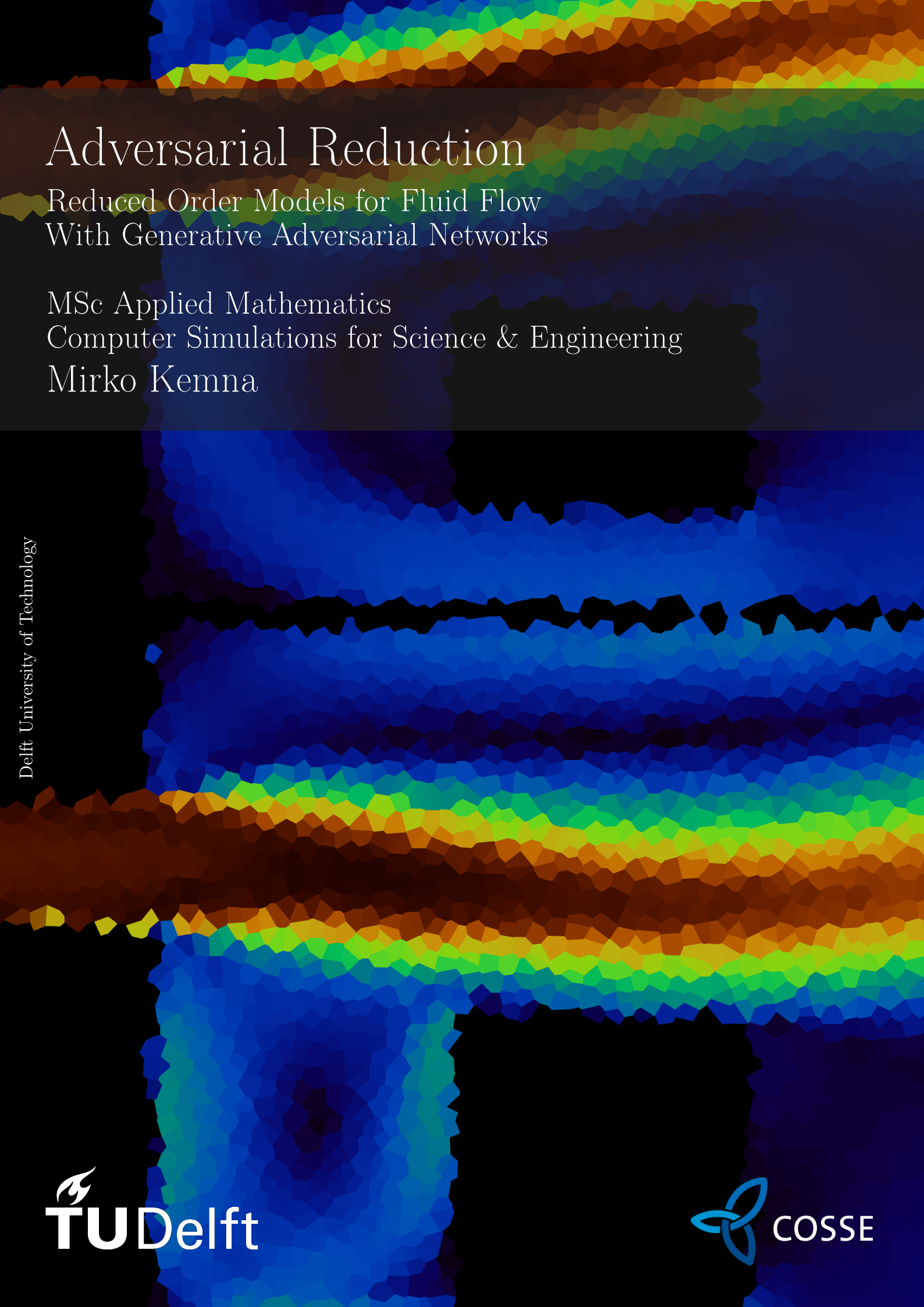


Adversarial Reduction

Reduced Order Models for Fluid Flow With Generative Adversarial Networks

MSc Applied Mathematics
Computer Simulations for Science & Engineering

Mirko Kemna

Delft University of Technology

TU Delft

COSSE

Reduced Order Models for Fluid Flow With Generative Adversarial Networks

by

Mirko Kemna

to obtain the degree of Master of Science in Applied Mathematics
at the Delft University of Technology.

Student number:	5606896	
Thesis committee:	Prof. dr. rer. nat A. Heinlein,	TU Delft, main supervisor
	Prof. dr. ir. C. Vuik,	TU Delft, second supervisor
	Prof. dr. K. S. Postek,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgments

I owe thanks and acknowledgment to everyone who supported me during my studies and enabled this thesis project to be completed. Special thanks first and foremost to my main supervisor Alexander Heinlein, for his valuable input and guidance. I also want to express my gratitude to second supervisor and coordinator of the COSSE program Kees Vuik, who's dedication to us students was a constant throughout my time at the TU Delft. Moreover, I want to thank Krzysztof Postek for kindly agreeing to join the thesis committee. A special thank you goes to Dylan Everingham and Genya Crossman for making my time in Delft so enjoyable, and to the rest of my fellow COSSE students for making this a very special program. Thanks also to Emil Baus for generously offering his time to proof-read this thesis. Lastly, I am very grateful for the rest of my friends & family, especially my partner Sofia, who I could always count on during this at times stressful period.

Mirko Kemna
Berlin, September 2022

Summary

The goal of this work is to evaluate the aptness of generative adversarial networks (GANs) for use as surrogate reduced order fluid models. In contrast to previously published work, the focus is placed on analyzing the specific effect of adversarial training, by comparing GAN outcomes with those from an identical generator network trained directly on ground truth (using an L1 loss). A dataset of 10 000 simulated examples of stationary flow through a 2D sudden expansion geometry containing a polygonal obstacle was created, alongside two additional datasets for testing generalization. The simulation data was interpolated to a regular image grid, and the neural networks were trained to predict the velocity field based on an image encoding the geometry. The gathered experimental data show clearly that adversarial training cannot reach the same accuracy as direct training. This was found to be true on unseen examples from the training distribution, as well as on geometries of unfamiliar type. On the other hand, GAN outcomes tend to *appear* more realistic, and exhibit a lower continuity residual. The qualitative differences were highlighted by considering bifurcation scenarios which were purposefully included in the data set. When the bifurcation parameter is at its critical value, two very different flow scenarios can occur essentially randomly. In such cases, the GAN essentially predicts just one of the possible flow outcomes, whereas the directly trained model outputs a superposition of both. This exemplifies a fundamental difference in prediction behavior. It is shown that these results can be well understood as a direct consequence of the different cost functions used during training. Furthermore, it is demonstrated that by using a sum of both types of loss functions (adversarial & L1), advantages of both models can be combined. In other results, the data show that the discriminator output cannot provide a reliable indication of the accuracy of a prediction, since no robust correlation between them was found. Also, it was observed that the discriminator appeared to dominate the adversarial game, and it was shown that improving this balance could lead to better results. Moreover, an investigation into predicting pressure alongside the velocity field was conducted. Results showed that adding an additional channel for pressure to the network architecture can achieve this goal, but is not necessary, as calculating the pressure field from the velocity output produces results of similar quality. In terms of resources, GAN training required a relative increase in computational time by approx. 50 %. Additionally, GAN models were also found to take many more training steps to reach convergence. Similarly, for a fixed number of steps, results showed that directly trained models can benefit more from larger datasets. In conclusion, GAN models are likely not the right choice for reduced order modeling in scientific contexts due to their lower accuracy. However, they could hold potential for use in creating visual effects or as an added regularizer during training.

Contents

1

Introduction & Motivation

All models of the physical universe are necessarily simplifications. They contain both implicit and explicit assumptions, which are never fulfilled exactly in reality, resulting in behavior that deviates from that of the object or system they are trying to mathematically mimic. These errors may be irrelevant, for example if they are below the scale of measurement accuracy, or they may be grave, and knowing which is the case is generally not trivial. Building models that work, by making the right assumptions for the right problem and following the ensuing logic, has been the pursuit of science over the last centuries, and with great success. Nevertheless, more computationally efficient models are always in demand. Now, new machine learning methods could hold promise to automate the process of model order reduction, as they have successfully done with other tasks previously restricted to human action. Model order reduction refers to techniques for finding models that are computationally simple yet still accurately predict reality under certain limitations.

To illustrate this potential, consider the example of the so called "protein folding problem", i.e. the prediction of a protein's three-dimensional structure from its amino acid sequence. This is a task of enormous relevance for drug design, biotechnology and other applications. The equations governing atoms and molecules are, of course, known to a great degree of precision. Yet still, even after a decades long global research effort, computational models had struggled to solve the problem for large proteins, which are made of of hundreds of amino acids. The reason is that solving the problem based on first principles[1] is prohibitively expensive in terms of necessary computing power. Thus, scientists have had to resort to developing surrogate models, either by simplifying physical laws or by directly inferring from experimental data of known protein structures. But the problem turned out to be more difficult than most, and progress was slow – until 2020, when the DeepMind's AlphaFold 2 AI model managed to achieve results described as "transformational"[5], handily outclassing the previously leading models [56] (see Figure 1.1).

If this type of success can be repeated in other areas, deep learning could establish itself as a useful additional tool for modeling. It would give scientists and engineers

[1] i.e. by solving Schrödinger's equation for all particles in all of the molecules of the protein, plus all those in the vicinity such as surrounding water molecules etc.

