School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Physics-Informed Neural Networks for Predicting Turbulent Fluid Flow Around an Airfoil

Adham Ali Mukadam

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2022/23

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (16/05/2023)*
Link to online code repository	*URL*	*Sent to supervisor and assessor (16/05/2023*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)

Summary

The optimisation and design of the aerial vehicle components greatly depend on high order numerical simulations which are extremely computationally expensive for turbulent flows. Most turbulent flow simulations utilise turbulence modelling, a method that relies on simplifying assumptions and statistical correlations to provide detailed qualitative information on turbulent flow fields at a variable level of accuracy and a high computational cost.

In recent years there has been an abundance of research that explores using neural networks (NNs) to simulate turbulent flows more efficiently. By leveraging the power of data-driven learning and physics-based modelling a variation of NNs called Physics Informed Neural Networks (PINNs) have shown great potential in predicting physical phenomena, such as turbulence, faster and more accurately than standard techniques. This project investigated the use of PINNs to predict turbulent fluid flows around an airfoil (the cross-sectional profile of a wing or a blade) more efficiently than numerical methods. We explored both established techniques and novel approaches to designing a PINN for accurate fluid flow predictions around an airfoil with limited flow data. Through analysis of applying the latest PINNs techniques, we highlight the advantages and disadvantages of applying PINNs for airfoil turbulence modelling with limited flow data and provide insights into the future direction of PINNS for airfoil optimisation.

Acknowledgements

I would like to thank my supervisor Dr Toni Lassila for his advice and my family for all their support.

Table of Contents

Chapter 1
Introduction and Background Research

1.1 Introduction

In the realm of fluid dynamics, the study of turbulent assumes a crucial role in understanding and predicting the behaviour of fluids within aerodynamic systems, such as airplanes. Turbulence refers to the chaotic and unpredictable nature of fluid flow, characterized by complex and intricate patterns. Despite its omnipresence in both natural phenomena and industrial applications, the precise modelling of turbulent flows remains a formidable challenge and an area of ongoing research.

From an engineering standpoint, when designing a component of an aerodynamic system, it is desirable to efficiently simulate the flow over the component under specific conditions. Computational Fluid Dynamics (CFD) provides a solution to this problem by enabling engineers to numerically simulate and analyse fluid flows and their interactions with solid objects. Among the various CFD methods, Direct Numerical Simulations (DNSs) are known for their exceptional accuracy. DNS involves the application of fundamental fluid flow equations at an extremely fine-grained level. However, this approach requires significant computational resources. For instance, Vinuesa et al. (2015) reported that their DNS of flow around a wing profile necessitated more than thirty-five million core hours and seventy-five terabytes of data to complete. Given the computational demands, DNS is not a feasible option for optimizing airfoil design. As a result, engineers traditionally resort to turbulence modelling techniques, which employ simplifying assumptions and statistical correlations to reduce computational costs while simulating turbulent flows. Although these techniques can struggle to capture all the intricacies of real-world turbulent flows, they are computationally more efficient compared to DNS.

With the rapid growth of available data, ever-increasing computational resources, and recent advancements in machine learning, Deep Neural Networks (DNNs) have demonstrated promising outcomes in various scientific domains, including fluid dynamics, as discussed by Brunton et al. (2020). In recent years, several studies, such as those by Ling et al. (2016), Beck et al. (2019), and Zhu et al. (2021), have explored the application of DNNs as an alternative approach to turbulence modelling. DNNs offer the potential to learn the complex relationships between inputs (e.g., positions in a flow field) and outputs (e.g., pressure distribution and flow velocity) directly from extensive datasets. By training DNNs on large volumes of data, they become capable of making accurate predictions without relying explicitly on turbulence models.

However, training DNNs solely based on flow quantities does not fully exploit the available prior knowledge. Raissi et al. (2019) introduced Physics-Informed Neural Networks (PINNs), a framework that incorporates the underlying laws of physics governing a problem to act as a regularization agent, constraining the solution space to a manageable size. PINNs integrate the laws of physics into the neural network architecture, enabling them to swiftly converge to accurate solutions and generalize effectively, even when only a limited number of training examples are available (Raissi et al., 2019).

The research on predicting fluid flow around airfoils using PINNs has been scarce, with Eivazi et al. (2022) being one of the few accessible papers in this area. Eivazi et al. (2022) successfully devised a PINN design for predicting incompressible turbulent flows. Building upon the concepts and implementation outlined in their work, our study aims to demonstrate the capability of PINNs to predict fluid flow characteristics using only partial data for fluid flow quantities.

This project aims to explore both established implementation techniques and novel architectures for Physics-Informed Neural Networks (PINNs) in the context of fluid flow predictions with limited data availability. The primary objective is to conduct a comprehensive analysis and testing to assess the strengths and weaknesses of different PINN implementations for fluid flow predictions with limited data. The limited availability of data poses a significant challenge in fluid flow predictions, making it crucial to evaluate and compare different approaches to determine their suitability and performance. Additionally, this research project aims to identify the optimal PINN design that can deliver quick and accurate results. By considering factors such as computational efficiency and predictive accuracy, the study seeks to find the most effective configuration of the PINN architecture.

1.2 Turbulence Modelling

1.2.1 NSE Equations

The Navier Stokes Equations NSEs are a set a of partial differential equations (PDEs) that govern the motion of fluids and are derived from fundamental principles of conservation of mass and momentum. In the context of two-dimensional, incompressible flows, the NSEs can be expressed as follows:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

Where (x, y) denotes a point in space, t is the time; The density of the fluid ρ and the Reynolds number Re are constants. The Reynolds number represents the ratio of the inertial forces to the viscous forces in the fluid. The instantaneous pressure, denoted by p, at a specific point (x, y) and time t can be written as a function $p(x, y, t)$ Similarly, u and v representing the instantaneous velocity in the x and y directions respectively, can also be written as functions. These three quantities can be referred to as flow quantities as they describe the motion of the flow in space and time.

As discussed in the introduction we can use these equations to simulate fluid flows extremely precisely using direct numerical simulation (DNS). However, for turbulence cases applying these equations is extremely computationally expensive. As a result, alternative approaches such as turbulence models are required to efficiently predict turbulent flows.

1.2.2 RANS Equations

One of the most powerful tools for predicting turbulence flows is the Reynolds averaged Navier-Stokes (RANS) equations introduced by (Reynolds,1895). The fundamental principle behind RANS equations lies in Reynolds decomposition, wherein each flow quantity c is decomposed into a time averaged component $\bar{c}$ and a fluctuating component c'. The time-averaged component $\bar{c}$ a at a specific point (x, y) is determined by taking the limit as time approaches infinity and integrating the instantaneous values of c over the time interval:

$$\bar{c}(x, y) = \lim_{T \to \infty} \int_0^T c(x, y, t)$$

To calculate c' we simply subtract the actual value of c from the time average value:

$$c'(x, y, t) = c(x, y, t) - \bar{c}(x, y)$$

This decomposition allows us to gain insights into the flow properties without the need for calculating the flow quantities at every instant in time. For instance, in the case of an airfoil, the average lift can be evaluated by examining the difference between the time-averaged pressures on the upper and lower surfaces. Similarly, the time-averaged flow fields can be analysed to determine whether the flow will adhere to the wing or undergo separation.

To derive the RANS equations, we apply Reynolds decomposition to the Navier-Stokes Equations (NSEs). This involves decomposing each flow quantity into its time-averaged and fluctuating components, followed by time-averaging each equation. By applying this procedure to the NSEs presented earlier, the RANS equations take the following form for an incompressible flow in two dimensions:

$$\frac{\partial \bar{u}}{\partial x} + \frac{\partial \bar{v}}{\partial y} = 0$$

$$\bar{u}\frac{\partial \bar{u}}{\partial x} + \bar{v}\frac{\partial \bar{u}}{\partial y} + \frac{\partial \overline{u'u'}}{\partial x} + \frac{\partial \overline{u'v'}}{\partial y} = -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x} + \frac{1}{Re}\left(\frac{\partial^2 \bar{u}}{\partial x^2} + \frac{\partial^2 \bar{u}}{\partial y^2}\right)$$

$$\bar{u}\frac{\partial \bar{v}}{\partial x} + \bar{v}\frac{\partial \bar{v}}{\partial y} + \frac{\partial \overline{u'v'}}{\partial x} + \frac{\partial \overline{v'v'}}{\partial y} = \rho g - \frac{1}{\rho}\frac{\partial \bar{p}}{\partial y} + \frac{1}{Re}\left(\frac{\partial^2 \bar{v}}{\partial x^2} + \frac{\partial^2 \bar{v}}{\partial y^2}\right)$$

While the RANS equations provide valuable insights into flow properties, the presence of terms containing fluctuating components, referred to as Reynolds stresses, impedes the study of the time-averaged flow fields. This is in essence the Reynolds-averaged closure problem, which involves finding models that express the Reynolds stresses solely in terms of time-averaged flow quantities. Various models, such as the k-ε model (Menter, 1994) and the Spalart-Allmaras model (Spalart and Allmaras, 1992), have been developed to address this closure problem. However, these models rely on simplifying assumptions and exhibit varying levels of accuracy depending on specific flow conditions. Also, that despite their relative efficiency, these turbulence models remain computationally demanding, often requiring substantial computational resources.

1.3 Neural Networks

1.3.1 Neural Network Architecture

(McCulloch and Pitts1943) first introduced the idea of Neural Networks (NN) inspired by the biological structure of the brain. They are a class of machine learning models capable of predicting complex non-linear mappings, making them highly suitable for tackling the intricate dynamics of turbulent flows.

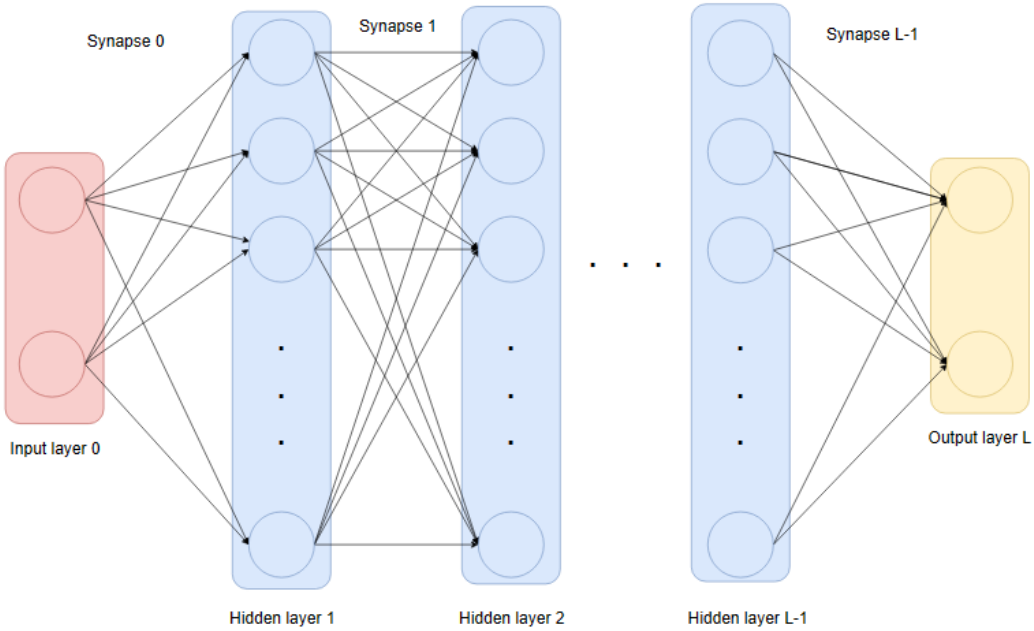


Figure 1.1 A Fully-connected Feed-forward Neural Network (FNN) architecture

A neural network, depicted in Figure 1.1, is composed of multiple layers of interconnected neurons linked by synapses. The network consists of an input layer, hidden layers, and an output layer. The input layer receives the input features, while the output layer provides the values of the output features. These layers do not alter the data but act as the entry and exit points of the network. Each neuron in the hidden layers is equipped with a bias variable and

