                 self.equations.pop("momentum_z")
202                 self.equations.pop("compatibility_u_z")
203                 self.equations.pop("compatibility_v_z")
204                 self.equations.pop("compatibility_w_x")
205                 self.equations.pop("compatibility_w_y")
```

```
206                     self.equations.pop("compatibility_w_z")
207                     self.equations.pop("compatibility_u_xz")
208                     self.equations.pop("compatibility_u_yz")
209                     self.equations.pop("compatibility_v_xz")
210                     self.equations.pop("compatibility_v_yz")
211                     self.equations.pop("compatibility_w_xy")
212                     self.equations.pop("compatibility_w_xz")
213                     self.equations.pop("compatibility_w_yz")
```

Listing A.3: The Navier Stokes Equations class

Navier Stokes class that allows the user to implement the conservation of mass and momentum into his loss function.

```
1  channel = Channel2D_centerfocused(
2          (channel_length_nd[0], channel_width_nd[0]),   #x1,y1
3          (channel_length_nd[1], channel_width_nd[1]),   #x2,y2
4          parameterization=pr,
5      )
6
7      inlet_2d = Line(
8          (channel_length_nd[0], channel_width_nd[0]),   #x1,y1
9          (channel_length_nd[0], channel_width_nd[1]),   #x1,y2
10         normal=-1,
11         parameterization=pr,
12     )
13
14     outlet_2d = Line(
15         (channel_length_nd[1], channel_width_nd[0]),   #x2,y1
16         (channel_length_nd[1], channel_width_nd[1]),   #x2,y2
17         normal=1,
18         parameterization=pr,
19     )
20
21     wall_btm = HLine(
22         (channel_length_nd[0], channel_width_nd[0]),   #x1, y1
23         (channel_length_nd[1], channel_width_nd[0]),   #x2, y1
24         normal=-1,
25         parameterization=pr,
26     )
27
28     wall_top = HLine(
29         (channel_length_nd[0], channel_width_nd[1]),   #x1,y2
30         (channel_length_nd[1], channel_width_nd[1]),   #x2,y2
31         normal=1,
32         parameterization=pr,)
```

Listing A.4: Creation of the domain in Nvidia Modulus Sym

To define the domain in Python we used the built-in features of Nvidia Modulus Sym.

```
Stenosis_coordTransform=CustomModuleArch(
        [Key("x"), Key("y"),Key("fc")],
        [Key("x_case"), Key("y_case")],
        module=Stenosis_2(channel_radius_nd,channel_length_nd[1]-
            channel_length_nd[0])
        )


    ge_net=CustomModuleArch(
        [Key("x_case"), Key("y_case"), Key("fc")],
        [Key("cline"), Key("radius")],
        module=GE(channel_radius_nd,channel_length_nd[1]-
            channel_length_nd[0])
        )


    warp=CustomModuleArch(
        [Key('x'),Key('y'),Key("x_case"), Key("y_case")],
        [Key("warpx"), Key("warpy")],
        module=warped()
        )

    ns = NavierStokes_CoordTransformed(nu=nu_nd, rho=rho_nd, dim=2,
        time=False)

    normal_dot_vel = NormalDotVec(["u", "v"])


    Dissq_net=CustomModuleArch(
        [Key("radius")],
        [Key("dissq")],
        module=Dissq()
        )

     incrorder_NN = CustomModuleArch(
        input_keys=[Key("cline"),Key("radius"),Key("dissq")],
        output_keys=[Key("cline_sq"),Key("radius_sq"),Key("
            cline_radius"),Key("cline_dissq"),Key("radius_dissq")],
        module=incrorder(),
    )
```

Listing A.5: Creation of list of nodes to unroll graph on in Nvidia Modulus Sym

By using the custom classes we created before, we created the nodes for our problem.