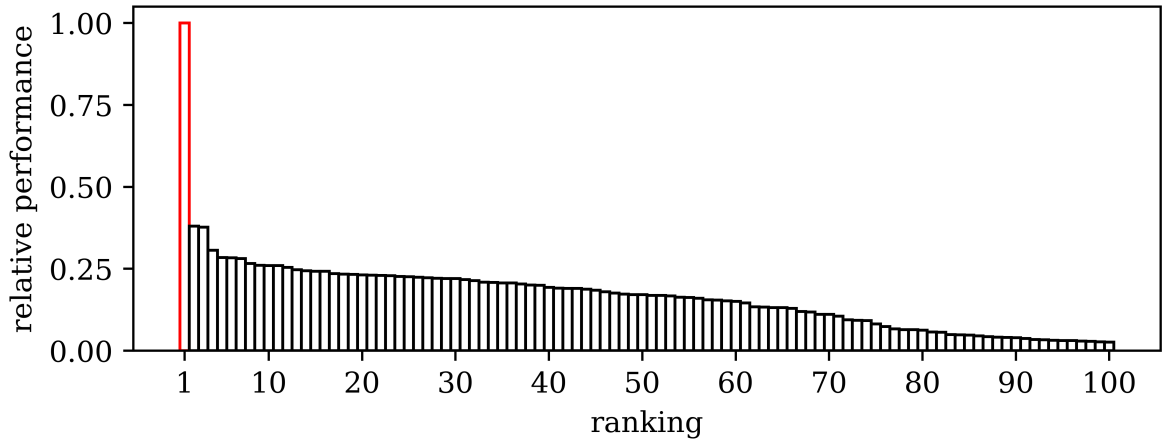


Figure 1.1: Performance of best 100 entries in 14th CASP protein folding challenge in 2020, relative to the scoring of AlphaFold 2. AlphaFold 2 result highlighted in red. Data taken from CASP website [45].

the ability to automatically infer reduced order models for problems both well studied and novel, directly from measurement or simulation data.

In this research project we explore the potential of a specific type of machine learning model for use as a reduced order fluid model, to quickly predict outcomes where normally computational fluid dynamics (CFD) simulations are necessary. Fast and accurate fluid models are crucial for a plethora of technical and scientific applications in fields like energy generation, transportation, geoscience and medicine. If successful, such AI powered reduced order methods could be helpful wherever computational efficiency is of utmost importance, such as in design space exploration or real-time prediction.

The particular model type of interest is the so-called generative adversarial network (GAN). These neural networks offer a completely different, indirect method of learning from data, by training a generator model in an adversarial game where it competes against another neural network acting as a 'critic'. This adversarial training imparts unique properties onto the generator and makes them an interesting object of study.

Although there have been previous studies published on using GANs as reduced order models, they did not clearly identify how the adversarial training specifically affects the model's performance, and whether it produces superior results compared to regular training. Results from GAN models were either not compared to a baseline neural network at all [10, 57, 25, 54], or the baseline model had a different architecture from the GAN generator network [26], which makes it difficult to draw conclusions. This thesis advances the research frontier in this respect by investigating how the results obtained from GANs differ from those produced by identical generator networks trained directly on ground truth, and on how these differences can be explained. We compare the two model types, as well as additional hybrid models, with standard architecture on a wide range of metrics, including accuracy on test set, generalization to other geometry types, and continuity residuals, among others. We also present novel results in this context w.r.t. the correlation between discriminator output and prediction accuracy, as well as on comparing different methods for predicting the pressure field.

This thesis is structured into six chapters, the first being this introduction. The second chapter provides an overview of the most relevant foundations for understanding this work, from outlining the fluid dynamics underlying the problem, to the numerical methods required for producing the training data, up to the basics of the machine learning methods used in the models. The third chapter gives an account of the most pertinent works published in the literature regarding the use of neural networks in general, and GANs in particular, for physical modeling. The following fourth chapter delves into the specific methodology used for this study, describing the process of generating the training and test data, as well as detailing the neural network architectures used. After that, in the fifth chapter, the experimental data are presented and analyzed with respect to a wide range of aspects. Finally, the last chapter draws conclusions based on the results, and offers explanatory frameworks for understanding them as a consequence of the different training methodologies.

2

Fundamentals

2.1. Modeling Incompressible Flow

Gases and liquids together make up a vast share of the matter on earth's surface, and as such, fluids have long been an important subject of study in physics. Today, computational fluid dynamics (CFD) is a mature field, with many highly developed numerical methods at the disposal of scientists and engineers. Some of the most important applications include modeling flow around structures (aviation, transportation, architecture), through turbomachinery (wind power, hydro power, jet engines), flow in the earth system (meteorology, hydrology) as well as through biological systems (e.g. the heart).

In this chapter, we present a brief outline of the mathematical theory of fluids insofar as it is relevant for the content covered in this thesis. We restrict our discussion of the topic to incompressible flow, which is sufficient for modeling liquids as well as gaseous flow so long as flow velocity is small compared to sonic speed. Remarks on the numerical treatment are given in Section 2.2.

2.1.1. Governing Equations

Fluids are generally modeled in the framework of continuum mechanics and described by field quantities (velocity, density, temperature etc.) as a function of space and time. To derive the equations that govern fluids, we simply apply well known conservation principles such as conservation of mass, momentum and energy, and demand that they hold for every part of the fluid. For instance, by assuming that no mass is created or destroyed, we get

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \mathbf{u}\right) = 0, \tag{2.1}$$

where ρ is the mass density of the fluid, and $\mathbf{u}$ is the vector describing its velocity. This is also known as the (mass) continuity equation. For an incompressible fluid, the density along the path of a fluid element is constant. We express this as

$$\frac{D\rho}{Dt} = 0, \tag{2.2}$$

using the so called material derivative notation, which is nothing but the time derivative in the Lagrangian reference frame[1]. In a fixed Eulerian frame, this operator translates to

$$\frac{D}{Dt} := \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla. \tag{2.3}$$

Using this in eq. (2.2), we can substitute into eq. (2.1) to obtain

$$\nabla \cdot \mathbf{u} = 0, \tag{2.4}$$

which implies that volume is conserved.

Next, we apply Newton's second law (or, equivalently, the principle of conservation of momentum) to an infinitesimal fluid element, which, in symbolic tensor notation, yields

$$\rho \frac{D\mathbf{u}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \mathbf{f}. \tag{2.5}$$

Here, the left side corresponds to the inertial force of a fluid element, and the right side to the forces acting upon it. We distinguish between forces arising due to stress within the fluid, which are described by the (2^{nd} order) Cauchy stress tensor $\boldsymbol{\sigma}$, and external, volumetric forces denoted by the vector $\mathbf{f}$, such as gravity. Note here that the divergence of a second order tensor is taken row-wise [33].

To close this system of equations, we need a way of relating the internal stresses to the velocity, or more specifically (when taking into account the principle of relativity) the velocity gradients. This relation cannot be easily derived from first principles of classical mechanics, as it depends on the molecular properties of the specific fluid. However, we may start by trying the simplest possible law and see how far it carries: that stresses be a linear function of velocity gradients. As it turns out, this supposition, first put forward by Newton, holds well for many of the most frequently studied liquids (also known as Newtonian fluids), in particular water and air under normal conditions. If we assume further that the incompressible fluid has isotropic properties, we arrive at the following relation, written in index notation [43, chapter 6]:

$$\sigma_{ij} = p\delta_{ij} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \tag{2.6}$$

where μ is a material constant known as the viscosity (specifically the dynamic viscosity), and δ_{ij} is the Kronecker delta. Here, we have implicitly made use of the fact that the stress tensor is symmetric, a condition that follows from the principle of conservation of angular momentum.

Substituting eq. (2.3) and (2.6) into the momentum equation (2.5), and again making use of the incompressibility constraint, we arrive at the desired momentum equation:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = \frac{1}{\rho} \left(-\nabla p + \mu \, \Delta \mathbf{u} \right) + \mathbf{g}, \tag{2.7}$$

where we have additionally assumed that the fluid has uniform viscosity, and replaced the general volumetric force with the gravitational acceleration $\mathbf{g}$. Together with eq.

[1] In the Lagrangian reference frame, the coordinate system is moving with the fluid parcel, whereas the Eulerian frame uses a coordinate system fixed in space.

(2.4), this constitutes a closed set of equations known as the Navier-Stokes equations for incompressible, uniform fluids.

For this work, we furthermore assume stationary flow, i.e. no changes in velocity over time (steady state). Moreover, we are exclusively modeling a uniform fluid filling out the whole domain, which means that a uniform volumetric force like gravity has no net effect. The Navier-Stokes equations then simplify to

$$\nabla \cdot \mathbf{u} = 0 \tag{2.8a}$$

$$\nu \, \Delta \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u} = \frac{1}{\rho} \nabla p, \tag{2.8b}$$

where $\nu = \rho/\eta$ is the so-called kinematic viscosity. This boundary value problem is a second order, nonlinear system of partial differential equations (PDEs). The first equation represents the condition of divergence free flow, and the second represents the force equilibrium condition for every fluid element. Specifically, the first term on the left hand side of (2.8b) represents the inertial forces acting on the fluid, while the second terms represents the viscous forces, and the right hand side represents an internal source of momentum in the form of the pressure gradient. Mathematically, the pressure can be understood as a Lagrange multiplier enforcing the continuity constraint (2.8a) in the variational formulation of the momentum equation [42].

2.1.2. Non-Dimensional Form

By reformulating the equation in terms of a natural length scale L and velocity scale U, we can non-dimensionalize the left side of (2.8b):

$$\tilde{\Delta}\tilde{\mathbf{u}} - \mathrm{Re}\,(\tilde{\mathbf{u}} \cdot \tilde{\nabla})\tilde{\mathbf{u}} = \frac{L}{\eta U} \tilde{\nabla} p. \tag{2.9}$$

where the tilde ($\tilde{\cdot}$) represents non-dimensional variables and

$$\mathrm{Re} = \frac{UL}{\nu} \tag{2.10}$$

is the Reynolds number that describes the characteristic ratio of inertial to viscous forces. Formulating a natural pressure scale is less obvious, but outside the Stokes regime it is typically taken as ρU^2 [27, page 53-54]. The full dimensionless system of equations then becomes

$$\tilde{\mathbf{u}} = 0 \tag{2.11a}$$

$$\mathrm{Re}^{-1} \tilde{\Delta}\tilde{\mathbf{u}} - (\tilde{\mathbf{u}} \cdot \tilde{\nabla})\tilde{\mathbf{u}} = \tilde{\nabla}\tilde{p}. \tag{2.11b}$$

Therefore we can assume that the flow conditions analyzed here are well characterized by the Reynolds number.