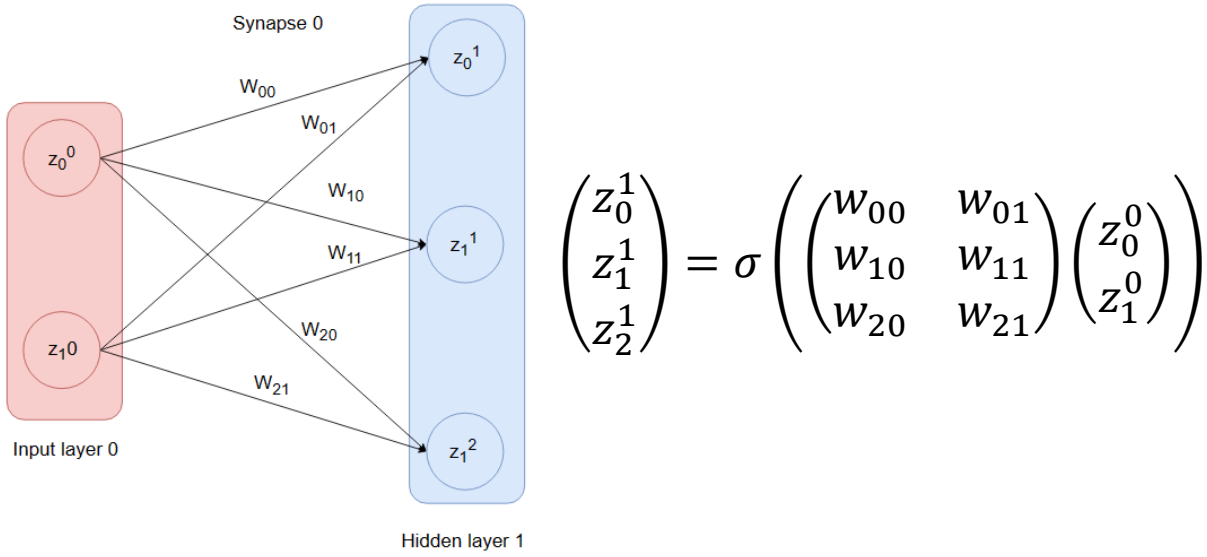
a non-linear activation function $\sigma(x)$. Additionally, each synapse possesses a weight variable. The neurons in the hidden layers receive input values from various synapses, which are summed along with the bias to compute a value we denote as x. This value is then inputted into the non-linear activation function $\sigma(x)$. The resulting output $\sigma(x)$ is transmitted to a synapse, which multiplies it by the weight associated with that synapse. The resulting value is then fed into any number of neurons in the subsequent layer through synapses. A FNN like in Figure 1.1 has a synapse from every neuron in a layer to every neuron in the next layer. When we pass a set of inputs through the neural network, which is known as a forward pass, we adopt an efficient computational approach. The output of the k^{th} layer of neurons is stored as a vector, denoted as z^k, and the biases are stored in a vector, denoted as b^k. The weights of each synapse are organized into an $n \times m$ dimensional matrix, represented as W^{k-1}, where n represents the number of neurons in layer k and m represents the number of neurons in layer $k-1$. This arrangement ensures that each row of the matrix W^{k-1} contains the weights associated with the synapses connected to the corresponding neuron in the k^{th} layer, as depicted in Figure 1. 2. Using this notation, the output of the neural network, denoted as z^L, can be expressed as:

$$z^0 = (z_0^0, z_1^0, \dots)^T$$

$$z^k = \sigma\left(W^{k-1}z^{k-1} + b^k\right) \qquad 1 \le k \le L-1$$

$$z^L = W^{L-1}z^{L-1} + b^L$$

Where z_i^0 is the input value of the i^{th} neuron in the input layer.



$$\begin{pmatrix} z_0^1 \\ z_1^1 \\ z_2^1 \end{pmatrix} = \sigma\left(\begin{pmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \end{pmatrix}\begin{pmatrix} z_0^0 \\ z_1^0 \end{pmatrix}\right)$$

Figure 1.2 Calculations of a forward pass through one synapse and one layer of neurons

The ability of neural networks to model complex non-linear functions stems from the combination of linear and non-linear functions applied to the input data. The weights and biases in the network are referred to as its parameters, and these parameters need to be

optimized to ensure effective prediction. This optimization process is known as backpropagation.

1.3.2 Loss Functions and Back Propagation

A NN is built to predict some observation (e.g., someone's height) given some information that affects the observation (e.g., their parents' heights) We call one input observation pair a data sample and a collection these is called a dataset. We use loss functions to determine how well the NN predicts the observation given the corresponding inputs. The loss function measures the error in the results which is called the residual. This error is iteratively calculated for many input observation pairs. For example, of the Sum of the Squared Residuals (SSR) and the Mean Squared Error (MSE) are given by:

$$SSR = \sum_{i=1}^{N} \left(O_i - \widehat{O_i}\right)^2$$

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \left(O_i - \widehat{O_i}\right)^2$$

Where O_i is an observation, we are trying to predict to predict, $\widehat{O_i}$ is the value predicted by the neural network and N is the total number of data sample in a dataset. The choice of an appropriate loss function depends on the specific requirements of the problem and the desired properties of the model's predictions.

The goal of training a NN is to minimise the loss function by changing the parameters and the algorithms that do this are referred to as optimisers. Most NN optimisers are variations on the gradient descend algorithm which takes small steps in the parameters' sizes in order to find the global minimum of the loss function. The step size is calculated by finding the partial derivative of the loss function L with respects to a parameter p and multiplying this by a learning rate coefficient lr. The step size is then subtracted from the value of the parameter to get a new value for the parameter. Updated parameters are calculated using the previous parameter value so after s steps the new parameter p_s can be calculated using the equation:

$$p_s = p_{s-1} - lr \frac{\partial L}{\partial p_{s-1}}$$

The process is repeated until the loss function is below a certain threshold value (we are close enough to the global minimum) or a predefined number of steps have been taken as shown in figure 1.3. Each repetition is called an epoch. The learning rate will determine how big the step size will be. Too big and the parameter might overshoot the global minimum and may fail to converge, too small and the parameter will fail to converge before the maximum number of steps is taken. Picking the correct learning rate is vital in making sure the parameters converge.

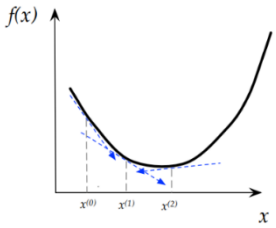


Figure 1.3 An example of gradient decent (Singer, 2016)

1.3.3 Optimisers

Batch gradient descent

There are many variations on the gradient decent algorithm and useful techniques to help a neural network converge on the best set of parameters. Batch gradient decent is the intuitive way of implementing gradient decent by calculating the loss over all data samples in the dataset before taking a step as shown in this equation:

$$p_s = p_{s-1} - lr \frac{\partial \sum_{i=1}^{N} \left(O_i - \widehat{O}_i\right)^2}{\partial p}$$

This method guarantees convergence however for very large dataset it is far too slow as to take one step it must iterate over the entire dataset.

Stochastic gradient descent

On the other hand, Stochastic gradient descent (SGD) updates the parameters after every data sample loss calculation. We take an input, feed it through the NN, calculate it's gradient and update the NN parameter so the equation for SGD is:

$$p_s = p_{s-1} - lr \frac{\partial \left(O_i - \widehat{O}_i\right)^2}{\partial p}$$

As we are only considering one point at a time when we update the parameters it is likely they can step in the wrong direct. This can cause the method loss function to fluctuate drastically as shown in Figure 1.4. However, the gradients are very fast to compute, and we update the parameter very frequently in small steps so the loss is reduced much quicker than in normal batch gradient descent. It is common to use this method with some form of learning rate decay to decrease the learning rate as the parameters approach the minimum.

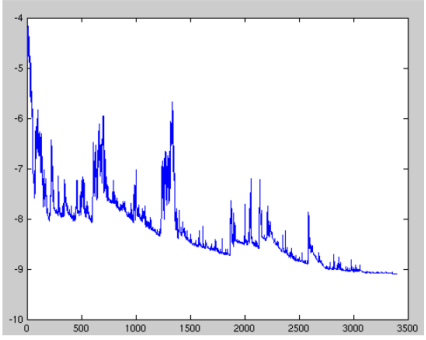


Figure 1.4 A graph of a loss function against the number of epochs for

Mini-batch gradient descent

Mini-batch gradient decent is a mixture of the first two methods. First the dataset is shuffled and then splits it into $M = \left\lceil \frac{N}{b} \right\rceil$ batches where b is the batch size. If N is not divisible by b the last batch is formed from the remaining data samples. The model calculates the loss over every datapoint in a batch before taking a step as shown in equation:

$$p_s = p_{s-1} - lr \frac{\partial \sum_{i=1}^{M} \left(O_i - \widehat{O_i}\right)^2}{\partial p}$$

This method is faster than the batch gradient descent as it needs to calculate the derivative of a much smaller term. The loss function is reduced quicker, requiring far less epochs to get in the range of the global minimum as it essentially takes more steps, and the loss does not fluctuate as much as in SDG.

AdaGrad optimiser

In traditional optimisation algorithms, the learning rate is fixed throughout the training process and is the same for all parameters however this may not be optimal for all parameters or at all stages of training. Varying the learning rate with each step for each parameter can help NNs to converge quicker. AdaGrad addresses this by adjusting the learning rate for each parameter based on its past gradient. AdaGrad stores a separate learning rate for each parameter and updates each one after every step. For a parameter p the learning rate after s steps have been taken can be given by the equation:

$$lr_s = \frac{lr_{s-1}}{\sqrt{\alpha_s + \epsilon}}$$

$$\alpha_s = \sum_{i=1}^{s} \left(\frac{\partial L}{\partial p_i}\right)^2$$

Where SSG coefficient α_s is the Sum of the Squared Gradients (SSGs) and ϵ is a very small number that ensures the detonator of lr_s will not equal 0. This means that parameters with larger gradients have their learning rate reduced, while parameters with smaller gradients have their learning rate increased. However, α_s can become too large for parameters with large gradients. This will cause the learning rate to tend towards 0 leading to the step size being very small.

RMSprop optimiser

To avoid this problem the RMSprop algorithm decays the learning rate denominator by using the concept of weighted average. RMSprop takes the weighted average of the previously calculated SSG coefficient α_{s-1} and the newly calculated SSGs:

$$\alpha_s = \beta \alpha_{s-1} + (1 - \beta) \sum_{i=1}^{s} \left(\frac{\partial L}{\partial p_i}\right)^2$$

We can then adjust the weight β depending on how much we want α_s to decay at each step. For example, a decay of $\beta = 0.9$ will give more weighting towards the previously calculated SSGs and less weighting towards the new value for the SSGs.

Adam optimiser

The Adam optimiser introduced by (Kingma and Ba 2014) combines these two techniques by having the same α_s term as RMSprop but also using the partial derivative of the loss function with respect to the parameter part of the equation. Now the instead of just the gradient for this term we look at the sum of the gradients at each step. For vanilla gradient decent we had:

$$p_s = p_{s-1} - lr\frac{\partial L}{\partial p_{s-1}}$$

Now with the Adam optimiser for each parameter, we can calculate the next value of the parameter using the equations:

$$p_s = p_{s-1} - lr_s \times \partial L_s$$

$$lr_s = \frac{lr_{s-1}}{\sqrt{\alpha_s + \epsilon}}$$

$$\alpha_s = \beta_1 \alpha_{s-1} + (1-\beta_1)\sum_{i=1}^{s}\left(\frac{\partial L}{\partial p_i}\right)^2$$

$$\partial L_s = \beta_2 \partial L_{s-1} + (1-\beta_2)\sum_{i=1}^{s}\frac{\partial L}{\partial p_i}$$

The Adam optimiser is “computationally efficient, has little memory requirements… and is well suited for problems that are large in terms of data and/or parameters” Kingma and Ba (2014). Along with the fact that empirical evidence that the Adam optimiser converges to a minimum in practice is why the algorithm is widely used especially for deep learning.

1.3.4 Non-linear Activation Functions

There are many types of non-linear activation functions each with there are own advantages and disadvantages. Some of the most popular activation functions include tanh, ReLu and leaky ReLu.

Tanh

The hyperbolic tangent or tanh activation function maps input values to a range $[-1,1]$ and is defined by the equation:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Tanh is a smooth function and differentiable across its entire range, enabling efficient backpropagation of gradients. It is also symmetrical allowing networks to capture both

positive and negative patterns in the data and its outputs are centred at zero which can help in creating a balance between positive and negative activations, making it easier for subsequent layers to learn and converge. The major drawback to Tanh is that it suffers from the vanishing gradient problem where, as the input value increase the gradient of the function tends towards 0 meaning it can be very slow to train and may require rescaling of observed data.

ReLu

The rectified linear unit or ReLu activation function maps input values to a range [0,∞] and is defined by the equation:

$$ReLu(x) = \begin{cases} 0 & if\ x \leq 0 \\ x & if\ x \geq 0 \end{cases}$$

The ReLu function is very computationally efficient, being a very simple to calculate function yet it allows for sparse activation by supressing negative values. This leads to reduced overfitting and efficient parameter updating as it can train different sets of parameters at each backwards propagation step. This makes it particularly beneficial when used with DNNs that can require many calculations for forward and back propagation and have to train large numbers of parameters. The function mostly avoids the vanishing gradient problem as it has a constant positive gradient, however it can suffer from a type of vanishing gradient problem called the dying ReLu problem where neurons can be become completely inactive for all inputs reducing the networks capacity and predictive power.

Leaky ReLu

Leaky ReLu activation function maps input values to a range [-∞,∞] and is defined by the equation:

$$LReLu(x) = \begin{cases} 0.01x & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

Leaky ReLu has most of the properties of the ReLu function however it is not as susceptible to the dying ReLu problem as the input as it does not completely kill negative values. Unfortunately, this comes at the cost of performance as it converges slower that the regular ReLu function.

1.3.5 Normalisation

When input features vary in scale, the optimisation process can be biased towards the features with larger magnitudes causing slower convergence allowing some input feature to have more influence over the direction of optimisation. This can be solved by normalising the input features so that each feature has a similar scale and falls within a certain range.

The same problems can occur with the output data where if the output features are on drastically different scale.

The most popular normalisation techniques are min-max scaling and Z-score normalisation. In min-max scaling the data is linearly transformed so that the minimum value becomes zero and maximum value become one. This is done by simply subtracting every value by the minimum value to every value and dividing by the maximum value. Min-max scaling preserves the original distribution of the data but may is very sensitive to outliers that may cause most of the data to be confined to a much smaller range than $[0,1]$. Z-score normalisation transforms the data, so it is centred around 0 and has an even distribution. This method is less sensitive to outliers.

1.4 Physics Informed Neural Networks

Physics-Informed Neural Networks (PINNs) have emerged as a powerful tool for solving complex scientific and engineering problems by combining the strengths of neural networks and the laws of physics. Raissi et al. (2019) observes that PINNs especially show their strength when working with small or noisy datasets as DNNs can lack robustness and fail to converge under these conditions.