```
  flow_net = instantiate_arch(
        input_keys=[Key("x_case"), Key("y_case"),Key("cline"), Key(
            "radius"),Key("dissq"),Key("cline_sq"),Key("radius_sq"),
            Key("cline_radius"),Key("cline_dissq"),Key("radius_dissq
```

```
            "),Key("fc"),Key("inlet_u")],
3          output_keys=[Key("u"), Key("v"), Key("p")],
4          cfg=cfg.arch.fully_connected,
5          activation_fn=Activation.RELU,
6      )
```

Listing A.6: Creation of the network

For the creation of the network we had to define the input and output keys of our problem,the type of the neural network and the activation function used.

```
1    # inlet
2      inlet = PointwiseBoundaryConstraint(
3          nodes=nodes,
4          geometry=inlet_2d,
5          outvar={"u": inlet_parabola, "v": 0},
6          batch_size=cfg.batch_size.inlet*batchsizefactor,
7          parameterization=param_ranges,
8      )
9      domain.add_constraint(inlet, "inlet")
10
11
12     # outlet
13     outlet = PointwiseBoundaryConstraint(
14         nodes=nodes,
15         geometry=outlet_2d,
16         outvar={"p": outlet_p},
17         batch_size=cfg.batch_size.outlet*batchsizefactor,
18         parameterization=param_ranges,
19     )
20     domain.add_constraint(outlet, "outlet")
21
22
23     # no slip
24     no_slip_wall_btm = PointwiseBoundaryConstraint(
25         nodes=nodes,
26         geometry=wall_btm,
27         outvar={"u": noslip_u, "v": noslip_v},
28         batch_size=cfg.batch_size.walls*batchsizefactor,
29         parameterization=param_ranges,
30     )
31     domain.add_constraint(no_slip_wall_btm, "no_slip_wall")
32
33
34     no_slip_wall_top = PointwiseBoundaryConstraint(
35         nodes=nodes,
36         geometry=wall_top,
37         outvar={"u": noslip_u, "v": noslip_v},
38         batch_size=cfg.batch_size.walls*batchsizefactor,
39         parameterization=param_ranges,
```

```
    )
    domain.add_constraint(no_slip_wall_top, "no_slip_wall")


    # interior contraints
    interior = PointwiseInteriorConstraint(
        nodes=nodes,
        geometry=volume_geo,
        outvar={"continuity": 0, "momentum_x": 0, "momentum_y": 0},
        batch_size=cfg.batch_size.interior*batchsizefactor,
        bounds=Bounds({x: channel_length_nd, y: channel_width_nd}),
        parameterization=param_ranges,
    )
    domain.add_constraint(interior, "interior")
```

Listing A.7: Definition of constrains in Nvidia Modulus Sym

The constraints used to define the loss function were the 2D Navier Stokes equations at the computational domain as well as the boundary conditions at the inlet, outlet and the walls.
In order to be able to start the running process we have to additionally use a configuration file in which basic parameters of the training are defined.

```
 defaults :
  - modulus_default
  - arch:
      - fully_connected





  - optimizer : adagrad
  - scheduler : tf_exponential_lr
  - loss : sum
  - _self_




arch:
    fully_connected:
        layer_size: 256
        nr_layers: 4
jit: false


scheduler:
  decay_rate: 0.95
```

```
27    decay_steps: 2000
28
29  training:
30    rec_validation_freq: 1000
31    rec_inference_freq: 1000
32    rec_monitor_freq: 1000
33    rec_constraint_freq: 2000
34    max_steps: 2000000
35
36
37  batch_size:
38    inlet: 160
39    outlet: 160
40    walls: 160
41    no_slip: 320
42    interior: 3200
```


Listing A.8: A basic configuration YAML file