2.1.3. Turbulence Modeling

A particular challenge in computational modeling of fluids is posed by turbulent flow. Turbulence refers to the fleeting chaotic structures that occur on small scales, down to the microscopic (see Kolmogorov microscale [12]), in flows with high Reynolds

numbers. Despite their size, they nevertheless can contribute significantly to transport of quantities such as momentum through the fluid, and thereby affect the macroscopic flow. However due to the small scales involved, correctly reproducing turbulence in a direct numerical simulation (DNS) is computationally infeasible for almost any practical problem due to the extremely fine spatial resolution that would be required.[2] Instead, specialized turbulence models are added to the Navier-Stokes equations, with the goal of modeling not the turbulence itself but its net effect on macroscopic flow. Since the flow conditions considered are in the turbulent range (channel flow with $\mathrm{Re} \approx 10^6$, see Section 4.1), we will rely on such a turbulence model for our simulations.

One framework for modeling turbulence effects are the so-called Reynolds-averaged Navier-Stokes equations (RANS). Reynolds-averaging refers to time-averaging the equation such that the short-lived turbulent fluctuations to approximately equalize. Applying this to (2.8) gives [11, section 10.3.5]

$$\nabla \cdot \bar{\mathbf{u}} = 0 \tag{2.12a}$$

$$\mu \, \Delta \bar{\mathbf{u}} - \rho(\bar{\mathbf{u}} \cdot \nabla)\bar{\mathbf{u}} = \nabla \bar{p} + \nabla \cdot \tau, \tag{2.12b}$$

where the overline ($\bar{\cdot}$) marks a time averaged variable. As one can see, the Reynolds-averaging left the structure of the Navier-Stokes equations mostly unchanged, but a new term $\nabla \cdot \tau$ appeared in the momentum equation. The symmetric 2nd order tensor τ represents

$$\tau = -\rho \overline{\mathbf{u}' \otimes \mathbf{u}'}, \tag{2.13}$$

where $\otimes$ denotes the outer product, and $\mathbf{u}'$ is the instantaneous turbulent velocity deviation from the mean, i.e. $\mathbf{u}' = \mathbf{u} - \bar{\mathbf{u}}$. In analogy to the Cauchy stress tensor, τ is termed the Reynolds stress tensor.

The goal with this approach is to simulate only the averaged quantities. Thus, in order to form a closed system of equations, a turbulence model is required that prescribes the six unknowns in τ as functions of the averaged flow quantities ("closure problem"). Early experimental results lead to Boussinesq's hypothesis that the net effect of turbulence can be modeled as a localized increase in viscosity ("eddy viscosity"). Based on this, the typical ansatz for the Reynolds stress tensor is:

$$\tau_{ij} = \mu_t \left(\frac{\partial \bar{u}_i}{\partial \bar{x}_j} + \frac{\partial \bar{u}_j}{\partial x_i} \right) - \frac{2}{3} \rho \delta_{ij} k. \tag{2.14}$$

The reason for this, at first glance slightly idiosyncratic, form is to conform with the definition turbulent kinetic energy

$$k = \frac{1}{2} \left(\overline{u_1' u_1'} + \overline{u_2' u_2'} + \overline{u_3' u_3'} \right), \tag{2.15}$$

which is used in many turbulence models [11, page 399]. The Boussinesq hypothesis therefore reduces the number of unknowns in τ from six down to two turbulence parameters: the eddy-viscosity μ_t, and k. Within this framework, many different models have been developed to finally close this eddy viscosity model. The most basic among them merely give an algebraic relation between the turbulence parameters

[2] Not to mention that simulating turbulence is not even possible with a stationary approach.

and the time averaged flow fields $\bar{\mathbf{u}}$ and $\bar{p}$, but these are too simplistic to generalize well. Instead, the most common turbulence models introduce additional transport equations, allowing them to take into account convection and diffusion of turbulence. One of the most common and well-validated models is the so-called k-ε model, which introduces the turbulence dissipation rate ε and two additional transport equations to close the system. This is the turbulence model used for simulations in this project. For more details on this specific model, see [17], and for a comprehensive treatise of the theory of turbulence refer to [12].

2.2. Numerical Treatment

In order to derive any use out of the equations derived above, we need a method of solving them for concrete parameters and boundary conditions. This is done using numerical methods, by first discretizing the continuous problem to reduce it to finite dimensionality, and subsequently solve it computationally. Most commonly used discretization methods can categorized as either "finite difference", "finite element" or "finite volume". In the following, we give a brief outline of the latter, which was used for this project, as well as the algorithm used to solve the discretized problem.

2.2.1. Finite Volume Method

The finite volume method (FVM) is often favored for solving of PDEs with underlying conservation laws, as it is inherently conservative (i.e. the PDE's conservation properties are upheld by the discretized system). At its core, FVM is a relatively straightforward solution framework. The domain is discretized into small, typically polyhedral, cells or control volumes (ergo the name), over each of which the PDE is integrated. Consider the following generalized conservation law:

$$\frac{\partial u}{\partial t} + \nabla \cdot \mathbf{f}(u) = 0 \tag{2.16}$$

with conserved variable u and a flux term $\mathbf{f}$. After integration, we have

$$V_i \frac{\mathrm{d}\overline{u}_i}{\mathrm{d}t} + \int_{C_i} \nabla \cdot \mathbf{f}(u)\, \mathrm{d}V = 0, \tag{2.17}$$

where $\overline{u}_i$ is the average of u on cell C_i with volume V_i. Note that here we have assumed that the cells are fixed in time. Instead of solving for u itself, we are thus solving for its average on the cell volumes. Next, we can transform the divergence term to fluxes integrated across the cell boundaries $\mathcal{S}_i$ using Gauss's theorem:

$$V_i \frac{\mathrm{d}\overline{u}_i}{\mathrm{d}t} + \oint_{\mathcal{S}_i} \mathbf{n} \cdot \mathbf{f}(u)\, \mathrm{d}V = 0, \tag{2.18}$$

with surface normal vector $\mathbf{n}$ pointing outside of the cell. The fluxes are central to the method, and they are evaluated in a zero-sum way, i.e. the flux leaving one cell through a shared boundary is the same as the one entering its neighbor. This way the conservative property is ensured.

The challenge that remains is to formulate an expression describing the fluxes through the boundary between two neighboring cells as a function of the average field

quantities inside those cells. One way to do this is to treat it similar to a finite difference problem by taking the average values as defined at the centers of each cell, and using them to evaluate discrete derivatives. Other approaches have also been developed, such as Godunov's scheme [55] which employs the technique of characteristics. For a more detailed treatise of the finite volume method, see [38].

2.2.2. Solver

The systems of equations posed by fluid problems, even stationary ones, are typically solved using iterative methods. For our purposes we used the fractional step method known as "Semi-Implicit Method for Pressure-Linked Equations" (SIMPLE), first proposed in [44]. Fractional step refers to the fact that the field quantities $\mathbf{u}$ and p are solved for in alternation. Specifically, in every time step at first the momentum equation is solved using the pressure field of the previous step. Then, the new pressure field is computed by solving a pressure correction equation, and the result is used to also update the velocity field. Finally, the remaining transport equations (in this case from the turbulence model) are solved using the newly computed fields. For details, refer to [17, Sections 6.4 & 6.5].

Although from an analytical perspective, the steady state problem is quite different from the transient one, the numerical treatment is often not too dissimilar. The iterative solving bears many similarities to time-stepping in a transient problem. For steady-state solving, implicit update methods are typically used (as in SIMPLE), since they allow for larger step sizes without loosing stability. In every iteration of the SIMPLE algorithm, first the momentum equation is solved using the pressure field from the previous iteration, in order to obtain an intermediate velocity field. Subsequently, correction steps are applied to both pressure and velocity, in order to make the velocity field conform to the continuity equation. For more details, refer to [11, section 7.2]. If an explicit turbulence model is used, additional equations must be solved simultaneously in order to obtain the Reynolds stress tensor, as discussed in Section 2.1.3.

2.3. Deep Learning

Enabled by the culmination of sustained exponential growth in computing power over the last decades [52] , the field of machine learning (ML) has made remarkable progress over the last decade. The term refers to techniques of enabling computers to solve problems, not by directly following a set of rules encoded by a programmer, but by inferring those rules from observation, i.e. data. The vast majority of contemporary artificial intelligence (AI) systems are based on some form of machine learning. Most of these make use of the concept of hierarchical representation, using consecutive layers of information processing units that build on each other to bootstrap more powerful representations of real world data. This paradigm is the origin of the term deep learning. [14, page 5]

In recent years, ground-breaking results have been achieved across fields such as computer vision, speech and natural language processing & synthesis [1].