1.4.1 Partial differential equations

In their paper reviewing the PINNs for fluid mechanics (Cai et al. 2021) explain that PINNs require partial differential equations (PDEs) that govern some law of physics given by:

$$f(\boldsymbol{x}, t, q, q_x, q_t, \dots, \boldsymbol{\lambda}) = 0, \qquad \boldsymbol{x} \in \Omega, t \in [0, T],$$

$$q(\boldsymbol{x}, t) = g_b(t), \qquad \boldsymbol{x} \in B\Omega, t \in [0, T]$$

$$q(\boldsymbol{x}, t_0) = g_0(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Omega, t \in [0, T]$$

Where the function f denotes the residual of a PDE measured by moving all the terms in a PDE to one side so they equal 0; $\boldsymbol{x} = (x_1, \dots x_d)$ is the spatial coordinate with d dimensions, $\Omega \subseteq \mathbb{R}^{\mathrm{d}}$ is the domain of $\boldsymbol{x}$, and t is the time; q is some physical quantity that varies in space and time of which there are usually more than one present in PDEs; $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots]$ are a set of unknown parameters we can optimise for. The subscript for flow quantities denotes the partial derivative of the flow quantity with respect to either a special or temporal dimension. g_b enforces an time independent condition, occurring at the boundary of an object, defined by the boundary domain $B\Omega \subset \Omega$. An example of this would be the no slip condition in fluid mechanics that assumes the velocity of the fluid is 0 where the fluid meets some solid boundary. g_0 enforces an initial condition at time $t = 0$ i.e., the velocity at every point in the domain need to be zero at the start of the simulation. Now if we want to accurately model some physical quantities that appear in a PDE we can use a DNN to predict these quantities. The we input them our into f and minimise the residual through back propagation.

1.4.2 Loss function and dataset

The loss function for a PINN now has to take into account not only the loss in the data but also the loss in the PDEs and in the initial and boundary conditions:

$$\mathcal{L} = \omega_1 \mathcal{L}_{data} + \omega_2 \mathcal{L}_{PDE} + \omega_3 \mathcal{L}_{IC} + \omega_4 \mathcal{L}_{BC}$$

Where ω_{1-4} are the weighting coefficients for the different loss terms and can be treated as hyperparameter that can be fine-tuned. The data loss term $\mathcal{L}_{data}$ is the same as the loss function described in section 1.3.2. The equation loss term $\mathcal{L}_{PDE}$ is the sum of the residuals of the PDEs. Therefore, $\mathcal{L}_{PDE}$ determines how well the PINN is predicting physical quantities with respects to the governing PDEs. The initial condition loss term $\mathcal{L}_{IC}$ and boundary condition loss term $\mathcal{L}_{BC}$ are imposed to satisfy the initial and boundary conditions of the system.

1.4.3 PINN architecture

The general architecture of a PINN shown in Figure 1.5 is a NN which takes the inputs features $\boldsymbol{x} \in \Omega$ and $t \in [0, T]$ and outputs the predictions for some physical quantities u, v, p and $\boldsymbol{\phi}$. The partial derivatives of u with respects to the input values $\boldsymbol{x}$ and t must be calculated to use in the equation loss. The data, equation, initial condition and boundary condition losses are then calculated, multiplied by their respective weights and added together according to the loss function $\mathcal{L}$. If the loss is below some threshold, then the NN can stop being trained. Otherwise, we can perform back propagation to update parameters according to the optimisation algorithm used and the process repeats with different inputs.

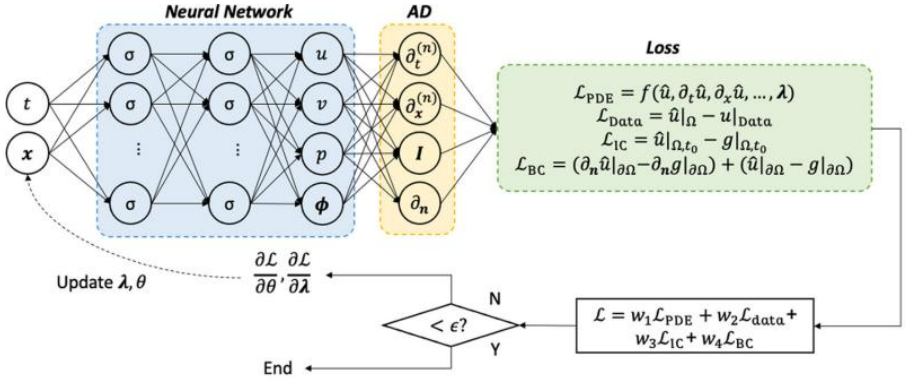


Figure 1.5 An example of a PINN architecture taken from Cai et al. (2021)

Some PDEs will requires taking the double derivate of a physical quantity that passes through the nonlinear activation functions at each neuron. For this reason, the tanh activation function is often used in PINNs as it is infinitely differentiable (He et al., 2020) and the ReLu function is often avoided as its double derivative is zero.

1.5 PINNs for Airfoil Design and Optimisation

Airfoil optimization refers to the process of refining the shape of an airfoil to achieve desirable aerodynamic characteristics, such as efficient lift generation, low drag, and stable flow behaviour. Airfoil optimization has significant applications in aircraft design, wind turbine blades, propellers, and other aerodynamic systems. The advancement of computational fluid dynamics (CFD) tools, along with optimization algorithms, has enabled more efficient numerical simulations that can take tens of core hours to compute. This in turn allows for the use of automated airfoil design processes, leading to improved performance of various aerospace and turbomachinery systems. In this project we will demonstrate that the process of simulation fluid flows can be even more efficient by using PINNs.

We will design a PINN that uses the RANS equations to predict the Reynolds averaged pressure and velocity flows around a two-dimensional airfoil with. As mentioned earlier Eivazi et al. (2022) have used a similar layout to demonstrate the ability of PINNs to accurately predict the turbulent incompressible flow around an airfoil, using the PINN design in Figure 1.6.

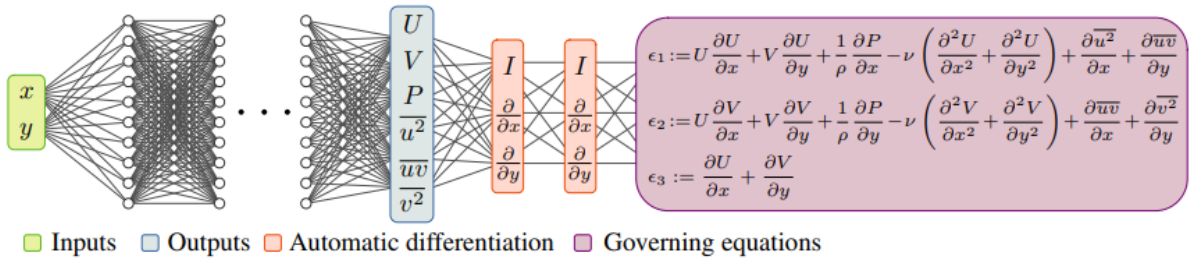


Figure 1.6 A PINN used by Eivazi et al. (2022) to predict turbulent flows

Their implementation however required the Reynold stress flow quantities for every point in the training database. Our project will build on the ideas from this paper and expand them to applications of PINNs where several quantities in the governing PDEs are unknown. This will mean predicting the Reynolds stresses with no data for them and therefore relying more on the equation loss. To account for this lack of Reynold stress data we will implement two novel PINN architectures that we hope will achieve better result that a regular FNN when we do not have data for some physical quantities. Additionally, we will evaluate all the hyperparameters of the PINN thoroughly to find a model that can predict fluid flows efficiently and as accuracy as possible. Once we have found the optimal configuration of hyperparameters and architecture we can display the predicted flow field visually to see whether PINNs with limited flow quantity data can visually simulate important flow features such as whether the flow will separate from the wing.

Chapter 2
Methods

2.1 Overview

As mentions in section 1.5 this project we will design a PINN to predict the Reynolds averaged pressure, velocity in the x direction, velocity in the y direction and Reynold stresses around an airfoil. We only have observed data for the mean flow quantities so cannot incorporate our predictions for the Reynolds stresses into the data loss term. We will use data from CFD simulations to train our PINN so firstly we will compile a training dataset that specifies the spatial coordinates (x, y) around the airfoil along with the flow quantities (p, u and v) and Reynold stresses ($\overline{uu}, \overline{uv}, \overline{vv}$) at that point. The dataset can be denoted by $\left\{x^i, y^i, \bar{p}^i, \bar{u}^i, \bar{v}^i, \overline{uu}^i, \overline{uv}^i, \overline{vv}^i\right\}_{i=1}^{N}$ where N is the number of sets in the dataset.The PINN will take x^i, y^i as input return the predictions for the time averaged quantities $\hat{O}^i = \left(\hat{p}^i, \hat{u}^i, \hat{v}^i, \widehat{uu}^i, \widehat{uv}^i, \widehat{vv}^i\right)$ as outputs. We can define the predicted values for just the quantities we can use in the data loss term as vector $\hat{D}^i = \left(\hat{p}^i, \hat{u}^i, \hat{v}^i\right)$. Similarly, we define the observed values that we will use in the data loss term as $D^i = \left(\bar{p},^i \bar{u}^i, \bar{v}^i\right)$.

We will test different variations on the network architecture and hyperparameters such as activation function, batching method of the optimiser, loss term weights and learning rate to find the optimum setup for the PINN. We will then train the PINN with optimal hyperparameters on 2 different CFD simulations and visualise the flow for each case to test the PINNs ability to predict flow quantities and important flow properties for different airfoil designs.

2.2 Libraries

2.2.1 VTK

VTK is an open-source software system for image processing, 3D graphics, volume rendering and visualization which has many advanced functionalities however in this project we only use it to extract data from flow field and airfoil geometry data store in .vtk files. We use the VTK python:

Module `vtkUnstructuredGridReader` to read spatial coordinates and flow field quantities from .vtk files as the flow field was stored as a `vtkUnstructuredGrid` object.

Module `vtkUnstructuredGridWriter` to write data to .vtk files so we can visualise the predicted values in ParaView.

Module `vtkPolyDataReader` to read in spatial coordinates of the boundary points of the airfoil as they were stored as a `vtkPolyData` object.

Method `vtk.util.numpy_support.vtk_to_numpy` to convert `vtkDataArray` objects to NumPy arrays.

2.2.2 Pandas and NumPy

Pandas is a fast and flexible open-source Python library that is useful for data analysis and manipulation. We used it to store flow field and boundary point data as Pandas `DataFrame` objects as they are easier to manipulate. Pandas `DataFrame` objects consist of columns, each of which are named, and rows which each have an ordered index. This allows for a highly flexible logic-based manipulating of `DataFrame` objects. NumPy is a Python library that specialises in dealing with any dimension of arrays. It also has a lot of utility with other libraries array structures such as being able to convert `vtkDataArray` objects to NumPy arrays and converting between panda `DataFrame`s and PyTorch tensors.

2.2.3 PyTorch

PyTorch is an open-source machine learning framework that provides a flexible and efficient platform for building and training deep learning models. It is widely well established in the field of deep learning with (Zeman et al. 2018) at Stanford employing it as a base to build a natural language analysis package.

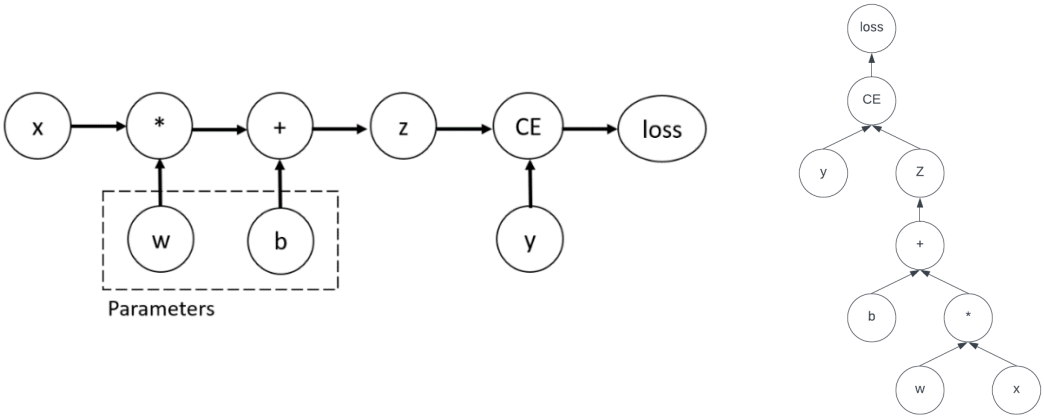


Figure 2.1 a computational graph stored in the loss term of a neural network

We use the `torch.tensor` object to store the inputs, weights and biases in tensors. This is due to the fact that if the attribute of a tensor `requires_grad` is set to `true`, PyTorch's `torch.autograd` engine records all the operations performed on it. Also, for tensors created from other tensors (e.g. the tensor formed from multiplying 2 tensors together) the tensors used to create this new tensor are stored also stored. It stores these operations and tensors as a graph, as shown in Figure 2.1, with the tensors used to create it being stored as leaves of the graph and the tensor itself being the root.

Each `torch.tensor` object also has a `grad` attribute which starts at zero when the tensor is initialled and gets added to. The `torch.tensor.backward(tensor)` method computes all the gradients of a tensor with respects to its graph leaves and accumulates it in

the `grad` attribute of the leaf. This method also releases the graphs of the root node and all the leaf nodes. The `torch.autograd.grad(outputs, inputs, create_graph)` method computes the gradient of the output tensor with respects to the input tensor without changing the attribute of the output tensors graph leaves. This is useful for PINNs where we may want to calculate the partial derivative of certain tensor with respects to neural network inputs. If the `create_graph` parameter is set to `True`, the graph of the derivative will be constructed and replace the old output tensor graph. This allows us to compute higher order derivative which is useful for PDEs with higher order derivative like the RANS equations.