```
1  file_path = "/data/Specific/Tube3_stenosis.csv"
2  if os.path.exists(to_absolute_path(file_path)):
3      mapping = {"x": "x", "y": "y", "u": "u", "v": "v", "p": "p"}
4      stenosis_var = csv_to_dict(to_absolute_path(file_path), mapping
          )
5
6      # Update with constant control factors
7      stenosis_var.update({"fc": np.full_like(stenosis_var["x"],
          0.17)})
8      stenosis_var.update({"inlet_u": np.full_like(stenosis_var["x"],
           0.7)})
9
10     stenosis_invar_numpy = {
11         key: value for key, value in stenosis_var.items() if key in
               ["x", "y", "fc", "inlet_u"]
12     }
13
14     print("File found and processed.")
15
16     stenosis_inferencer = PointwiseInferencer(
17         nodes=nodes,
18         invar=stenosis_invar_numpy,
19         output_names=["u", "v", "p", "transformed_x", "
              transformed_y"]
20     )
21     domain.add_inferencer(stenosis_inferencer, "StenosisInference")
22 else:
23     print(f"File not found: {to_absolute_path(file_path)}")
```

Listing A.9: Inferencing in Nvidia Modulus

```python
class AntiStenosis(torch.nn.Module):
    def __init__(self, r, l):
        super().__init__()
        self.register_buffer("r", torch.tensor(r), persistent=False
            )
        self.register_buffer("l", torch.tensor(l), persistent=False
            )  # Length of Line

    def forward(self, x):
        x_ref = x[..., 0:1]
        y_case = x[..., 1:2]
        fc = x[..., 2:3]

        # Transform x_ref to range [-1, 1] for the cosine function
        x_ref_transformed = 2 * (x_ref - self.l / 8) / (3 * self.l
            / 8 - self.l / 8) - 1

        # Create masks for the piecewise function
        mask1 = (x_ref >= -self.l / 2) & (x_ref < -self.l / 8)
        mask2 = (x_ref >= -self.l / 8) & (x_ref < self.l / 8)

        # Compute y_ref based on the interval
        y_ref = torch.where(
            mask1,
            y_case,
            torch.where(
                mask2,
                y_case / (1 - fc * (1 + torch.cos(x_ref_transformed
                    * torch.pi))),
                y_case
            )
        )