Beyond being a very active field of study itself, ML has also been successfully applied to advance scientific frontiers in other areas of research. In particular, as discussed in the introduction, ML has been explored as an alternative to conventional

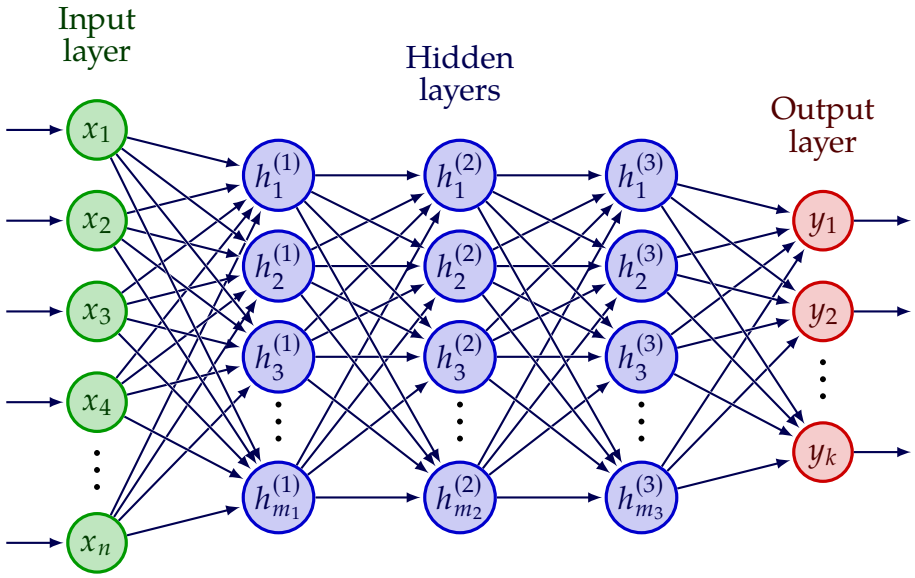


Figure 2.1: A graph visualizing the structure of a standard artificial neural network with three hidden layers. Each node corresponds to a neuron and each arrow to a weight.

methods from computational science, a discipline referred to as scientific machine learning (SciML). For a comprehensive overview of the field, see [2]. A survey of published work on applying ML to PDE problems is given in Chapter 3.

Machine learning methods can be separated into two categories, supervised and unsupervised learning. The key difference is that in supervised learning, each training example has a label, and the goal is to map from unseen examples to the correct label. With unsupervised learning on the other hand, data are unlabeled, and the goal is generally to, in some form, learn the probability distribution underlying the dataset. [14, page 105]. The focus of this work is on a specific type of machine learning framework known as generative adversarial networks (GANs) which can combine aspects of both paradigms (see Section 2.3.3).

2.3.1. Artificial Neural Networks

An artificial neuron network (ANN, hereafter also simply referred to as neural network, NN) is a type of machine learning architecture designed in loose analogy to the networks formed by biological neurons found in the brains of humans and animals. As the name suggests, ANNs are made up of individual information processing units called (artificial) neurons, which are arranged in connected layers. An example is visualized in figure 2.1. The principal setting of supervised learning is to model the relation underlying a set of observations and labels i.e. to find a way of relating a feature to its proper label. We generally distinguish between regression tasks, where the label is continuous (e.g. predicting the market value of a house), and classification tasks, where it is discrete (e.g. recognizing a handwritten digit).

Artificial Neurons

A single neuron can be represented as a mapping from a number of inputs a_i to a single output h known as its activation level. Specifically, this takes the form of a weighted sum of all inputs, which is passed through a so-called activation function $f : \mathbb{R} \to \mathbb{R}$

after a fixed offset known as the bias has been added. We collect the inputs in a vector $\mathbf{a}$ and the corresponding weights in a vector $\mathbf{w}$. Together with the bias b we can thus write

$$h = f\left(\mathbf{w} \cdot \mathbf{a} + b\right). \tag{2.19}$$

The weights and bias parameterize a hyperplane in the input space known as the decision boundary, which is defined by $\mathbf{w} \cdot \mathbf{a} + b = 0$. A geometric interpretation of the operation performed by a single neuron is that it computes the euclidean distance d of the input point $\mathbf{a}$ to the decision boundary, scales it by the norm of the weight vector and applies the activation function to the result, i.e.

$$h = f\left(d\,|\mathbf{w}|\right). \tag{2.20}$$

Network Structure

As mentioned above, the neurons in an ANN are arranged in layers. These are typically connected in a sequential order, such that information propagates uniformly from input to output, although other variations exist such as the U-Net architecture used in the models considered here (see Section 2.3.2 & Figure 4.4) or recurrent neural networks (RNNs). Each layer defines an operation acting on the output of the previous. Data is always fed into the network at the input layer, in analogy to, for instance, sensory organs in biological cognition. Then it passes through a number of so-called hidden layers, until it reaches the output layer where the response of the network is read off. While the widths of input and output layers (n and k respectively in Figure 2.1) are imposed by the problem setting, the number and size of hidden layers is a design parameter. If we represent each layer (including the input) as a vector, then the operation that a layer j performs on the output of the previous layer i can be written as

$$\mathbf{h^{(j)}} = \mathbf{F}\left(\mathbf{W}_j\,\mathbf{h^{(i)}} + \mathbf{b}\right), \tag{2.21}$$

where $\mathbf{h}$ represents a vector containing all activations in that layer, $\mathbf{W}_j$ is an matrix of dimensions $m_j \times m_i$ (i.e. the sizes of layers i and j respectively), $\mathbf{b}$ is a vector containing the biases and $\mathbf{F}$ is the element wise application of the activation function f. If $\mathbf{W}_j$ is a dense matrix, we say layer j is fully connected. The connectivity structure is an important design parameter, and is explored further in Section 2.3.2.

Looking at the network as a whole, we simply have a parameterized function $\mathcal{N}$ mapping from an input space X to an output space Y:

$$\begin{aligned} &\mathcal{N} : X \to Y \\ &\mathbf{y} = \mathcal{N}(\mathbf{x}; \boldsymbol{\omega}), \end{aligned} \tag{2.22}$$

where $\boldsymbol{\omega}$ represents all weights and biases in the network, and $\mathbf{y}$ is the network's output. Typically, both input and output are taken as vectors in $\mathbb{R}$. However, if the data has a grid-like 2D or 3D structure it can make sense to reflect that in the mathematical representation of the network. Note that size of input and output layers may be drastically different; take for instance the task of classifying 256×256 pixel images on the basis of whether or not they depict a cat. In this case, the network will have an input layer with a size of approx. 65 000, but only a single, binary output neuron.

Training

A large neural network can easily have millions of trainable parameters, with the biggest containing over 100 billion (for instance GPT3 language model, see [4]). The goal is to choose their values in such a way that they represent the structures underlying the training data and solve the task at hand. In the case of supervised learning, the task is to approximate the unknown law between observations and labels. This law could be a simple functional relation, $\mathbf{y}(\mathbf{x})$, but is typically instead modeled in the more general framework of stochastics to account for randomness and noise, i.e. as conditional distribution $p(\mathbf{y}|\mathbf{x})$.

We want to approach the problem empirically, i.e. have the ANN learn from training examples. Our training set S consists of N features $\mathbf{x} \in X$ and associated labels $\mathbf{y} \in Y$, drawn from the probability distribution $p(\mathbf{x}, \mathbf{y})$. As described in eq. (2.22), the ANN can generate an output $\mathbf{y}$ for each feature sample $\mathbf{x}$. Initially, weights are typically randomly initialized based on some heuristic. Obviously, such an untrained network cannot be expected to solve a given task. In order to improve then, we first need a measure of how good (or bad) the network is performing. This is provided by the cost (or loss) function $c : Y \times Y \to \mathbb{R}$, which assigns a cost to every combination of y and $\hat{y}$. The ultimate goal of the learning process is to minimize the expected cost C given the underlying data distribution. The optimization problem is therefore

$$\min_{\boldsymbol{\omega}} C = \min_{\boldsymbol{\omega}} \mathbb{E}_{p(\mathbf{x},\mathbf{y})} \left[c\left(\mathbf{y}, \mathbf{y}\right) \right]. \tag{2.23}$$

However, in practice of course we do not know $p(\mathbf{x}, \mathbf{y})$ exactly (otherwise we would already have solved the problem). Therefore we approximate C by the average training error C_T on our training set

$$\min_{\boldsymbol{\omega}} C_\mathrm{T} = \min_{\boldsymbol{\omega}} \frac{1}{N} \sum_{S} c(\mathbf{y}, \mathbf{y}). \tag{2.24}$$

This is not necessarily a good approximation, in fact it becomes essentially useless in the case of overfitting (see [14, section 5.2]). To avoid this, sometimes additional so-called regularization terms are added to the cost function (see [14, chapter 7]).

The optimization problem (2.24) is typically highly non-convex, and solutions are by no means unique. In fact, just by permutation of neurons, a single fully connected layer of size L multiplicatively contributes $L!$ equivalent solutions. At any rate, using direct methods for finding the optimum is infeasible due to size and complexity of the problem. Instead, iterative gradient based methods are typically used. The simplest, known as gradient descent, works by computing the derivative of the training error with respect to every trainable parameter, and use that information to update them all at once. This is equivalent to taking a step in the opposite direction of the gradient of the training error (in parameter space). This guarantees that weights are updated in the (locally) most optimal way. Using Einstein notation, we can formulate the update for the i^{th} weight as

$$\Delta\boldsymbol{\omega}_i = -\eta \left.\frac{\partial C_\mathrm{T}}{\partial \boldsymbol{\omega}_i}\right|_{\boldsymbol{\omega}_0} = -\eta \left.\left(\frac{\partial C_\mathrm{T}}{\partial \mathbf{y}_j} \frac{\partial \mathcal{N}_j}{\partial \boldsymbol{\omega}_i}\right)\right|_{\boldsymbol{\omega}_0}, \tag{2.25}$$

where ω_0 are the current model parameters and $\eta \in \mathbb{R}$ is a parameter scaling the update step, also known as learning rate. The learning rate is typically adapted as training progresses to allow for more fine-grain optimization closer to the optimum.

A limitation of gradient descent is, that it is by no means guaranteed to converge to a global optimum. Instead, it will often get stuck in a local minimum, which exactly depending on the random initialization. For relatively shallow minima, this can be overcome (literally) by adding "momentum" terms to the descent kinetics (see [14, section 8.3]), but the core issue remains. Nevertheless, gradient descent has been proven in practice to be a very successful workhorse of neural network optimization. On this issue, the authors remark in [14, page 153]:

> In the past, the application of gradient descent to non-convex optimization problems was regarded as foolhardy or unprincipled. Today, we know that the machine learning models [...] work very well when trained with gradient descent. The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful.