A `torch.optim.Adam(parameters, learning rate)` object takes an inerrable or dictionary of tensors as the parameters of a neural network. Then when the `torch.optim.step()` method is called, the Adam optimiser will use the accumulated gradient stored in the `grad` attribute of its parameters to take a step according to the Adam algorithm. `optimizer.zero_grad()` must then be called to set the `grad` attribute of all the parameters to zero.

The `torch.nn.Module` class is the base class for all neural networks and basically only defines an empty `forward(input)` method that subclasses should define. The `torch.nn.Sequential(OrderedDict[str, Module])` class is a subclass of `torch.nn.Module and is` initialised with an `OrderedDict` of `torch.nn.Module` objects. It's `forward(input)` method takes the input and calls the forward method of the first module in the `OrderedDict`, takes this output and inputs it into the forward method of the second module and so on. By doing this it can chain together layers of neurons and synapses. We use a `torch.nn.Sequential` object to chain together `torch.nn.Linear` modules which act as the synapse layers and `torch.nn.Tanh` which act as the neuron layers. `torch.nn.Tanh` is the Tanh activation function. The `torch.nn.Linear` module then stores weights and performs the matrix multiplication according to the process defined in section 1.3.1.

We use the `torch.utils.data.Dataset` to define a dataset as it is necessary to pass as a parameter to a `torch.utils.data.DataLoader(dataset, batch_size)` object. This object will create a batched dataset that we can iterate over. If the `batch_size` is set to one, the optimization algorithm becomes SGD and if `batch_size` is equal to the size of the dataset, the optimization algorithm becomes batch gradient descent. Anywhere in-between becomes mini-batch gradient decent.

2.2.4 Matplotlib and seaborn

Matplotlib is a comprehensive library for plotting graphs in Python. The tool can display figures, tables, and graphs while providing a range of features to style each output. We employ this library to graph how evaluation metric changes with the number of epochs passes during training of the PINN.

2.3 Data Extraction, Reduction and Scaling

2.3.1 Airfoil Data

(Schillaci et al., 2021) compiled a dataset of 2D RANS simulations of the turbulent flow around NACA 4-digits airfoils that we will utilise. It contains a set of numerical simulations of turbulent flow around a family of airfoils using the simplest and computationally affordable computational fluid dynamics (CFD) approach. That is solving the RANS equations using the Spalart-Allmaras turbulence model. We briefly mentioned this model in section 1.2.2 and how it can be inaccurate, however this model was created specifically to model aircraft aerodynamics. Also focus of this project is to look at the increase in computational efficiency that a PINN can achieve. Numerical simulations like these often take hours to complete but we hope to achieve promising results in a fraction of that time. We will use three of these airfoil CFD simulations (0005, 0006 and 8646) to test whether our model can be trained on different airfoils.

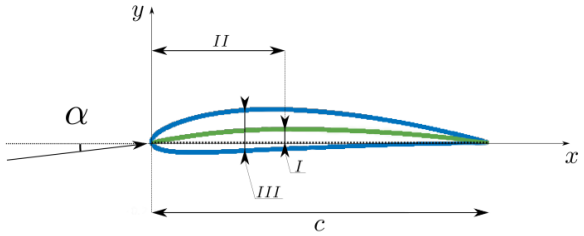


Figure 2.2 Sketch of an airfoil taken from Schillaci et al. (2021)

In the Schillaci et al. (2021) dataset for each simulation, the x and y coordinates measured in terms of chord length (length from leading edge to trailing edge) are centred around the tip of the leading edge of the airfoils. So, the chord length is always 1, the tip of the leading edge is at $x = 0$ and the tip of the trailing edge is at $x = 1$. Figure 2.2 shows an example of an airfoil with the important features labelled. c symbolises the chord length, α is the angle of attack and III measures the maximum thickness of the airfoil measured in units of $c/100$.

2.3.2 Data Preparation and Scaling

The data for each airfoil CFD simulation is stored in data in two vtk files. One file defines the geometry and the airfoil by storing the boundary points of the airfoil as a `vtkPolyData` object. The other file stores the points and field flow quantities as a `vtkUnstructuredGrid` object. We can extract the boundary points and field flow data using `vtkPolyDataReader` and `vtkUnstructuredGridReader` respectively and put into NumPy arrays using the `vtk.util.numpy_support.vtk_to_numpy` method as shown in Figures C.2 and C.3 in appendix C.

Each airfoil CFD simulations has a radius larger than 500 and consists of 999000 datapoints each with a spatial coordinate (x, y, z) and flow quantity data $(\bar{p}, \bar{u}, \bar{v}, \bar{w})$. The datapoints have a third dimension because the simulations were created in a 3D CFD simulation by making

the thickness of the wing spanwise very thin. This means the variation of the flow in the spanwise direction is negligible so we can drop the third dimension. We can store the point data in a Pandas `DataFrame` object and then drop the z and w columns. We will be felt with the columns x, y, p, u and v and 499,500 rows. We manipulate the `DataFrame` object to reduce the problem domain, as shown in Figure C.1, from a circle with radius 500 to a square of length 0.5 with a top right corner is at $(0,0)$ as shown in Figure 2.3.

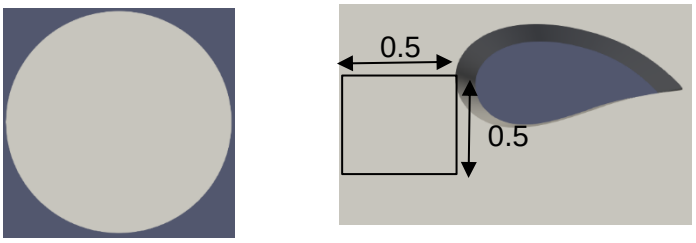


Figure 2.3 The domain before and after being reduced. In first image the airfoil is so small compared to the size of the domain that it cannot be seen

The reduced domain now has 7,119 datapoints points. We randomly sample 7,000 points from this reduced domain. The first 6,000 will form the training dataset and the last 1000 will form the testing set as shown in Figure C.4.

The output labels for Reynolds averaged pressure and velocity will be on drastically different scales with pressure being in the range of $[49, 445]$, velocity in x direction being in the range of $[0.51, 28.2]$, and velocity in y direction being in the range of $[-5.5, 9]$. Therefore, as mentioned in section 1.3.5 it is important that we scale both the input and output data. We will use min-max scaling as we do not want to affect the distribution of data. The implementation for this can be found in Figure C.15.

2.4 PINN model and Implementation

2.4.1 PDEs, Boundary Conditions and the Loss function

For the PDEs of the PINN we will use the two-dimensional RANS equations discussed in section 1.2.2. The equations residuals can be given by:

$$f_1 := \frac{\partial \bar{u}}{\partial x} + \frac{\partial \bar{v}}{\partial y} = 0$$

$$f_2 := \bar{u}\frac{\partial \bar{u}}{\partial x} + \bar{v}\frac{\partial \bar{u}}{\partial y} + \frac{\partial \overline{u'u'}}{\partial x} + \frac{\partial \overline{u'v'}}{\partial y} + \frac{1}{\rho}\frac{\partial \bar{p}}{\partial x} - \frac{1}{Re}\left(\frac{\partial^2 \bar{u}}{\partial x^2} + \frac{\partial^2 \bar{u}}{\partial y^2}\right) = 0$$

$$f_3 := \bar{u}\frac{\partial \bar{v}}{\partial x} + \bar{v}\frac{\partial \bar{v}}{\partial y} + \frac{\partial \overline{u'v'}}{\partial x} + \frac{\partial \overline{v'v'}}{\partial y} - \rho g + \frac{1}{\rho}\frac{\partial \bar{p}}{\partial y} - \frac{1}{Re}\left(\frac{\partial^2 \bar{v}}{\partial x^2} + \frac{\partial^2 \bar{v}}{\partial y^2}\right) = 0$$

Where Re is fixed at 3,000,000 as specified by the paper and ρ is fixed at 1.2.The airfoil will be subject to the no slip boundary condition which is given by $\bar{u} + \bar{v} = 0$.

As we are predicting Reynolds averages quantities our model is independent of time therefore the inputs spatial coordinates but no time input and there will be no initial condition.

We will use these equations, boundary condition and the training dataset to create the loss function. The Loss function will consist of 3 terms: the data, PDE and boundary condition loss terms. Each of the loss terms will take the MSE for a particular amount of datapoints is dependent on the batch size:

$$\mathcal{L}_{loss} = \mathcal{L}_{data} + \mathcal{L}_{PDE} + \mathcal{L}_{BC}$$

$$\mathcal{L}_{data} = \frac{1}{N_M}\sum_{i=1}^{N_M}\left|D^i - \widehat{D}^i\right|^2$$

$$\mathcal{L}_{PDE} = \frac{1}{N_M}\sum_{e}^{3}\sum_{i}^{N_M} f_e\left(x^i, y^i\right)^2$$

$$\mathcal{L}_{BC} = \frac{1}{N_B}\sum_{i}^{N_B} \bar{u}^2 + \bar{v}^2$$

Where N_M is the batch size, D and $\widehat{D}$ are defined in section 2.1 and N_B is the number boundary points we will use for each batch. There are approximately 1000 boundary points stored for each airfoil so for small batch size it is too computationally expensive to use all of them for every batch. Therefore, we will randomly sample $\min\left(\frac{N_M}{10}, 1000\right)$ Boundary points per batch to apply the boundary condition to.

2.4.2 PINN Architecture

All neural network architectures in this section are implemented in PyTorch using methods defined in section 2.2.3. We will implement 3 different neural network architectures. All of the architectures consist of an input layer with red neurons that takes the spatial coordinates (x, y), an output layer with yellow neurons that outputs the predicted values of the flow quantities $\hat{O}$, the partial differentiation layers with green neurons. The $\hat{O}$ represents the output layer being passed to the loss function unchanged and the same with the first order derivatives $\partial\hat{O}$. Every PINN design we try will have 8 hidden layers of 20 neurons but the connections between these neurons will change depending on the architecture.

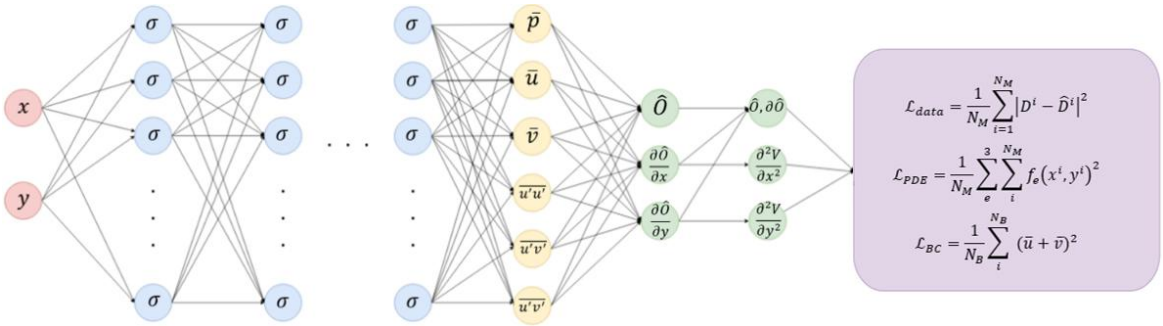


Figure 2.4 Fully connect FNN structure

The first architecture shown in Figure 2.4 architecture we will look at is the basic FNN architecture taken from Eivazi et al. (2022) as shown in Figure 1.6. The only different will be the different loss function displayed in the purple box.

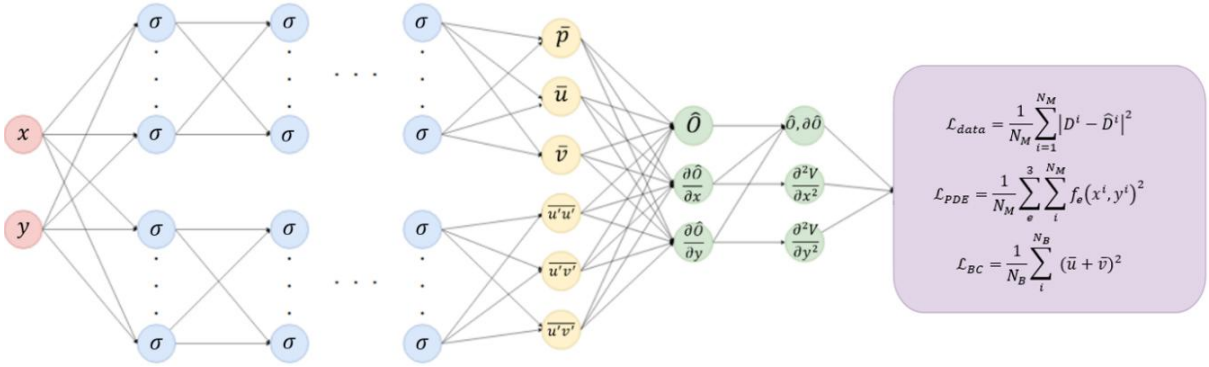


Figure 2.5 SPINN architecture where flow quantities and Reynold stresses predictions are calculated separately

We introduce a Split Physics Informed Neural Network (SPINN) architecture shown in Figure 2.5 where we train two neural networks and concatenate the outputs so they can be used to calculate the equation loss. When a SPINN applies back propagation the parameters of the top NN will be used to predict D and will be affect by all three loss terms. On the other hand, the parameters of the top NN will be used to predict the Reynolds stresses and will only be affected by the equation and boundary condition loss. By separating the networks, we hope that the data loss will converge quicker as the entire predictive power of the top layer is now being used solely for predicting D while still being constrained by the RANS equations.