        return torch.cat((x_ref, y_ref), -1)
```

Listing A.10: The Anti-Stenosis class used to transform Stenosis x,y from COMSOL to non Stenotic channel x,y

Bibliography

[1] Harrison Kinsley and Daniel Kukiela. *Neural Networks from Scratch in Python*. Sentdex, Kinsley Enterprises, Dallas-Fort Worth Metroplex, 2020.

[2] M. Raissi, P. Perdikaris and G. E. Karniadakis. *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*. *Journal of Computational Physics*, 378:686–707, 2019.

[3] Modulus Contributors. *NVIDIA Modulus: An open-source framework for physics-based deep learning in science and engineering*. 2023.

[4] Maziar Raissi, Alireza Yazdani and George Em Karniadakis. *Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations*. *Science*, 367(6481):1026–1030, 2020.

[5] Kyle A Williams, Allison Shields, Mohammad Mahdi Shiraz Bhurwani, SV Setlur Nagesh, Daniel R Bednarek, Stephen Rudin and Ciprian N Ionita. *Use of high-speed angiography HSA-derived boundary conditions and Physics Informed Neural Networks (PINNs) for comprehensive estimation of neurovascular hemodynamics*. *Medical Imaging 2023: Physics of Medical Imaging*, volume 12463, pages 194–204. SPIE, 2023.

[6] Jeremías Garay, Jocelyn Dunstan, Sergio Uribe and Francisco Sahli Costabal. *Physics-informed neural networks for parameter estimation in blood flow models*. *Computers in Biology and Medicine*, 178, 2024.

[7] Xuelan Zhang, Baoyan Mao, Yue Che, Jiaheng Kang, Mingyao Luo, Aike Qiao, Youjun Liu, Hitomi Anzai, Makoto Ohta, Yuting Guo and Gaoyang Li. *Physics-informed neural networks (PINNs) for 4D hemodynamics prediction: An investigation of optimal framework based on vascular morphology*. *Computers in Biology and Medicine*, 164, 2023.

[8] Maziar Raissi, Paris Perdikaris and George Em Karniadakis. *Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*. *arXiv preprint arXiv:1711.10566*, 2017.

[9] Ameya D Jagtap, Kenji Kawaguchi and George Em Karniadakis. *Adaptive activation functions accelerate convergence in deep and physics-informed neural networks*. *Journal of Computational Physics*, 404:109136, 2020.

[10] Michael Betancourt. *A geometric theory of higher-order automatic differentiation*. *arXiv preprint arXiv:1812.11592*, 2018.

[11] Jesse Bettencourt, Matthew J Johnson and David Duvenaud. *Taylor-mode automatic differentiation for higher-order derivatives in JAX. Program Transformations for ML Workshop at NeurIPS 2019*, 2019.

[12] Hong Shen Wong, Wei Xuan Chan, Bing Huan Li and Choon Hwai Yap. *Strategies for multi-case physics-informed neural networks for tube flows: a study using 2D flow scenarios. Scientific Reports*, 14:11577, 2024.

[13] Ashish Rajanand and Pradeep Singh. *ErfReLU: adaptive activation function for deep neural network. Pattern Analysis and Applications*, 27(2):68, 2024.

[14] J.L. Randall. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems. Choice Rev. Online*, 45, 2008.

[15] Isaac E Lagaris, Aristidis Likas and Dimitrios I Fotiadis. *Artificial neural networks for solving ordinary and partial differential equations. IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[16] Isaac E Lagaris, Aristidis C Likas and Dimitris G Papageorgiou. *Neural-network methods for boundary value problems with irregular boundaries. IEEE Transactions on Neural Networks*, 11(5):1041–1049, 2000.

[17] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang and George Em Karniadakis. *Learning Nonlinear Operators via DeepONet Based on the Universal Approximation Theorem of Operators. Nature Machine Intelligence*, 3(3):218–229, 2021.

[18] Lu Lu, Yeonjong Shin, Yanhui Su and George Em Karniadakis. *Dying ReLU and Initialization: Theory and Numerical Examples. Communications in Computational Physics*, 28(5):1671–1706, 2020.

[19] Lu Lu, Xuhui Meng, Zhiping Mao and George Em Karniadakis. *DeepXDE: A deep learning library for solving differential equations. SIAM Review*, 63(1):208–228, 2021.

[20] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang and Liu Yang. *Physics-informed machine learning. Nature Reviews Physics*, 3(6):422–440, 2021.

[21] Félix Fernándezde la Mata, Alfonso Gijón, Miguel Molina-Solana and Juan Gómez-Romero. *Physics-informed neural networks for data-driven simulation: Advantages, limitations, and opportunities. Physica A: Statistical Mechanics and its Applications*, 610, 2023.

[22] Atilim Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul and Jeffrey Mark Siskind. *Automatic Differentiation in Machine Learning: a Survey. Journal of Machine Learning Research*, 18:1–43, 2018.