Part of the reason why the method performs well in practice despite the limitations is down to the fact that we are not actually interested in finding global or even local minima of (2.24), as this solution would most likely not correspond to a good solution of the original problem (2.23) due to overfitting (see again [14, section 5.2]).

Fortunately, the gradient can be evaluated relatively easily using the back-propagation algorithm (or, more generally, automatic differentiation), at roughly the same computational complexity as evaluating the network's output in the first place. So far we have discussed the case of using the whole dataset for each weight update, a procedure known as batch gradient descent. Instead, one may also only use a subset ("mini-batch") or even just a single sample, known as stochastic gradient descent (SGD).

As alluded to above, using the gradient as the direction of the update is only optimal for an infinitesimal step size. In practice of course, we do not want to choose η too small in order to keep the number of iterations to an acceptable level. However, this introduces higher order errors that can lead to a very suboptimal optimization path. The magnitude of these higher order terms essentially depends on the product of activations across the layers. Therefore it is desirable that activations are generally small in magnitude. This is achieved elegantly by the so-called batch normalization method. It ensures that across a batch (or mini-batch), the distribution of each activation has zero mean and unit standard deviation [21].

While initially sigmoid functions were a popular choice of activation function in the field of machine learning, so-called rectified linear unit functions have become the standard for modern deep networks [14, page 174]. This is because they avoid the problem of "stretching out" the loss landscape w.r.t. parameters in shallow layers[3], while retaining a non-linearity. The general formula for the rectified linear unit is

$$f(z) = \max(z, az), \ a \in [0, 1], \tag{2.26}$$

[3]which is caused by the vanishing gradient of sigmoid functions for inputs of large magnitude

which is a modification of the original rectified linear unit with $a = 0$ (ReLU, [40]). If $0 < a \ll 1$, it is also referred to as leaky ReLU [60].

For more information on optimization techniques in neural networks, refer to [14, chapter 8]

2.3.2. Convolutional Networks

Much of the data encountered in real world problems has tensorial structure, such images or time series. A common way to reflect this in the structure of an ANN is through use of convolutional layers, which imparts useful priors onto the model that can greatly improve performance and lower the computational cost of training.

Convolutional Layers

Digital image or signal processing often employs discrete convolutions for feature detection. Here, the feature is encoded in a small filter kernel or stencil, which is shifted across the image or signal (in the following we focus on application to images). At each location, the data in the kernel range is multiplied by the corresponding kernel elements and summed up. The resulting grid of values is called a feature map, as it indicates the presence of the feature in the original image. A simple example is edge detection.

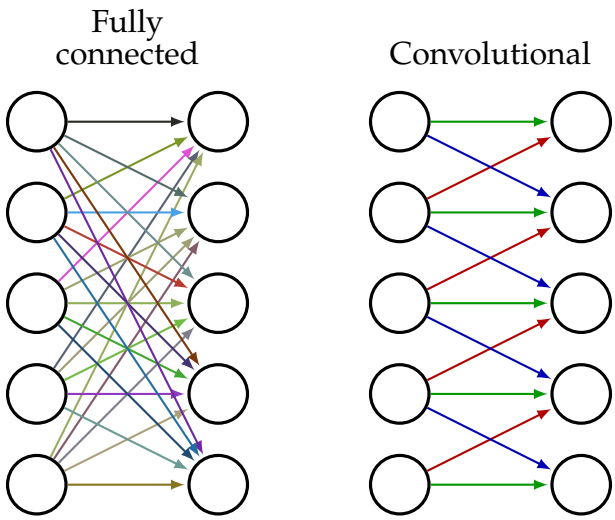


Figure 2.2: Comparison between fully connected and convolutional layer. Same colors indicate shared weights; the fully connected layer has 25 independent weights, while the convolutional layer has only 3.

The same idea is used in convolutional layers, except now the features to search for are determined as part of the learning process. In the regular ANN framework, this corresponds to a sparsely connected layer (each neuron is connected only to its neighborhood), where additionally weights are shared between all neurons in the layer. The number of parameters therefore only depends on kernel size, not on the number of nodes in the layer. This drastically reduces the parameter space, as shown in Figure 2.2. However, we typically apply not just one but many kernels to a single layer, and thus we get multiple feature maps as output, which are also known as channels. For the case of a 2D image, we can therefore represent a convolutional layer as a 3D block, where

each sublayer in the block represents a different feature map. If another convolutional layer follows, it will act on the whole block, i.e. each kernel will have not just a width and height, but also a depth equal to that of the previous block. If we represent feature maps as vectors, we can write the operation performed by a convolutional layer as a sum of matrix vector multiplications:

$$\mathbf{y}_i = \sum_j \mathbf{C}_{ij}\mathbf{x}_j, \tag{2.27}$$

where $\{\mathbf{x}_i\}_{i=1...I}$ are the I input channels, $\{\mathbf{y}_i\}_{i=1...J}$ are the J output channels and $\{\mathbf{C}_{ij}\}$ are the weight matrices defining the convolution. Each $\mathbf{C}_{ij}$ has the same sparsely diagonal structure, where each row of a matrix contains the same kernel elements but shifted. Not by coincidence, this structure is very similar to that of matrices resulting from finite element discretizations of PDE problems (here the kernel corresponds to the stencil). If $\mathbf{C}_{ij}$ is a square matrix, then the input and output feature maps are of the same size, corresponding to the kernel being evaluated at every point of the input grid. Often however, the kernel is only evaluated at larger regularly spaced intervals, a method known as striding. This corresponds to removing rows from $\mathbf{C}_{ij}$, and effectively downsamples the input. For the image example, a stride of two in both directions reduces the rows in $\mathbf{C}_{ij}$ and thereby the length of $\mathbf{y}_i$ by a factor of four (the square of the stride). Moreover, there are different strategies for dealing with the points on the boundary, where the kernel extends beyond the input data (e.g. padding the data with zeros). For more details, see [7]. Another operation that is often combined with convolutions is so-called pooling. In a pooling operation, each pixel in the feature map is replaced with the average (mean pooling) or maximum (max pooling) of the values in its neighborhood. For a more in-depth treatise on convolutional layers and their use, refer to [14, Chapter 15].

Convolutional Architecture

Neural networks that make use of convolutional layers are called convolutional neural networks (CNNs). A common feature of CNN architecture is the repeated use of downsampling convolutional layers, combined with a simultaneous increase in number of channels. The idea here is that each layer can assemble higher level features based on the information in the channels of the previous layer. The striding also causes the receptive field to increase from layer to layer, allowing deeper layer to detect features much larger than its kernel size. The receptive field of a kernel is made up by all pixels in the input image which influence a given pixel in the feature map.

Consider again the example of classifying images based on whether or not they depict a cat. The first layers might detect basic features such as edges and circles, while later layers can use this information to detect more complex features such as a snout or tail, and finally a whole cat.

For tasks where both input and output have tensorial structure, CNNs commonly use a bottleneck architecture, similar to an autoencoder, as visualized in Figure 2.3. It consists of an encoding pipeline, which "featureizes" data, a bottleneck combining the high level features in latent space, and a decoder that synthesizes the output using transpose convolutions, also known as up-convolutions (and sometimes misleadingly referred to as deconvolutions). Transpose convolutions are convolutions where striding

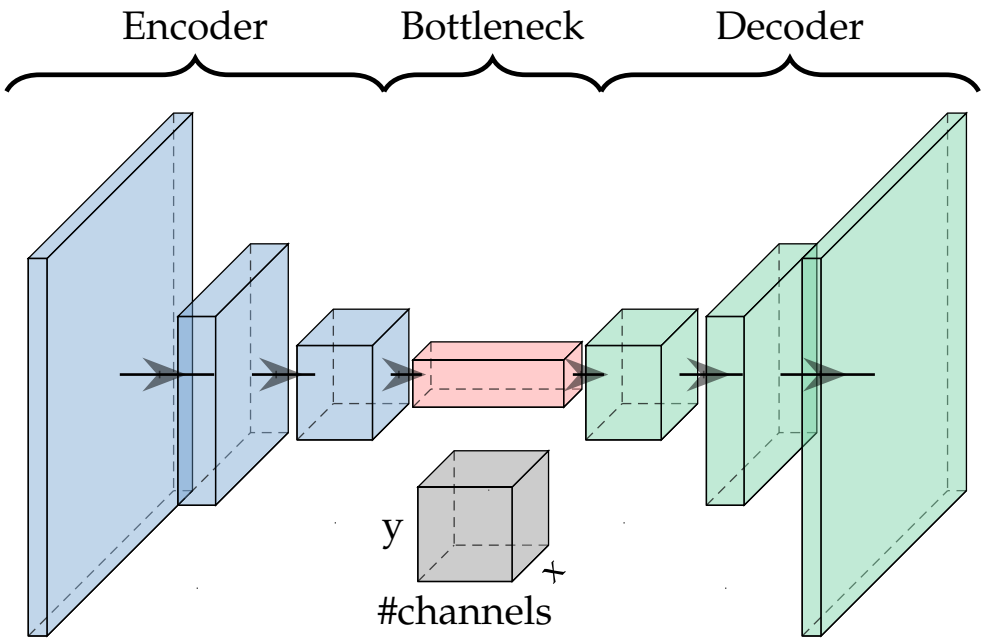


Figure 2.3: Convolutional bottleneck architecture (schematically). Visualization based on [22].

is applied at the output instead of the input. The name stems from the fact that if the weight matrix in (2.27) defines a down-convolution, then an up-convolution is defined by its transpose (see also [7]).