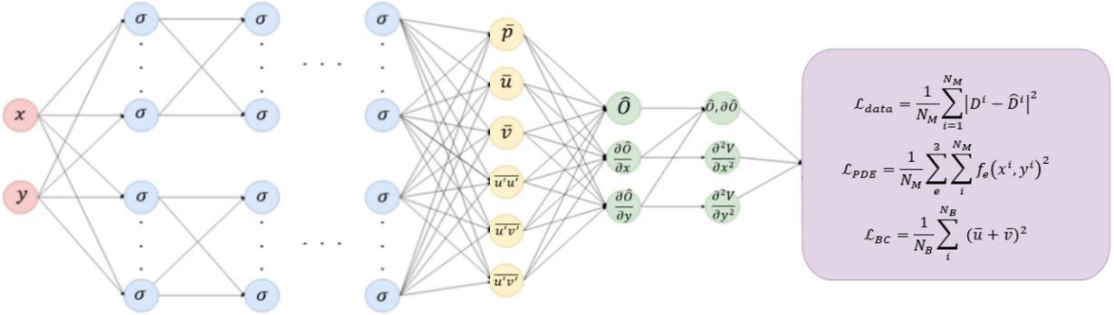


Figure 2.6 MSPINN architecture which is similar to a SPINN, but the last hidden layer forms a fully connected synapse or mixing synapse to the output layer

The last architecture shown in Figure 2.6 is a variation on the SPINN where the top and bottom neural networks are combined through a mixing synapse before providing outputs. We call a Mixed Split Physics Informed Neural Network (MSPINN). We hope this architecture will have a similar affect to the SPINN architecture but with more connectivity between the $\widehat{D}$ and Reynold stress predictions.

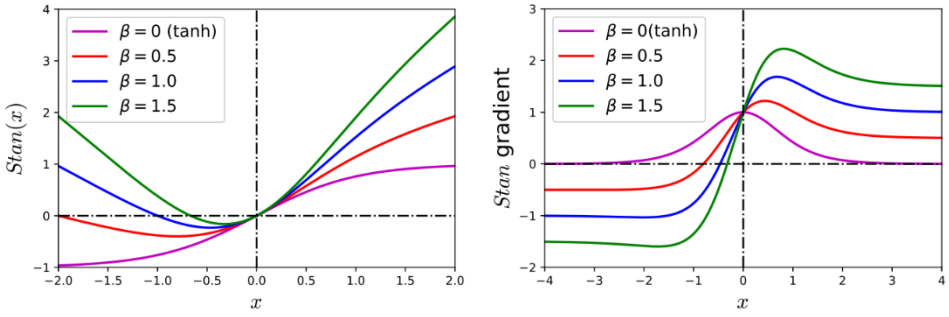
2.4.3 Stan Activation function

Most PINNs use the tanh activation function due to it being infinitely differentiable He et al. (2020) and its smooth gradient over the entire support (LeCun et al., 2012). As established in section 1.3.4.1 tanh suffers from the vanishing gradient problem, however (Gnanasambandam et al. 2022) explains that the bounded nature of the output of tanh also causes problems. The output from tanh is in the range [-1,1] so if we use tanh for the activation function for a network the last layer needs to be a linear layer of synapses. If we are predicting values well out of the range [-1,1] this will lead the final layers weights to be higher in magnitude than the rest of the layers as it will scale the outputs of the penultimate neuron layer from the range [-1,1] to a much larger scale. “It is preferred that the model gradually scales into different orders of magnitude than just using the last layer for scaling up. Both of the mentioned issues are more pronounced when the outputs are of higher orders of magnitude.” Gnanasambandam et al. (2022).

To address these issues Gnanasambandam et al. (2022) creates a new Self-scalable Tanh activation function that adds a self-scaling term to the tanh function.

$$Stan(x) = \tanh(x) + \beta_k^i \tanh(x)$$

Where β_k^i is defined separately for each neuron and i specifies which the neuron in the k^{th} layer the β_k^i is for. “This self-scaling term can help to better map different orders of magnitude of input and output allowing the gradients to flow without vanishing” Gnanasambandam et al. (2022)



(a) Stan activation function (b) Stan activation function gradient

Figure 2.7 Graphs of the Stan function compared to the tanh function

As shown in Figure 2.7 the Stan function can widen the range of outputs of a neuron by changing the β_k^i term allowing it to predict values in any range. Each β_k^i in a network is an additional parameter to optimise using gradient decent. Gnanasambandam et al. (2022) goes on to show that Stan with a constant learning rate outperforms tanh especially when outputs are in higher orders of magnitude. Stan also tends to converge towards a global minimum and is very stable when trained with different initializations of β_k^i.

Chapter 3
Results

3.1 Architecture and Hyperparameter configuration Testing

The hyperparameters for the FNN, SPINN and MSPINN architectures are the activation function, loss term weights, batch size, learning rate and number of epochs. The values we will test for each hyperparameter are given in Table 3.1.

Hyperparameter	Search Space
Activation Function	{tanh, Stan}
Loss term weights	{[1,1,1], [1,2,1], [2,1,1]}
Batch Size	{10, 100, 1000, 3000, 6000}
Learning Rate	{0.01, 0.001, 0.0001, 0.00001}
Epochs	{20, 50, 100, 200, 500}

Table 3.1 All the potential values of each hyperparameter

To tune the hyperparameters, we will train each architecture on many different promising hyperparameter configurations. To evaluate a hyperparameter configuration we calculate the data, equation, boundary loss and error in $\bar{p}$, $\bar{u}$ and $\bar{v}$ for the training set after every epoch so we can graph them. This will give us insight into how each hyperparameter affect the training process and will show us which configurations converge the quickest and smoothest. We also record the time taken to train the model and compare it to other models to find the most computationally efficient configuration. Once the model has finished training, we can evaluate the losses and errors on the testing set to see how well the model generalises to data within the reduced domain. The error for $\bar{p}$ can be given by:

$$E = \frac{1}{N}\sum_{i}^{N} \mathrm{abs}\left(\frac{\bar{p}^i - \hat{p}^i}{\bar{p}^i}\right) \times 100$$

Where the $\mathrm{abs}(x)$ function will take the absolute value of x. This equation is the same for $\bar{u}$ and $\bar{v}$.

3.2.1 FNN Results

Figure 3.1 Evaluation data throughout the training process for the best FNN hyperparameter configuration. Took 352 seconds to run 500 epochs.

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.006576	0.0008778	0.001163	12.69	12.81	10.07
Testing Set	0.006155	0.0008804	0.001164	13.02	11.59	9.81

Table 3.2 Evaluation metric for the best FNN hyperparameter configurations

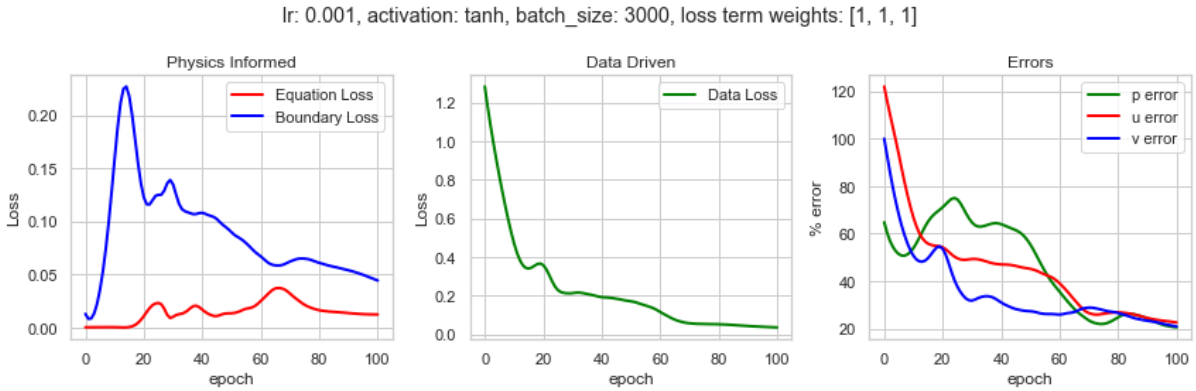


Figure 3.2 Evaluation data throughout the training process for the second-best FNN hyperparameter configuration. Took 454 second to run 100 epochs

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.006298	0.001449	0.001084	13.66	13.64	9.848
Testing Set	0.005781	0.001268	0.001084	13.88	12.31	9.556

Table 3.3 Evaluation metric for the second best FNN hyperparameter configurations

During the evaluation of various PINN architectures with different hyperparameters, it was observed that the FNN architecture demonstrated promising performance. The FNN

architecture achieved an error reduction of approximately 10% for the flow quantities before reaching a plateau at around 200 epochs. The optimal learning rate was determined to be 0.001, ensuring effective convergence during training. Additionally, for optimal accuracy and computational efficiency, a batch size of 3000 was identified as the preferred choice.

3.2.2 SPINN Results

Figure 3.3 Evaluation data throughout the training process for the best SPINN hyperparameter configuration. Took 470 seconds to run 200 epochs

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.005806	0.001200	0.001319	12.10	13.17	9.32
Testing Set	0.005304	0.001121	0.001319	12.37	12.05	9.045

Table 3.4 Evaluation metric for the best SPINN hyperparameter configurations

Figure 3.4 Evaluation data throughout the training process for the second-best SPINN hyperparameter configuration. Took 406 second to run 200 epochs

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.006541	0.001363	0.001386	12.79	13.73	9.877
Testing Set	0.006048	0.001283	0.001386	13.05	12.39	9.581

Table 3.5 Evaluation metric for the second-best SPINN hyperparameter configuration

During the evaluation of different PINN architectures with varied hyperparameters, the MSPINN architecture exhibited similar performance to the FNN architecture. However, when utilizing a smaller number of epochs, the MSPINN architecture demonstrated faster convergence compared to FNN. The optimal learning rate for the MSPINN architecture, like the FNN architecture, was determined to be 0.001. Furthermore, a batch size of 3000 proved to be the optimal choice for achieving both accuracy and computational efficiency.

3.2.3 MSPINN Results

The best parameter configurations for the FNN architecture are shown in Figure 3. and took 253 second to run 20 epochs.

Figure 3.5 Evaluation data throughout the training process for the best MSPINN hyperparameter configuration. Took 253 seconds to run 20 epochs

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.0130	0.004280	0.002231	18.25	13.08	10.93
Testing Set	0.01146	0.003967	0.002156	18.86	12.52	10.19

Table 3.6 Evaluation metric for the best MSPINN hyperparameter configuration

Figure 3.6 Evaluation data throughout the training process for the second best MSPINN hyperparameter configuration. Took 108 second to run 50 epochs

Testing set	Loss			Error		
	Data	Eqn	Boundary	p	u	v
Training Set	0.03034	0.008026	0.03004	17.41	18.74	19.17
Testing Set	0.02809	0.007922	0.03004	17.88	17.54	18.52

Table 3.7 Evaluation metric for the second-best MSPINN hyperparameter configuration

When evaluating the performance of different PINN architectures with varying hyperparameters, the MSPINN architecture exhibited the poorest performance. Specifically, when using larger batch numbers, increasing the number of epochs beyond 50 had minimal impact on reducing the error. Conversely, for smaller batch numbers, the error fluctuated significantly throughout the training process. As a result, the final evaluation metrics of the model became highly dependent on chance, as they were influenced by the random fluctuations of the errors at that particular epoch.

3.2 Optimal Model Evaluation

Based on the evaluation results, the SPINN architecture emerges as the optimal choice due to its ability to converge to a smaller error plateau within a smaller number of epochs. By employing the Stan activation function, a batch size of 1,000, and a learning rate of 0.001, the SPINN model can achieve convergence within 100 epochs, as depicted in Figure 3.5. Therefore, the recommended optimal model configuration is to utilize SPINN with the hyperparameters (learning rate: 0.001, activation function: Stan, batch size: 1,000, layer configuration: [1, 1, 1]).

To further test the capabilities of the optimal model, we train it on two different airfoils, namely NACA8646 and NACA0005 and visualise the flow. The objective is to examine whether the optimal PINN design can effectively capture and visualize important flow features specific to different airfoil designs. Specifically, we are interested in observing how

closely the optimal PINN model can represent essential flow characteristics compared to real flow features. Among the crucial flow features of interest are the pressure distribution below the leading edge, where a low pressure can indicate high lift. Additionally, the velocity distribution in both the x-direction and y-direction provides insights into the flow patterns around the airfoil. By evaluating these aspects, we can determine the usefulness of the optimal PINN design in capturing important flow features accurately.