[23] Enzo Grossi, Riccardo Marmo, Marco Intraligi and Massimo Buscema. *Artificial Neural Networks for Early Prediction of Mortality in Patients with Non Variceal Upper GI Bleeding (UGIB). Biomedical Informatics Insights*, 1:BII.S814, 2008. PMID: 27429551.

[24] Violetta Schäfer. *Generalization of PINNs for Various Boundary and Initial Conditions.* Master Thesis, University of Kaiserslautern, Faculty of Mathematics, Kaiserslautern, Germany, 2022.

[25] Oleg Rudenko, Oleksandr Bezsonov and Kyrylo Oliynyk. *First-Order Optimization (Training) Algorithms in Deep Learning. Proceedings of the International Conference on Computational Linguistics and Intelligent Systems (COLINS)*, Kharkiv, Ukraine, 2020. CEUR Workshop Proceedings.

[26] Naveen Kumar Subramanian. *Physics Informed Neural Networks in Fluid Dynamics.* Master's thesis, Technische Universität München, Munich, Germany, 2021.

[27] Xiaowei Jin, Shengze Cai, Hui Li and George Em Karniadakis. *NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. Journal of Computational Physics*, 426:109951, 2021.

[28] Yuekun Yang and Youssef Mesri. *Learning by neural networks under physical constraints for simulation in fluid mechanics. Computers and Fluids*, 248:105632, 2022.

[29] Philipp Moser, Wolfgang Fenz, Stefan Thumfart, Isabell Ganitzer and Michael Giretzlehner. *Modeling of 3D Blood Flows with Physics-Informed Neural Networks: Comparison of Network Architectures. Fluids*, 8(2):46, 2023.

[30] Revanth Mattey and Susanta Ghosh. *A physics informed neural network for time-dependent nonlinear and higher order partial differential equations. arXiv preprint arXiv:2106.07606*, 2021.

[31] Benjamin Wu, Oliver Hennigh, Jan Kautz, Sanjay Choudhry and Wonmin Byeon. *Physics informed RNN-DCT networks for time-dependent partial differential equations. International Conference on Computational Science*, pages 372–379. Springer, 2022.

[32] Samuel Burbulla. *Physics-informed neural networks for transformed geometries and manifolds. arXiv preprint arXiv:2311.15940*, 2023.

[33] Zongyi Li, Nikola Borislavov Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Prakash Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli and Anima Anandkumar. *Geometry-Informed Neural Operator for Large-Scale 3D PDEs. Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023).* NVIDIA, 2023.

[34] Khemraj Shukla, Vivek Oommen, Ahmad Peyvan, Michael Penwarden, Nicholas Plewacki, Luis Bravo, Anindya Ghoshal, Robert M. Kirby and George Em Karniadakis. *Deep neural operators as accurate surrogates for shape optimization. Engineering Applications of Artificial Intelligence*, 129:107615, 2024.

[35] Mateus Dias Ribeiro, Abdul Rehman, Sheraz Ahmed and Andreas Dengel. *DeepCFD: Efficient steady-state laminar flow approximation with deep convolutional neural networks. arXiv preprint arXiv:2004.08826*, 2020.

[36] Jan Oldenburg, Finja Borowski, Alper Öner, Klaus Peter Schmitz and Michael Stiehm. *Geometry aware physics informed neural network surrogate for solving Navier-Stokes equation (GAPINN). SpringerOpen*, 2022.

[37] Alvaro Abucide-Armas, Koldo Portal-Porras, Unai Fernandez-Gamiz, Ekaitz Zulueta and Adrian Teso-Fz-Betono. *Convolutional Neural Network Predictions for Unsteady Reynolds-Averaged Navier-Stokes-Based Numerical Simulations. Journal of Marine Science and Engineering*, 11:129, 2023.

[38] Koldo Portal-Porras, Unai Fernandez-Gamiz, Ainara Ugarte-Anero, Ekaitz Zulueta and Asier Zulueta. *Alternative Artificial Neural Network Structures for Turbulent Flow Velocity Field Prediction. Mathematics*, 9(16):1939, 2021.

[39] Amanda A Howard, Mauro Perego, George Em Karniadakis and Panos Stinis. *Multifidelity deep operator networks for data-driven and physics-informed problems. Journal of Computational Physics*, 493:112462, 2023.

[40] Han Gao, Luning Sun and Jian Xun Wang. *PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain. Journal of Computational Physics*, 428:110079, 2021.

[41] Shinjan Ghosh, Amit Chakraborty, Georgia Olympia Brikis and Biswadip Dey. *Rans-pinn based simulation surrogates for predicting turbulent flows. arXiv preprint arXiv:2306.06034*, 2023.

[42] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin and George Em Karniadakis. *Physics-informed neural networks (PINNs) for fluid mechanics: A review. Acta Mechanica Sinica*, 37(12):1727–1738, 2021.

[43] Prakhar Sharma, Llion Evans, Michelle Tindall and Perumal Nithiarasu. *Stiff-PDEs and Physics-Informed Neural Networks. Archives of Computational Methods in Engineering*, 30:2929–2958, 2023.

[44] Mitchell Daneker, Shengze Cai, Ying Qian, Eric Myzelev, Arsh Kumbhat, He Li and Lu Lu. *Transfer learning on physics-informed neural networks for tracking the hemodynamics in the evolving false lumen of dissected aorta. Nexus*, 1, 2024.