Some architectures, such as the "U-Net" introduced by Ronneberger, Fischer, and Brox in [51] also introduce direct, so-called skip connections between down-convolution and up-convolution layers on the same level. This can help if input and output share a lot of low-level structure. In particular this is true of predicting a flow field from geometry (the boundaries are visible in both input and output), wherefore this type of architecture is used for the models tested in this study. It also bears resemblance to the multigrid method from scientific computing, as explored in [32].

2.3.3. Generative Adversarial Networks

As deep learning was already achieving impressive results for discriminative problems (regression & classification), the field initially struggled to repeat the same for generative tasks. An important step in this regard was the introduction of the framework of adversarial networks in 2014 by Goodfellow et al. [15]. Since then, research on GANs, both theoretical and practical, has grown to a sizable field within machine learning, and across a wide range of generative tasks GANs have achieved state-of-the-art results. These include synthesis of hyperrealistic human faces [28, `thispersondoesnotexist.com`], text-to-image translation [19], image-to-image translation [24, 61] as well as multimodal combinations of these [20], among many others.

Nevertheless, it need not be withheld that alternative models for generative tasks have also been developed and show some promise, most notably variational autoencoders [29], flow-based models [50] and diffusion models [6].

Motivation and Concepts

Before any neural network can be trained, one has to lay out the "grading scheme" that is the cost function. For some types of problems, such as classification, the choice is typically straightforward. Take again the task of detecting whether an image contains a cat. Given a set of labeled training data, we penalize incorrect guesses by the network, taking into account its level of confidence with the cross-entropy cost function. An arguably much more difficult task, however, is to *generate* realistic images of cats. The key difference is that defining a cost function that properly accounts for the target distribution (i.e. what exactly does it mean for something to "look like a cat") is difficult.

GANs are designed to solve this problem in an elegant way. Instead of having to hand-craft the cost function that assesses the performance of the generator, it becomes part of the training process. This is done by training a second ANN, known as the discriminator, to classify images based on whether they are from the training set or merely a creation of the generating network. Generator and discriminator are trained together in an adversarial zero sum game, where the former is trained to 'fool' the latter, while the latter is trained to expose the 'forgeries' of the former.

Generative Problems

The canonical generative setting in which GANs are applied is finding a useful mapping from a latent space input $\mathbf{z}$ to the probability distribution $p(\mathbf{x})$ underlying the training data. This is a purely unsupervised task. A classical example is generating novel images of ostensibly human faces. However, GANs can also easily be extended to work with conditional distributions $p(\mathbf{x}|\mathbf{c})$ with conditional $\mathbf{c}$ (then also known as cGAN), which is a semi-supervised setting. For the example of facial generation, $\mathbf{c}$ could be as simple as an integer value representing age, or as complex as a picture of a face whose appearance is to be artificially aged or youthened.

Conditional generative problems fall on a spectrum. On the one side there are settings where we are actively interested in sampling the data distribution through the latent space, such as when building a facial generator. On the other side, we have more translative problems, which may or may not be strictly deterministic, but for which we are typically only interested in getting a single, high-quality result. Examples would be image upscaling [34] or generating a city map from satellite imagery [24]. In such cases, the latent input is superfluous and may be omitted.

For this study, we trained GAN models to perform a mapping from an image encoding geometry to one showing the resulting flow field. This task falls into the categorize of translative problems. That being said, there is stochasticity introduced by the numerical solution, which in some particular cases (bifurcation) can be very significant (see Section 4.1).

Foundations and Training

As described above, training a GAN can be understood as a game where the two 'players' (generating and discriminating ANNs) are competing against each other. The generator G defines a mapping from the latent space to the space of training samples, given a conditional. The discriminator on the other hand takes in a sample, together with the corresponding conditional, and outputs a value corresponding to its

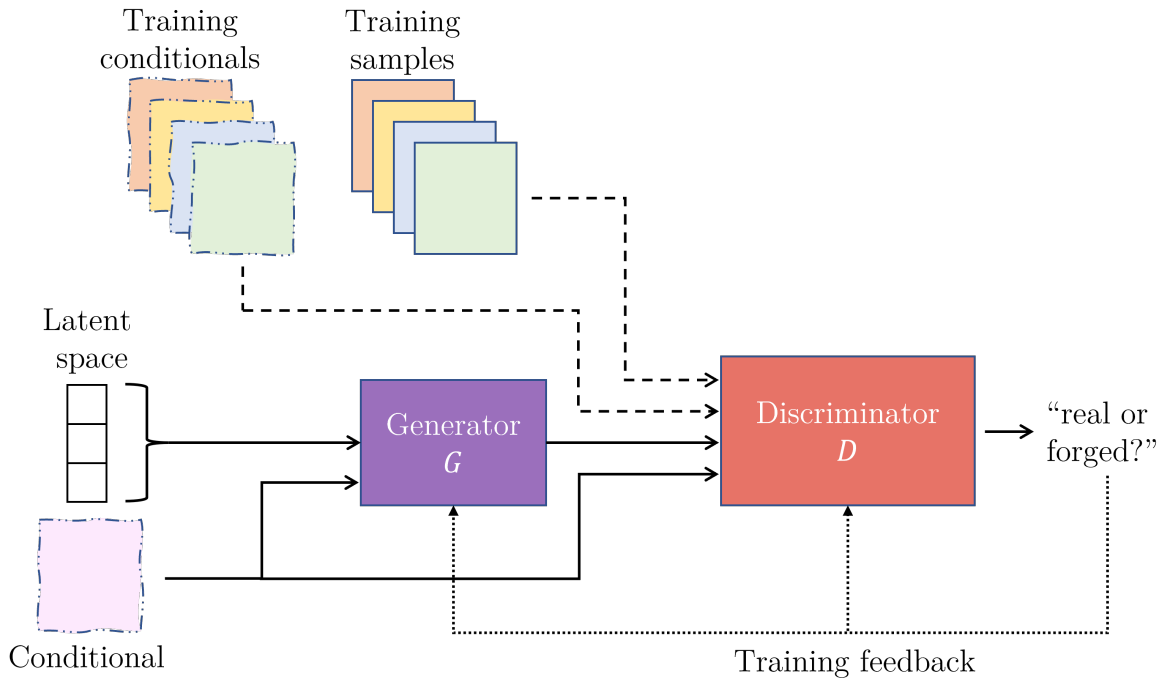


Figure 2.4: Chart schematically illustrating the information flow through a conditional GAN.

confidence in the input being 'real', i.e. from the training set as opposed to created from the generator. Figure 2.4 illustrates the flow of information through the combined network. Mathematically, we have

$$G(\mathbf{z}, \mathbf{c}; \boldsymbol{\omega}_G) \tag{2.28a}$$

$$D(\mathbf{x}, \mathbf{c}; \boldsymbol{\omega}_D) \in [0, 1], \tag{2.28b}$$

where $\boldsymbol{\omega}_G$ and $\boldsymbol{\omega}_D$ are the trainable parameters parameterizing the generator and discriminator respectively. The adversarial minimax game can be cast as the optimization problem

$$\arg\min_G \max_D V(G, D), \tag{2.29}$$

of the objective function V given by

$$V(G, D) = \mathbb{E}_{p(\mathbf{x},\mathbf{c})}\left[\log D(\mathbf{x}, \mathbf{c})\right] + \mathbb{E}_{p(\mathbf{z},\mathbf{c})}\left[\log\left(1 - D\left(G(\mathbf{z}, \mathbf{c})\right)\right)\right]. \tag{2.30}$$

Here cross entropy was used as a measure of discriminator error. For the generator then, the cost function is $C = V$, whereas for the discriminator we have $C = -V$. Using these cost functions, the model can be optimized by updating generator and discriminator in alternation. For such an algorithm, it was shown in the original paper [15] that given enough model capacity and training, G will eventually learn to perfectly mimic the underlying data distribution, i.e

$$p\left(G(\mathbf{z}, \mathbf{c})|\mathbf{c}\right) p(\mathbf{z}) = p(\mathbf{x}|\mathbf{c}) \tag{2.31}$$

At that point, an optimal discriminator will reach maximum uncertainty, i.e. $D(\mathbf{x}, \mathbf{c}) = 0.5$. This corresponds to a global optimum of (2.29), and a Nash equilibrium[4] of the

[4] A Nash equilibrium is a concept from game theory, where no player has something to gain from changing their behavior.

two-player game.

GAN Architectures & Challenges

While GANs are conceptually compelling, they have proven to be notoriously unstable during training. The learning progress of generator and discriminator should be well balanced in order to avoid one of the networks dominating the other, which can lead to very slow convergence overall. Another common undesirable effect is known as mode collapse, a phenomenon where the generator becomes stuck in a state of producing only a very narrow range of outputs [8].

In particular, there were initial difficulties making the GAN concept work with convolutional architectures. One of the earliest successful attempts at doing so used a very specific family of architectures, named by the authors deep convolutional GAN (DCGAN) [46]. After extensive exploration, the authors identified three architecture features that helped improve training stability, which have been influential in the further development of GAN architectures:

- Replace all pooling operations with strided convolutions.
- Remove fully connected layers.
- Use batch normalization.

Moreover, the authors found the ReLU activation function to yield the best results in the hidden layers of the generator, while for the discriminator leaky ReLU was found to work best.

Some further suggestions for improvement were presented in [53], including modifications to the cost functions and methods to help the discriminator identify unwanted modal collapse.

3

Related Work

3.1. Machine Learning in Scientific Computing

The use of neural networks in finding approximate solutions to problems with an underlying PDE is an emerging field at the intersection of machine learning and scientific computing. Research activity in this area can be roughly split into the following three groups, although hybrids and variations exist.

The first group encompasses techniques that use machine learning to improve upon classical numerical methods. Published work on this includes areas such as RNN assisted solution upscaling for multigrid schemes [37], learning based preconditioning for domain decomposition methods [18] and use of neural networks for obtaining closure terms in turbulence modeling [3].