3.2.1 NACA8646 Flow Visualisation

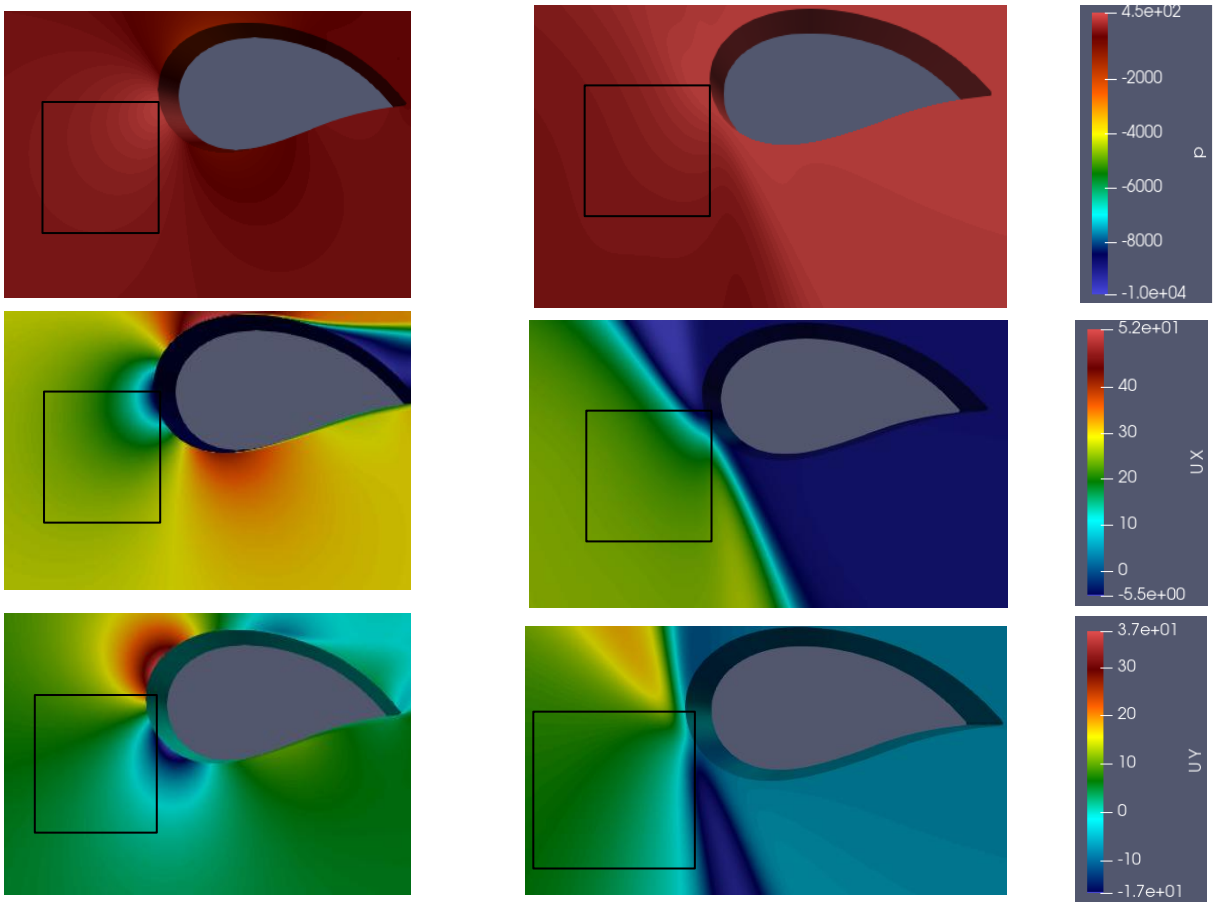


Figure 3.7 Simulated (on the left) vs predicted Reynolds (on the right) averaged flow quantities around the NACA 8646 airfoil. The images from top to bottom are the pressure, velocity in the x direction and velocity in the y direction. The problem domain is outline with a black box

Regarding the pressure distribution, the predicted flow maintains a remarkably similar pressure pattern to that of the simulated flow. The pressure contours exhibit a curvature that resembles the circular patterns formed by the simulated flow. This suggests that the predicted pressure distribution is capturing the essential features of the flow accurately. The velocity in the x-direction, the distribution is also reasonably well-maintained in the predicted flow. However, there is a noticeable difference in at the top right of the square. The small circular pattern seen in the simulated flow does not appear in the predicted flow, and there is less curvature overall. Despite these discrepancies, the general distribution of the velocity in

the x-direction is captured to a satisfactory extent. In terms of the velocity in the y-direction, the predicted distribution is again highly accurate. Within the problem domain, the small region of blue representing low velocity and the small region of yellow indicating velocity in the 10-20 range are accurately predicted. This suggests that the predicted flow field is capable of capturing the distributions of the quantities accurately but struggles with small curves patterns in the flow.

3.2.1 NACA0005 Flow Visualisation

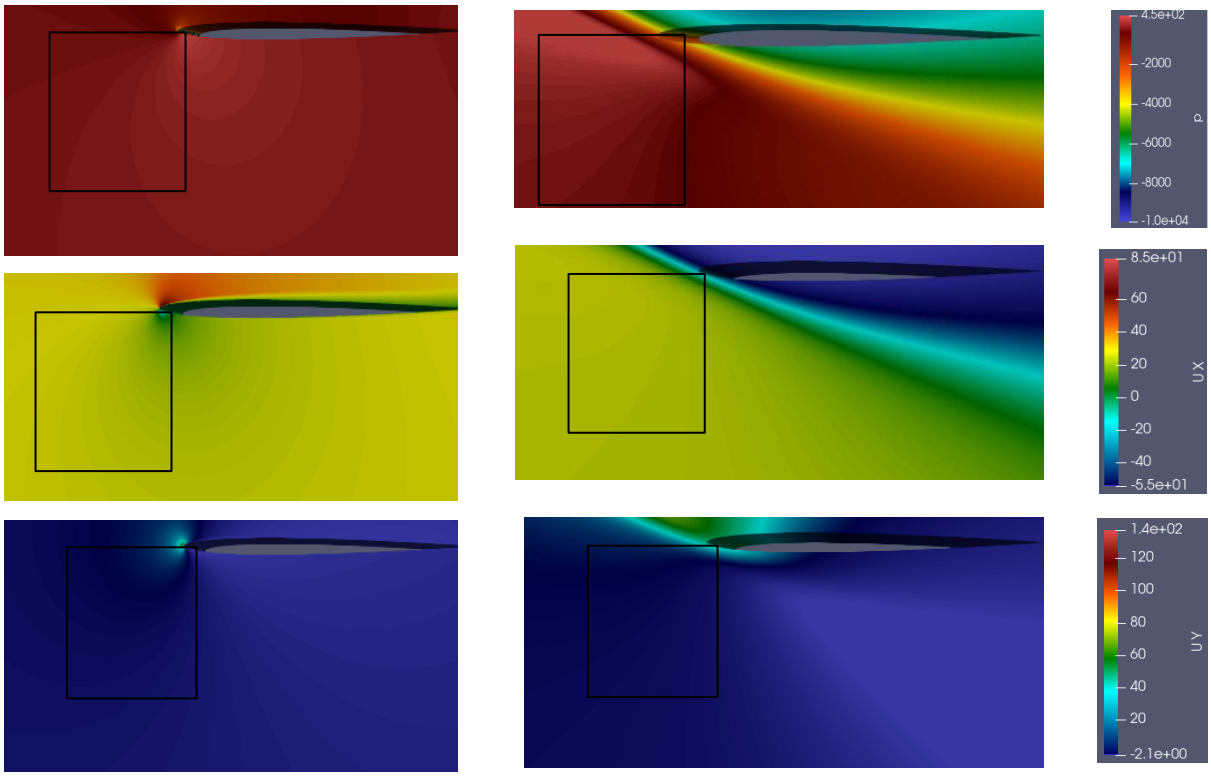


Figure 3.8 Simulated (on the left) vs predicted Reynolds (on the right) averaged flow around the NACA 0005 airfoil. The images from top to bottom are the pressure, velocity in the x direction and velocity in the y direction

Upon evaluating the model's performance on the NACA0005 airfoil, which was not included in the model optimization process, some insights can be gained. The results show that the selected hyperparameters perform reasonably well in capturing certain flow features of the NACA0005 airfoil. When examining the distributions, the predicted flow maintains a similar pressure pattern to that of the simulated flow, similar to the observations for the NACA8646 airfoil. However, there are noticeable differences in the shapes of the flow field. The predicted flow field exhibits less curvature compared to the simulated flow, even though the circular patterns are not particularly small. These deformations suggest that the model's ability to accurately capture the intricate details of the flow, especially in terms of curvature, may vary when applied to different airfoil designs.

Chapter 4
Discussion

4.1 Conclusions

Our results show the PINN can typically predict flow quantities with an error of approximately 10%. It should be noted that all models plateaued to an error within less than 5 minutes making this much more efficient regular CFD techniques with usually require many core hours. However, the average error of 10% is far too high for any numerical application we may want to use for engineering however we can get a fairly accurate read of the pressure and velocity distributions of flow and get an abstract deformed shape of the flow by visualising the data. This may be useful for engineer to run quick and dirty test to get an idea of what sort of pressure distributions the airfoil will be subject to.

The obtained results from this project demonstrate that the PINN approach can predict flow quantities with an error of approximately 10%. This level of accuracy, while not suitable for precise numerical engineering applications, still provides valuable insights into the pressure and velocity distributions of the flow. Additionally, the visualized data allows for obtaining an abstract representation of the deformed shape of the flow. Compared to traditional computational fluid dynamics (CFD) techniques, the advantage of the PINN lies in its efficiency. The models in our study consistently reached a plateau in terms of error within a relatively short timeframe of less than 5 minutes. This efficiency is a significant improvement over conventional CFD methods, which often require extensive computational resources and hours of computation time. Although Eivazi et al. (2022) did not provide data into the speed of their model we can safely assume that our optimal model is faster as it has a reduced data loss term and less parameter to optimise.

Although the average error of 10% may be considered high for rigorous engineering applications, it is important to note the utility of these predictions as quick and rough estimates. Engineers can use the PINN-based approach to obtain a rapid and approximate understanding of the pressure distributions that an airfoil may experience. This can provide valuable insights during the initial stages of design or when conducting preliminary analyses. By visualizing the data, engineers can gain qualitative insights into the flow behaviour and obtain a broad understanding of the flow patterns. This allows for a quick rudimentary assessment and evaluation of different airfoil designs without the need for time-consuming simulations.

4.2 Ideas for future work

This project has demonstrated the potential of PINNs in predicting the flow field around an airfoil given limited information. However, due to limitations in computational resources, we focused on predicting a small area of the flow, restricting the problem domain. In future work, it would be advantageous to overcome these limitations by leveraging the power of GPU computing through PyTorch's CUDA support. By utilizing parallel computing, PINN calculations can be significantly accelerated, enabling researchers to incorporate additional inputs and output features into the model to simulate various variations of the airfoil case.

Future work on PINNs for predicting turbulent fluid flow around an airfoil can focus on leveraging GPU computing, extending the model to three dimensions, enhancing generalization to different airfoils, accounting for compressible flows, or incorporating variations in flow conditions. These advancements will contribute to a more comprehensive and versatile prediction framework for airfoil design and analysis.

One aspect to consider is extending the PINN to predict flows in three dimensions instead of just two dimensions. This enhancement would be crucial for airfoil design, as three-dimensional flows play a vital role. By introducing a third dimension as an input, the PINN can simulate and capture the complexities of three-dimensional flow phenomena.

Moreover, the current PINN model focuses on predicting the flow around a single airfoil. To enhance its generalization capability, incorporating input features that represent the characteristics of NACA 4 digits airfoils would allow the PINN to predict the flow around any airfoil within the NACA 4-digit family. This expansion would greatly increase the versatility and applicability of the model to airfoil design and optimisation. A good starting point for this could simply use the 4-digit code that uniquely identifies these airfoils as these number as represent the maximum thickness, mean camber and position of mean camber of the airfoils. These are the only featured required to create one of these airfoils so it should be enough to capture how the flow will vary with the geometry of the airfoil.

Another important aspect to address is the limitation of predicting only incompressible flow, assuming a constant density. By incorporating a density output feature, the PINN can account for spatially varying densities, enabling predictions for compressible flows. This extension would allow for a wider range of applications.

Furthermore, the PINN was trained on specific flow conditions, such as a fixed free stream velocity and angle of attack. To capture a broader range of flow scenarios, additional input features can be included to vary the angle of attack and represent different airfoil flow conditions. By incorporating these variations, the PINN can provide insights into the flow characteristics under various operating conditions.

List of References

Vinuesa, R., Hosseini, S.M., Hanifi, A., Henningson, D.S. and Schlatter, P., 2015. Direct numerical simulation of the flow around a wing section using high-order parallel spectral methods. In Ninth International Symposium on Turbulence and Shear Flow Phenomena. Begel House Inc.

Ling, J., Kurzawski, A. and Templeton, J. (2016) "Reynolds averaged turbulence modelling using deep neural networks with embedded invariance," Journal of Fluid Mechanics. Cambridge University Press, 807, pp. 155–166. doi: 10.1017/jfm.2016.615.

Zhu, L., Zhang, W., Sun, X., Liu, Y. and Yuan, X., 2021. Turbulence closure for high Reynolds number airfoil flows by deep neural networks. Aerospace Science and Technology, 110, p.106452.

Beck, A., Flad, D. and Munz, C.D., 2019. Deep neural networks for data-driven LES closure models. Journal of Computational Physics, 398, p.108910.

Reynolds, O., 1895. IV. On the dynamical theory of incompressible viscous fluids and the determination of the criterion. Philosophical transactions of the royal society of london.(a.), (186), pp.123-164.

McCulloch, W.S. and Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5, pp.115-133.

Singer, Y., 2016. Advanced optimization. Lecture Notes, AM221 Lecture7. pdf.

Ruder, S., 2016. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Brunton, S.L., Noack, B.R. and Koumoutsakos, P., 2020. Machine learning for fluid mechanics. Annual review of fluid mechanics, 52, pp.477-508.

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

Menter, F.R., 1994. Two-equation eddy-viscosity turbulence models for engineering applications. AIAA journal, 32(8), pp.1598-1605.

Spalart, P. and Allmaras, S., 1992, January. A one-equation turbulence model for aerodynamic flows. In 30th aerospace sciences meeting and exhibit (p. 439).

Kingma, D.P. and Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Cai, S., Mao, Z., Wang, Z., Yin, M. and Karniadakis, G.E., 2021. Physics-informed neural networks (PINNs) for fluid mechanics: A review. Acta Mechanica Sinica, 37(12), pp.1727-1738.

Eivazi, H., Tahani, M., Schlatter, P. and Vinuesa, R., 2022. Physics-informed neural networks for solving Reynolds-averaged Navier–Stokes equations. Physics of Fluids, 34(7), p.075117.

He, Q., Barajas-Solano, D., Tartakovsky, G. and Tartakovsky, A.M., 2020. Physics-informed neural networks for multiphysics data assimilation with application to subsurface transport. Advances in Water Resources, 141, p.103610.

Agarap, A.F., 2018. Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375.

Zeman, D., Hajic, J., Popel, M., Potthast, M., Straka, M., Ginter, F., Nivre, J. and Petrov, S., 2018, October. CoNLL 2018 shared task: Multilingual parsing from raw text to universal dependencies. In Proceedings of the CoNLL 2018 Shared Task: Multilingual parsing from raw text to universal dependencies (pp. 1-21).

Schillaci, A., Quadrio, M., Pipolo, C., Restelli, M. and Boracchi, G., 2021, January. Inferring functional properties from fluid dynamics features. In 2020 25th International Conference on Pattern Recognition (ICPR) (pp. 4091-4098). IEEE.

LeCun, Y., Bottou, L., Orr, G.B. and Müller, K.R., 2002. Efficient backprop. In Neural networks: Tricks of the trade (pp. 9-50). Berlin, Heidelberg: Springer Berlin Heidelberg.

Gnanasambandam, R., Shen, B., Chung, J. and Yue, X., 2022. Self-scalable tanh (stan): Faster convergence and better generalization in physics-informed neural networks. arXiv preprint arXiv:2204.12589.

Appendix A
Self-appraisal

A.1 Critical self-evaluation

The primary objective of this project was to conduct a comprehensive analysis and testing of various PINN implementations for fluid flow predictions with limited data. Through systematic evaluation of hyperparameter configurations, the project aimed to identify the most efficient and accurate PINN design. Overall, the project successfully achieved this objective by finding a PINN design that demonstrated fast computation time and competitive accuracy compared to other configurations.

As a secondary objective, the project aimed to deliver quick and accurate results compared to numerical simulations. Although the exact time data for the CFD simulations used in Schillaci et al. (2021) was not available, it is well-known that turbulence modelling CFD simulations typically require hours to complete. Therefore, the fact that the PINN model could be trained in a few minutes indicates its computational efficiency. However, the accuracy of the model, with an error of approximately 10%, falls short of numerical applications' requirements for airfoil design.

Despite the limitation in accuracy, the project demonstrated that visualizing the model data can still provide valuable information, such as accurate pressure and velocity distributions, as well as a simplified representation of the flow shape. This visualization aspect can be particularly useful in engineering applications, allowing for quick assessments of pressure distributions and flow characteristics. It should be noted that the PINN was trained on a laptop without a GPU. Expanding the search space of hyperparameters and training the model on a GPU-equipped system may lead to more accurate predictions.

In conclusion, this project successfully fulfilled its main goal of thoroughly analysing different PINN implementations for fluid flow predictions with limited data. While it falls short in terms of accuracy compared to numerical simulations, the project offers alternative uses of the model that can still provide valuable insights in the engineering discipline it aims to assist.

A.2 Personal reflection and lessons learned

Overall, the project can be considered mostly successful and has the potential to become a valuable tool in assisting airfoil design and optimization. Despite encountering numerous hiccups and obstacles along the way, I am proud of the final results and the progress made throughout the project. The process was long and often frustrating, but it was ultimately rewarding to see the project come together.

One of the most satisfying aspects of this project was the significant learning experience it provided. Prior to embarking on this project, I had almost no knowledge of neural networks or fluid-based modelling. However, driven by my fascination for both physics and artificial intelligence, I was motivated to learn and acquire expertise in these areas. I am proud of the amount of knowledge I was able to gain within a relatively short period of time and the level of expertise I developed through the background research conducted for this project.

One key skill that I developed during the project was the ability to perform in-depth background research. Understanding the existing literature and current approaches in the field of fluid flow predictions and neural networks was crucial in guiding the development of the project. This skill allowed me to gather relevant information, identify gaps in the research, and explore novel architectures.

Additionally, the project built my ability to code efficient neural networks, including both established and novel architectures. Building and implementing neural networks with optimized performance and accuracy required careful consideration of various factors such as hyperparameters, activation functions, data pre-processing and feature scaling. The project provided valuable hands-on experience in designing and fine-tuning neural network architectures to achieve the desired results.

In summary, despite the challenges faced, the project yielded valuable outcomes and contributed to my personal growth and skill development. The project's success and the knowledge gained through extensive research and coding will serve as a solid foundation for further exploration and advancements in the field of fluid flow predictions and neural networks.

Appendix B
External Materials

As described in section 2.1 the dataset used to train the PINN models in this project was compiled by Schillaci (2021). It contains a set of numerical simulations of turbulent flow around an airfoil using the simplest and computationally affordable CFD approach. That is solving the RANS equations using the Spalart-Allmaras turbulence model. The dataset can be downloaded from the URL https://zenodo.org/record/4106752

The software ParaView version 5.11.0 was used to visualise the flow fields. ParaView is an open-source data visualization and analysis software widely used in scientific and engineering fields. It is particularly useful for datasets generated from computational simulations such as the dataset used for this project.

The python libraries VTK, NumPy, Pandas, Matplotlib, Seaborn and PyTorch were used. For more details see section 2.2.

Appendix C
Code

C.1 data_extracter.py

```
def __init__(self, NACA_code):
    # change current working directory to the directory this file is in
    current_script_directory = os.path.dirname(os.path.abspath(__file__))
    os.chdir(current_script_directory)
    self.NACA_code = NACA_code

    ''' Airfoil geometry '''
    self.III = int(self.NACA_code[2:4]) / 100
    # get airfoil geometry from the vtk file and store it as a pandas dataframe
    vtk_airfoil_gemetry_data = self._get_airfoil_geometry_data()
    self._geometry = pd.DataFrame(converter.vtk_to_numpy(vtk_airfoil_gemetry_data.GetPoints().GetData()),
                                  columns=["x", "y", "z"])

    ''' Flow field '''
    # get the field data from the vtk file and store it as pandas dataframe
    coords, pressure, velocity = self._get_flow_field_data()
    self._flow_field = pd.concat([pd.DataFrame(coords, columns=["x", "y", "z"]), pd.DataFrame(velocity,
                                                                                              columns=["u", "v", "w"])], axis=1)
    self._flow_field["p"] = pd.Series(pressure)

    # get 2D boundary points and flow field datasets
    self.boundary_points = self._geometry.copy()
    # remove third dimension
    self.boundary_points = self.boundary_points.loc[self.boundary_points['z'] == 0].copy()
    self.boundary_points = self.boundary_points.get(["x", "y"])

    self.reduced_domain, self.train, self.test,= self._get_datasets(-0.5, 0, -0.5, 0, 4000, 1386)
```

Figure C.1

```
def _get_airfoil_geometry_data(self):
    # airfoil geometry file
    AGF = os.path.join("data", self.NACA_code, self.NACA_code+"_walls.vtk")
    if os.path.isfile(AGF):
        # read data from airfoil geometry file
        reader = vtkPolyDataReader()
        reader.SetFileName(AGF)
        reader.Update()
        return reader.GetOutput()
```

Figure C.2

```
# convert vtk flow field file data to numpy arrays
def _get_flow_field_data(self):
    # field flow file
    FFF = os.path.join("data", self.NACA_code, self.NACA_code+".vtk")
    if os.path.isfile(FFF):
        # read in data from field flow file
        reader = vtkUnstructuredGridReader()
        reader.SetFileName(FFF)
        reader.Update()

        # We now need to convert this data into array
        #we can convert vtkDataArray into a numpy array using vtk_to_numpy method imported with the vtk.util.numpy_support module
        data = reader.GetOutput()

        # the data consists of point coordinates and point data and is stored as a vtkUnstructeredGrid:
         # point coordinates stored as a vtkDataArray in a vtkPoints class
        coords_data = converter.vtk_to_numpy(data.GetPoints().GetData())
        # point data stored as vtkPointData which is a subclass of vtkFieldData and consists of multiple vtkDataArrays
        flow_field_data = data.GetPointData()


        # the point data consists of the pressure and velocity fields:
         # pressure field is the first vtkDataArray
        pressure_field = converter.vtk_to_numpy(flow_field_data.GetArray(0))
        # velocity field is the second vtkDataArray and is a 2D array
        velocity_field = converter.vtk_to_numpy(flow_field_data.GetArray(1))
        return coords_data, pressure_field, velocity_field
```

Figure C.3

```python
def _get_datasets(self, min_x, max_x, min_y, max_y, train_size, test_size):
    # define original reduced and extrapolation domains
    original_domain = self._flow_field.copy()
    # remove third dimension
    original_domain = original_domain.loc[original_domain['z'] == 0]
    original_domain = original_domain.get(["x", "y", "p", "u", "v"])

    # get reduced domain
    reduced_domain = original_domain.loc[(self._flow_field['z'] == 0) &
                                         (self._flow_field["x"] > min_x) &
                                         (self._flow_field["x"] < max_x) &
                                         (self._flow_field["y"] > min_y) &
                                         (self._flow_field["y"] < max_y)].copy()

    # randomly sample the necessary number of points from the reduced domain
    # random state set to 42 for reproducability
    dataset_points = reduced_domain.sample(n=train_size + test_size, random_state=42).copy()
    #split the points among the train, validation and test sets
    train = dataset_points.iloc[:train_size]
    test = dataset_points[train_size:]

    return reduced_domain, train, test
```

Figure C.4

```python
# create new flow_field data using the PINN prediction and creat a vtk file with them
def generate_vtk_file(self, model, coords_mins, coords_ranges, D_mins, D_ranges):
    # field flow file
    FFF = os.path.join("data", self.NACA_code, self.NACA_code+".vtk")
    if os.path.isfile(FFF):
        # read in data from field flow file
        reader = vtkUnstructuredGridReader()
        reader.SetFileName(FFF)
        reader.Update()

        # change the _flow_field data
        data = reader.GetOutput()
        data.GetPointData().Initialize()

        # make vtkDoubleArray for pressure
        new_p = vtk.vtkDoubleArray()
        new_p.SetNumberOfComponents(1)
        new_p.SetNumberOfTuples(len(self._flow_field))
        new_p.SetName("p")


        # make vtkDoubleArray for pressure
        new_velocity = vtk.vtkDoubleArray()
        new_velocity.SetNumberOfComponents(2)
        new_velocity.SetNumberOfTuples(len(self._flow_field))
        new_velocity.SetName("velocity")

        coords = (torch.tensor(np.array(self._flow_field[["x", "y"]])) - coords_mins) / coords_ranges
        D_hat = (model(coords)[:, 0:3] * D_ranges) + D_mins
        p_data, velocity_data = D_hat[:, 0].detach().numpy(), D_hat[:, 1:3].detach().numpy()

        for i, value in enumerate(p_data):
            new_p.SetValue(i, value)

        for i, value in enumerate(velocity_data):
            new_velocity.SetTuple(i, value)

        data.GetPointData().AddArray(new_p)
        data.GetPointData().AddArray(new_velocity)

        # write the grid
        writer = vtkUnstructuredGridWriter()
        writer.SetFileName(os.path.join("preds", self.NACA_code+"_pred"+".vtk"))
        writer.SetInputData(data)
        writer.Write()
```

Figure C.5

C.2 PINN.py

```python
# implement Stan activation function as a pytorch module
class Stan(nn.Module):
    def __init__(self, layer_size):
        super(Stan, self).__init__()
        # initialise beta for each neuron in a layer
        # neuron is an nn.Parameter so it will be trained through when the PINN makes a backwards pass
        self.beta = nn.Parameter(torch.zeros(layer_size))

    # the forward method of an pytorch neuron implements the non linear activation function
    def forward(self, x):
        # Stan activation function equation Stan(x) = tanh(x) + β_k^i tanh(x)
        output = torch.tanh(x) + self.beta * x * torch.tanh(x)
        return output
```

Figure C.6

```python
def __init__(self, NN_architecture, activation_function, loss_term_weights, batch_size, lr, num_epochs, layers, restore_file=None):
    super().__init__()

    # initialise equation constants
    # fixed 1/Re
    self._1_div_Re =1/3000000
    self._1_div_density = 0.81632

    # initialise PINN architecture
    if NN_architecture == "FNN":
        # FNN is a basic feed forward NN and it's structure is initialised by the initialise_NN method
        # this method outputs a list that is then converted to an OrderedDict and then to a NN
        self.NN = nn.Sequential(OrderedDict(self.initialise_NN(layers, activation_function)))

    elif NN_architecture == "SPINN" or NN_architecture == "MSPINN":
        # for both SPINN and MSPINN we need the top and bottom NNs which are each half the size of the FNN
        half_layers = list((np.array(layers) / 2).astype(int))
        half_layers[0] = 2
        half_layers[-1] = 3

        if NN_architecture == "SPINN":
            # structure of both the top and the bottom NN initialised by the initialise_NN method
            self.top_NN = nn.Sequential(OrderedDict(self.initialise_NN(layers, activation_function)))
            self.bottom_NN = nn.Sequential(OrderedDict(self.initialise_NN(layers, activation_function)))

        if NN_architecture == "MSPINN":
            half_layers.pop()
            print(half_layers)
            # we remove the last output layer of the top and bottom NNs
            # as we need to define the mixing layer between the secondlast and last layer
            self.top_NN = nn.Sequential(OrderedDict(self.initialise_NN(half_layers, activation_function)))
            self.bottom_NN = nn.Sequential(OrderedDict(self.initialise_NN(half_layers, activation_function)))
            # nn.Bilinear implements the mixing synapse
            self.mixing_synapse = nn.Bilinear(half_layers[-1], half_layers[-1], 2 * half_layers[-1])

    else:
        print("unrecognises neural network type provided please try again")
```