The second contains methods that seek to approximate the PDE solution directly in the parametrized function space given by the neural network, as already proposed by Lagaris et al. in 1997 [31]. In other words, the ANN itself serves as the discretization, in the sense of reducing the original problem to finite dimensionality. Perhaps the purest implementation of this approach are the physics-informed neural networks (PINNs) as introduced by Raissi et al [47, 48]. In this framework, the solution $u(x)$ is directly approximated by the neural network $\mathcal{N}(x;\omega)$.

The work presented here belongs to a third group of using neural networks as reduced order surrogate models. Instead of training a network to solve a specific problem as in the previous group, here we train a network in order to obtain a model which can then be applied to solve a whole range of problems (e.g. with different geometries, boundary conditions etc.). These neural network models operate on existing discretizations. They are typically trained to map from a raster image of the domain, together with information about boundary / initial conditions and possibly other parameters, onto one or multiple images showing the predicted solution fields. Due to the regular grid-like structure of inputs and outputs, these methods typically rely on a convolutional architecture. Examples include using neural networks to build reduced models for steady state flow [16, 9].

As is always the case with reduced order models, we want to sacrifice some generality of our model for easier evaluation. The motivation for using ANNs in constructing the surrogate model is to be able to infer the essential relations in the relevant problem range directly from data. To draw an analogy, the learning process

could be akin to how humans can build up a physical intuition without ever studying physics. It could also reduce the need for hand-crafted domain specific reduced order models built using assumptions based on expert knowledge, which are commonplace in some domains.

Increased computational efficiency is always desirable, but for many applications it is crucial. For instance it can help with design space exploration, where it can enable scanning a large number of configurations for optimality, or even providing design feedback to an engineer in real-time.

3.2. GANs as Surrogate Models for PDEs

Since the GAN framework was first proposed in 2014 [15], numerous studies have looked at applying it to modeling problems. They typically fall into the third group as described above, i.e. they are surrogate models operating on existing discretizations.

One of the earliest attempts was by Farimani et al. in 2017 [10]. The authors trained GANs to solve 2D boundary value problems, specifically Laplace's equation and the incompressible steady-state Navier-Stokes equations, see eq. (2.8). The generator was tasked with mapping from an image encoding the domain and boundary conditions, which is supplied as conditional input, to an output image showing the respective solution field. The generator loss function was a combination of the discriminator loss and an L1 loss with respect to the ground truth. For the case of Navier-Stokes, there are three output channels, one for each of the variables (velocity components u,v and pressure p). The discriminator operates not on the whole image, but on smaller patches (patchGAN, see section 4.2). The authors were able to obtain high accuracy on a test set with a relative mean absolute error (MAE) of less than 1%, and claimed that the neural network model outperforms state-of-the-art finite difference solvers in terms of prediction speed by an order of magnitude. The authors did not investigate how much the adversarial part of the loss function actually improved results compared to a direct L1 ground-truth based training.

Multiple studies have since applied GAN models to solve fluid problems in particular. In a 2020 paper, the authors used a GAN for predicting time series of convective flow with energy transport in a 2D square domain from initial and boundary conditions [25]. The method was found to provide fast and accurate solutions for the analyzed test cases. In another recent publication from 2021, the authors applied a GAN setup to model stationary flow through a more complex 3D domain of dispersed spherical obstacles, a relevant setting for modeling certain multiphase flow [57]. The authors found the GAN based result to outperform an older reduced order model that was developed specifically for this application, but did not comment on the relative computational effort.

Others have implemented GAN models for predicting stress in solids on a 2D domain with complex geometry [26]. In their 2021 publication, the authors supply separate images encoding domain geometry, loads and boundary conditions as input and extract a single output image of the domain showing von Mises stress. The architecture was largely based on the same model as used in for this study (pix2pix, see Section 4.2). The authors found the GAN to consistently outperform a previous purpose built baseline CNN model. However, since the two models used different generator architectures it remained unclear to what degree the adversarial training

was actually responsible for the improvement.

While all the work mentioned so far relies on purely data-driven approaches, other authors have investigated the effects of incorporating physics constraints into the training process. In one 2019 publication, a GAN was trained to predict how a given flow field around a cylinder would have evolved after a certain time step into the future [35]. The authors compared four variants: GANs as well as regular CNNs, one of each trained with and without a physics based loss contribution (conservation of mass and momentum). While all versions offered some success for prediction in unseen flow regimes, the GAN trained without physics based loss was found most successful at predicting recursively multiple time steps into the future.

There have already been more application-related studies published using GANs for fluid modeling as well. An architecture firm used a GAN implementation (pix2pix, similar to what was used here, see Section 4.2) to predict wind speeds in urban settings based on a building height map, as published in a conference paper in 2020 [54]. Again they trained the generator both on the discriminator and on the ground truth loss. Generalization performance was mixed, but the authors did not benchmark it against a comparable NN or reduced order model.

In conclusion, while there have been a number of publications giving a 'proof-of-concept' for using GANs in physical modeling, high-quality studies focused on comparing with an equivalent neural network model that is trained directly on ground truth are still missing as of yet.

4

Methodology

As laid out in the introduction, the goal of this work is to investigate and compare the potential for training GANs as reduced order models for predicting fluid flow. For this purpose, a popular GAN model was adapted from literature, and trained to map from an image of 2D geometry to images showing the resultant 2D flow field. The training and test data were generated by means of a standard numerical simulation. In this chapter, we briefly describe the specific methodology for generating this data, as well give an overview of the neural network architecture used for the project.

4.1. Flowfield Generation

The first step in creating any such training set is to design an algorithm that automatically generates a desired number of different problem geometries from a certain distribution. For this study, we trained on a set of geometries containing a 2D channel flow through a sudden expansion. Additionally, the larger section is partially blocked by an obstacle. A sketch of an exemplary problem domain is shown in Figure 4.1. The outer perimeter is parameterized by just two quantities, the lower and upper shoulder lengths S_1, S_2. These are randomly chosen by drawing two values $\{v_1, v_2\}$ from the uniform distribution $\mathcal{U}(0, D)$ where $v_1 \leq v_2$, and setting $S_1 = v_1$ and $S_2 = D - v_2$. Together with the fixed parameters $D = 1\text{m}$, $L = 3\text{m}$, $l = 0.5\text{m}$, this fully constrains the perimeter. For the velocity we impose a Dirichlet boundary condition at the inlet ($\mathbf{u} = U\hat{\mathbf{e}}_x$) and walls ($\mathbf{u} = \mathbf{0}$), as well as a homogeneous Neumann condition at the outlet ($\partial_x \mathbf{u} = \mathbf{0}$). As inlet speed we choose $U = 3\ \text{ms}^{-1}$. The kinematic viscosity was set to $\nu = 10^{-5}\ \text{m}^2\text{s}^{-1}$. If we take U as the characteristic velocity and D as the characteristic length, then the Reynolds number evaluates to $\text{Re} = 10^6$. Since the critical Reynolds number for pipes is on the order of 2×10^4, we can safely assume turbulent flow. For calculating the pressure from the simulation data, we use a density value of $\rho = 1000\,\text{kg}\,\text{m}^{-3}$. These values roughly correspond to the properties of liquid water at room temperature.

The sudden expansion scenario was initially chosen due to the interesting bifurcation behavior it exhibits within the Reynolds regime considered here. This behavior was documented experimentally in [39], and it was reproduced successfully in the simulation. Specifically, as the incoming flow leaves the small inlet channel, it will attach to the upper or lower wall of the large channel, depending on whichever is closer

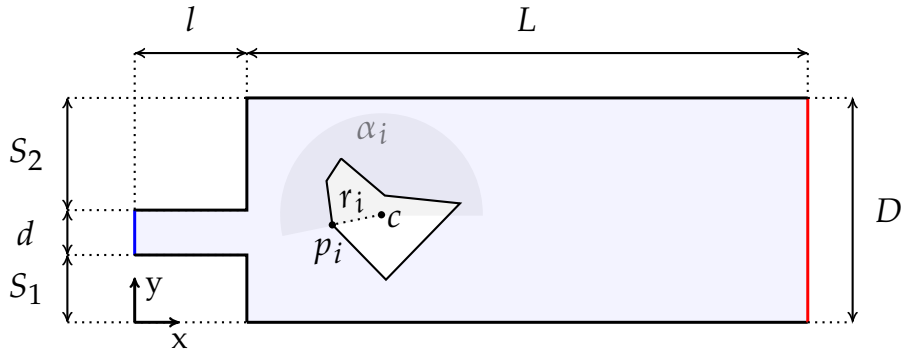


Figure 4.1: Illustration of an exemplary domain showing the relevant geometrical parameters. The annotations around the obstacle indicate how a vertex p_i is placed relative to the designated center c based on a randomly chosen distance r_i and an angle α_i. The different boundary types are color coded as inlet (blue), outlet (red) and wall (black).

to the inlet. This is particularly instructive for understanding the different behavior of GAN models compared to conventionally trained ones, as will be discussed in detail in Section 5.2.

Inside the large channel, a single star-shaped polygonal obstacle is placed. To determine the shape, first a center point and the number of vertices are randomly chosen. Then, the coordinates of each vertex are (randomly) selected in polar coordinates (with origin at the center point), and the boundary is drawn between them in clockwise direction. In order to avoid problems later during the meshing process caused by overly thin domain regions, vertices are moved onto the wall boundary if they are too close to it. Moreover, fully closed off "domain islands" between obstacle and walls are prevented by removing all vertices between two vertices on the boundary. In order to avoid excessively high flow velocities, the distance between the highest and lowest points of each obstacle is capped at $1 - d$. For more details on how the obstacles are generated, see Algorithm 1 on page 27, describing the process in pseudocode. A total number of 10 000 samples were generated this way.