Figure C.7

```python
# initialises a simple FNN where layers is a list that defines how many nerons are in each layer
def initialise_NN(self, layers, activation_function):
    # stores synapses and neurons
    synapses_and_neurons = list()

    # for every hidden layer synapse except the last we alternate between
    for i in range(0,len(layers)-1):
        # adding a synapse
        synapse = nn.Linear(layers[i],layers[i+1])
        synapses_and_neurons.append(("lin"+str(i+1), synapse))

        # and adding a layer of neurons
        if i < len(layers)-2:
            if activation_function == Stan:
                synapses_and_neurons.append(("Stan"+str(i+1), activation_function(layers[i+1])))
            else:
                synapses_and_neurons.append(("Tanh"+str(i+1), activation_function()))

    return synapses_and_neurons
```

Figure C.8

```python
# implments forward pass of the PINN
def forward(self, z_0):
    # for FNN we can simply pass the inouts through the NN and get the outputs
    if self.NN_architecture == "FNN":
        z_L = self.NN(z_0)
        return z_L

    if self.NN_architecture == "SPINN" or self.NN_architecture == "MSPINN":
        top_z_L = self.top_NN(z_0)
        bottom_z_L = self.bottom_NN(z_0)

        if self.NN_architecture == "MSPINN":
            # pases the outputs of the last hidden layer of the top and bottom NNs
            # through mixing synapse to the output layer
            mixed_z_L = self.mixing_synapse(top_z_L, bottom_z_L)
            return mixed_z_L

        combined_z_L = torch.cat((top_z_L, bottom_z_L), dim=1)
        return combined_z_L
```

Figure C.9

```python
# implements calulating loss and back propogation as well as storing eval data for graphs
def train(self, data_loader):
    # initialise start time and calculate loss before training so it can be plotted as the datapoint at epoch 0
    start_time = time.time()
    if len(self.epochs) == 0:
        self.epochs.append(0)
        self.evaluate(data_loader.dataset, training=True)

    # gradient decent
    for epoch in range(1,num_epochs+1):
        # for batch in dataset
        for coords_batch, D_batch in data_loader:
            #
            self.optimiser.zero_grad()
            # set requires_grad to True for the imputs so we can partially differentiate with then in thr equation loss
            coords_batch.requires_grad_()
            # forward pass takes a batch of coords and ouputs a batch of flow quatities (a batch_size by 6 matrix)
            # This is much more efficient than a for loop as it uses matrix multiplication and operations
            O_hat_batch = self.forward(coords_batch)
            # slice batch of outputs to get the batch of flow quantities used to calculate data loss
            D_hat_batch = O_hat_batch[:, 0:3]

            # apply equations for data, equation and boundary condition loss
            data_loss = torch.mean(torch.norm(D_batch - D_hat_batch, dim=1) **2)
            equation_loss = self.eqn_loss(coords_batch, O_hat_batch)
            boundary_loss = self.boundary_condition_loss(data_loader.dataset.boundary_points, 0.1)

            # multiply each by their corresponding loss term weight and add them all together
            loss = self.w[0]*data_loss + self.w[1]*equation_loss + self.w[2]*boundary_loss
            # uses gradient descent to calculate the derivative of the loss with respects to each parameter (dl/dp)
            # then adds this value to p.grad (the stored sum of the gradients)
            loss.backward()
            # the adam optimiser then takes a step for each parameter using its equations
            # it also updates some of its stores values as the sum dl/dp and the sum of (dl/dp)^2
            self.optimiser.step()

        # after every epoch we store the eval data so it can be graphed
        self.epochs.append(len(self.epochs))
        self.evaluate(data_loader.dataset, training=True)

        # stopping criteria: if the loss is less than 0.001 or training takes longer than 10 mintes
        print(epoch)
        if loss < 0.0001:
            print("Success")
            break
        elif (time.time() - start_time) > (10*60):
            print("Training took too long")
            break

    # display graphs of the evaluation data against epoch so we can track the conbergence
    self.display_graphs()
    time_taken = time.time() - start_time
    return time_taken
```

Figure C.10

```python
def eqn_loss(self, coords, O_hat_batch):
    # slice the ouput tensor to get a batch of each flow quantity. E.g. p is 1 batch_size x 1 vector
    p, u, v, uu, uv, vv = O_hat_batch[:, 0], O_hat_batch[:, 1], O_hat_batch[:, 2], O_hat_batch[:, 3], O_hat_batch[:, 4], O_hat_batch[:, 5]

    # cakculate the derivatives of the flow quatities with respects to the input coords
    u_derivs = torch.autograd.grad(u, coords, grad_outputs=torch.ones_like(u), create_graph=True, retain_graph=True)[0]
    u_x, u_y = u_derivs[:, 0], u_derivs[:, 1]

    v_derivs = torch.autograd.grad(v, coords, grad_outputs=torch.ones_like(v), create_graph=True, retain_graph=True)[0]
    v_x, v_y = v_derivs[:, 0], v_derivs[:, 1]

    p_derivs = torch.autograd.grad(p, coords, grad_outputs=torch.ones_like(p), create_graph=True, retain_graph=True)[0]
    p_x, p_y = p_derivs[:, 0], p_derivs[:, 1]

    uu_derivs = torch.autograd.grad(uu, coords, grad_outputs=torch.ones_like(uu), create_graph=True, retain_graph=True)[0]
    uu_x, uu_y = uu_derivs[:, 0], uu_derivs[:, 1]

    uv_derivs = torch.autograd.grad(uv, coords, grad_outputs=torch.ones_like(uv), create_graph=True, retain_graph=True)[0]
    uv_x, uv_y = uv_derivs[:, 0], uv_derivs[:, 1]

    vv_derivs = torch.autograd.grad(vv, coords, grad_outputs=torch.ones_like(vv), create_graph=True, retain_graph=True)[0]
    vv_x, vv_y = vv_derivs[:, 0], vv_derivs[:, 1]

    # calculate the doube derivatives
    u_doublex_derivs = torch.autograd.grad(u_x, coords, grad_outputs=torch.ones_like(u_x), create_graph=True, retain_graph=True)[0]
    u_xx, u_xy = u_doublex_derivs[:, 0], u_doublex_derivs[:, 1]

    u_doubley_derivs = torch.autograd.grad(u_y, coords, grad_outputs=torch.ones_like(u_y), create_graph=True, retain_graph=True)[0]
    u_yx, u_yy = u_doubley_derivs[:, 0], u_doubley_derivs[:, 1]

    v_doublex_derivs = torch.autograd.grad(v_x, coords, grad_outputs=torch.ones_like(v_x), create_graph=True, retain_graph=True)[0]
    v_xx, v_xy = v_doublex_derivs[:, 0], v_doublex_derivs[:, 1]

    v_doubley_derivs = torch.autograd.grad(v_y, coords, grad_outputs=torch.ones_like(v_y), create_graph=True, retain_graph=True)[0]
    v_yx, v_yy = v_doubley_derivs[:, 0], v_doubley_derivs[:, 1]

    # calculate the equation residuals
    f1 = u_x + v_y
    f2 = (u*u_x) + (v*u_y) + uu_x + uv_y + (self._1_div_density*p_x) - (self._1_div_Re*(u_xx + u_yy))
    f3 = (u*v_x) + (v*v_y) + uv_x + vv_y + (self._1_div_density*p_y) - (self._1_div_Re*(v_xx + v_yy))
    # sum the residuals to get the equation loss
    equation_loss = torch.mean(f1**2 + f2**2 + f3**2)

    return equation_loss
```

Figure C.11

```python
# calculates the boundary condition loss for Nb the boundary points
# where Nb = batch_size * sample_frac
def boundary_condition_loss(self, boundary_points, sample_frac):
    # randomly sample boundary points
    Nb = min(self.batch_size * sample_frac, len(boundary_points))
    indicies = torch.randperm(len(boundary_points))[:Nb]
    sampled_boundary_points = boundary_points[indicies]

    # calculate boundary loss
    boundary_conditons_loss = 0
    O_hat = self.forward(sampled_boundary_points)
    u, v = O_hat[:, 1], O_hat[:, 2]
    return torch.mean(u**2 + v**2)
```

Figure C.12

```python
# calculates the PINNs evaluation metrics on a given dataset
def evaluate(self, dataset, training=False):
    # This code is very similar to the train method code where the loss is calculated
    coords = dataset.coords
    D = dataset.D

    O_hat = self.forward(coords)
    D_hat = O_hat[:, 0:3]

    # except we convert the tensors to floats so we can plot the values
    data_loss = float(torch.mean(torch.norm(D - D_hat, dim=1)**2))
    equation_loss = float(self.eqn_loss(coords, O_hat))
    boundary_loss = float(self.boundary_condition_loss(dataset.boundary_points, 1))
    total_loss = self.w[0]*data_loss + self.w[1]*equation_loss + self.w[2]*boundary_loss

    # and we calculate the error in each flow quantity
    p = D[:, 0] + (dataset.D_mins[0] / 1000)
    p_error = float(torch.mean(abs((p - D_hat[:, 0]) / p) * 100))
    u = D[:, 1] + (dataset.D_mins[0] / 1000)
    u_error = float(torch.mean(abs((u - D_hat[:, 1]) / u) * 100))
    v = D[:, 2] + (dataset.D_mins[0] / 1000)
    v_error = float(torch.mean(abs((v - D_hat[:, 2]) / v) * 100))

    # if this method is called while training the NN it will add the evaluation metric to their respective lists
    # this is called every epoch so the lists will track the evaluation data per epoch so it can be graphed
    if training:
        self.data_losses.append(data_loss)
        self.eqn_losses.append(equation_loss)
        self.boundary_condition_losses.append(boundary_loss)
        self.total_losses.append(total_loss)
        self.p_errors.append(p_error)
        self.u_errors.append(u_error)
        self.v_errors.append(v_error)
    # otherwise the method was called after the PINN has been trained so we just print the evaluation data
    else:
        print(f"% error in p {p_error}")
        print(f"% error in u {u_error}")
        print(f"% error in v {v_error}")
        print(f"MSE of data is {data_loss}")
        print(f"MSE of equation is {equation_loss}")
        print(f"MSE of boundary points is {boundary_loss}")
```

Figure C.13

```python
# plots graphs of evaluation data thoughout the training process
def display_graphs(self):
    # create a plot made of 3 subplots
    fig, axes = plt.subplots(1, 3, figsize=(12, 4))

    # the first plot graphs of equation and boundary condition loss against epoch
    sns.set(style="whitegrid")
    sns.lineplot(x=self.epochs,y=self.eqn_losses, color="red", linewidth=2, label="Equation Loss", ax=axes[0])
    sns.lineplot(x=self.epochs,y=self.boundary_condition_losses, color="blue", linewidth=2, label="Boundary Loss", ax=axes[0])
    axes[0].set_ylabel("Loss")
    axes[0].set_xlabel("epoch")
    axes[0].set_title('Physics Informed')

    # the second plot graphs of data loss against epoch
    sns.set(style="whitegrid")
    sns.lineplot(x=self.epochs, y=self.data_losses, color="green", linewidth=2, label="Data Loss", ax=axes[1])
    axes[1].set_ylabel("Loss")
    axes[1].set_xlabel("epoch")
    axes[1].set_title('Data Driven')

    # the third plot the flow quantity error against epoch
    sns.set(style="whitegrid")
    sns.lineplot(x=self.epochs, y=self.p_errors, color="green", linewidth=2, label="p error", ax=axes[2])
    sns.lineplot(x=self.epochs,y=self.u_errors, color="red", linewidth=2, label="u error", ax=axes[2])
    sns.lineplot(x=self.epochs,y=self.v_errors, color="blue", linewidth=2, label="v error", ax=axes[2])
    axes[2].set_ylabel("% error")
    axes[2].set_xlabel("epoch")
    axes[2].set_title('Errors')

    plt.suptitle(f"lr: {self.lr}, activation: {self.activation_function}, batch_size: {self.batch_size}, loss term weights: {self.w}")
    plt.tight_layout()
    plt.show()
```

Figure C.14

```python
class Create_train_dataset(Dataset):
    def __init__(self, coords, D, boundary_points):
        coords = torch.tensor(coords)
        self.coords_mins = torch.min(coords, dim=0).values
        self.coords_maxs = torch.max(coords, dim=0).values
        self.coords_ranges = self.coords_maxs - self.coords_mins
        self.coords = (coords - self.coords_mins) / self.coords_ranges
        self.coords.requires_grad_()

        boundary_points = torch.tensor(boundary_points)
        self.boundary_points = (boundary_points - self.coords_mins) / self.coords_ranges

        D = torch.tensor(D)
        self.D_mins = torch.min(D, dim=0).values
        self.D_maxs = torch.max(D, dim=0).values
        self.D_ranges = self.D_maxs - self.D_mins
        self.D = (D - self.D_mins) / self.D_ranges

    def __len__(self):
        return len(self.coords)

    def __getitem__(self, index):
        return self.coords[index], self.D[index]
```

Figure C.15

```python
# creates a pytorch dataset for the test testing datasets
# needs mean and std of training dataset so it can normalise the data the same way the training dataset was normalised
class Create_test_dataset(Dataset):
    def __init__(self, coords, D, boundary_points, coords_mins, coords_ranges, D_mins, D_ranges):
        coords = torch.tensor(coords)
        self.coords = (coords - coords_mins) / coords_ranges
        self.coords.requires_grad_()

        boundary_points = torch.tensor(boundary_points)
        self.boundary_points = (boundary_points - coords_mins) / coords_ranges

        D = torch.tensor(D)
        self.D_mins = D_mins
        self.D = (D - D_mins) / D_ranges

    def __len__(self):
        return len(self.coords)

    def __getitem__(self, index):
        return self.coords[index], self.D[index]
```

Figure C.16

```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import OrderedDict
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
from torch.linalg import norm

from data_extracter import AirfoilData
import importlib
import sys
sys.path.append("stored_params")
```

Figure C.17