For testing the generalization capabilities of the trained models, two additional test sets were created. These differ from the training set by the type of obstacle present. The first, termed the "double" set, contains 100 samples with not one but two polygonal obstacles placed in the channel. For technical reasons to do with the meshing process, the obstacles were not permitted to be in contact with one another or the walls. The second test set again only includes a single obstacle in each sample, but of a shape that is distinctly different from the distribution in the training set. Each obstacle $\mathcal{O}$ is defined by a condition of the following type:

$$\mathcal{O} = \left\{ (x, y) \quad | \quad ax^2 + b\left(y + cx^2\right)^2 \leq T \right\}. \tag{4.1}$$

The obstacle coordinates are then randomly rotated and shifted to obtain a varied dataset. The parameters are taken from the following distributions: $a \sim \mathcal{U}(30, 60)$, $b \sim \mathcal{U}(10, 100)$, $c \sim \mathcal{U}(0, 10)$, $T \sim \mathcal{N}(2.5, 2)$, where the latter is kept to the interval $[1, 8]$. The origin point is chosen identically to the center point in of the polygonal obstacles in the training set (see Algorithm 1 on page 27). We will refer to this as the "curved" set. For examples of each of the three sets, refer to Figure 4.3.

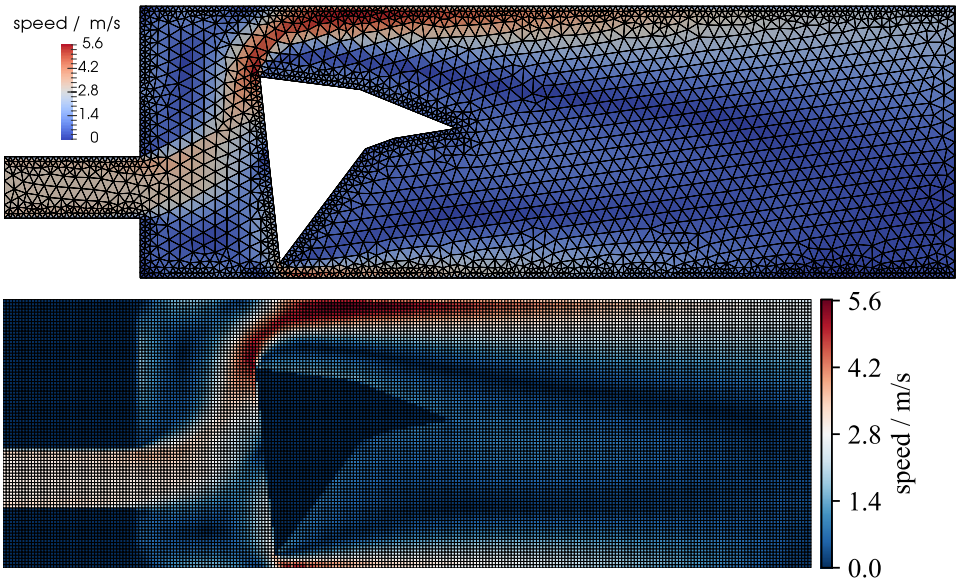


Figure 4.2: Example of flow speed data on computational grid (top) and after interpolation to a regular image grid (bottom). Both images have the same scale and their x-coordinates line up. The computational grid is made up of 4949 cells for this geometry, while the image grid contains 21 760 . Magnify for proper display.

Next, each case had to be spatially discretized by dividing the domain into a computational mesh. This mesh must be fit for use with the specific method for PDE discretization, in this case a finite volume scheme. For this purpose we used the free open source software gmsh [13] to create an unstructured quasi-2D mesh of triangular prisms. How fine the mesh ought to be at a given location in order to obtain accurate result is generally related to how large the gradients of field quantities are expected to be at that location. In particular this means that near walls, a very fine meshing is typically required to properly simulate the boundary layer. As a consequence, this requirement leads to a large increase in the total number of cells in the mesh, which amounts to a drastic increase of computational effort. To avoid this, methods have been developed to model the effect of the boundary layer instead of simulating it; these models are known as wall functions. Given that the focus of this research is on the training of GANs on fluid data, rather than obtaining the absolute best simulation accuracy, the much greater computational cost of simulating the boundary layer did not appear justified in this case. For this reason, we decided to use a wall function for modeling boundary layers, and therefore only applied a minor refinement to the wall mesh. The gmsh size parameter was chosen as 2 cm at walls as opposed to 5 cm elsewhere within the domain. This way, the number of cells was kept to around $5000 - 10000$, depending on the geometry.

For solving the RANS equations presented in Section 2.1, we used the popular free open source toolbox OpenFOAM [59]. It includes a vast range of tools and solvers for continuum mechanics problems, with a focus on CFD. Specifically, we used the simpleFOAM solver [41]. As per its name, it implements the quasi-transient fractional step method by the name of SIMPLE, summarized in Section 2.2.2. The settings for solvers and schemes were based roughly on the OpenFOAM tutorial case "backwardFacingStep2D" (see the repository [49] for the relevant files). The content

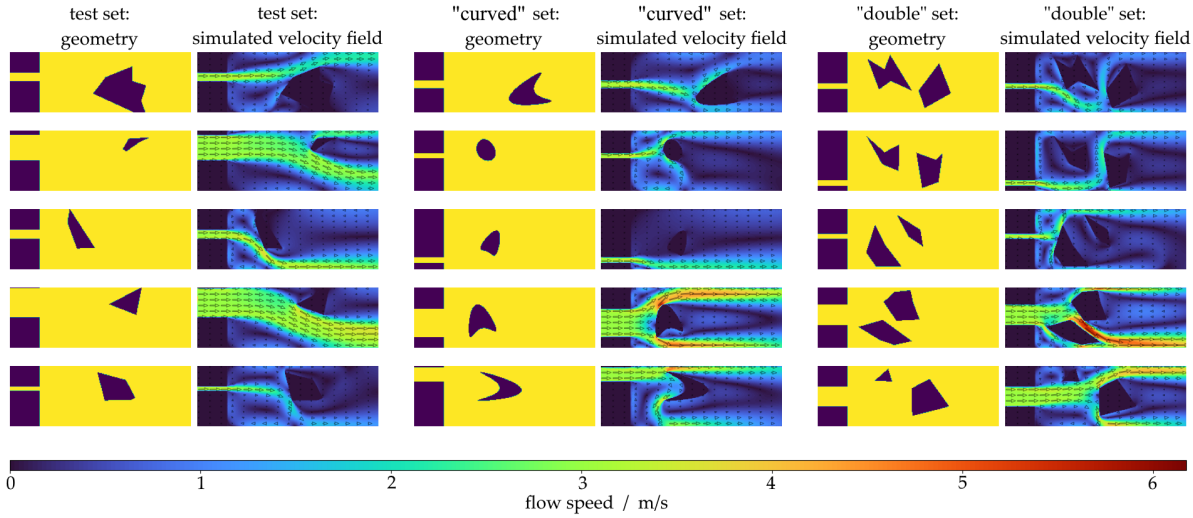


Figure 4.3: Random selection of samples from the three datasets generated for the project. In each column, the image on the left shows the geometry input, while the one the right visualizes the simulated velocity field. This is done through a color scale representing the velocity magnitude, as well as a vector plot displaying the flow direction.

of the most import configuration files are included in Appendix A. As discussed in Section 2.1.3, we used the k-ε turbulence model. Occasionally, a simulation would diverge. In order to reduce the frequency of this occurrence, small relaxation factors of 0.5 were used.

Given the relatively high Reynolds range and the random geometry generation, it was not guaranteed that problems would be well-posed as steady-state. This would manifest as oscillation rather than convergence during solving. If convergence did not occur after 3000 steps, the result as well as the geometry was scrapped and the generation process was started over. Solving a case took approx. between 3 and 15 seconds on an Intel Core i7-6700K CPU at 4.00 GHz, depending mostly on how quickly it converged. After a solution was obtained on the computational mesh, it had to be interpolated to an image, i.e. a regular grid. For that purpose, the sampling utility of OpenFOAM (see [36] for details) was used, specifically the "cellPoint" sampling method. Here, each cell is decomposed into tetrahedrons, whose vertices coincide with the cell center as well as three cell vertices. Then, each vertex is assigned a value. That of the center vertex is given by the cell's value, and the others are taken as a convex combination using the center value as well as the neighboring cell values. Then, the value at any point in the volume can be calculated by linear interpolation on the respective tetrahedron. In order to focus on the relevant fluid mechanics and avoid an overly large aspect ratio, we restricted sampling to the first 3 m in horizontal direction. The sampling was done with a resolution of 256×85. This might seem overly large, given that it results in a number of pixels that is larger than the number of cells in the simulation. However, it must be considered that in contrast to the computational mesh, the image grid is not specifically adapted to the geometry. The resolution was therefore chosen to greatly reduce the issue of 'jagged edges', and improve the overall quality of the geometry representation on the image grid. Figure 4.2 shows an exemplary result on the computational grid alongside the interpolated result on the image grid.

Algorithm 1 Procedure for generating a star-shaped obstacle in pseudocode. New obstacles are generated until one is found that meets all criteria. $\{c_x, c_y\}$: coordinates of center point; $\{r, \alpha\}$: vertex polar coordinates relative to center; s: scaling parameter; X: list of vertex x-coordinates; Y: list of vertex y-coordinates. All operations on lists are performed element-wise. Units in m.

admissible $\leftarrow 0$
while admissible = 0 **do**
  draw c_x from $\mathcal{U}(1, 3)$
  draw c_x from $\mathcal{N}(\mu, \sigma)$
  if $c_y \notin [0.05, 0.95]$ **then**
    skip to next iteration
  end if
  draw N from $\mathcal{U}_p(3, 9)$
  draw list α of N elements from $\mathcal{U}(0, 2\pi)$, sort ascending
  draw list r of N elements from $\mathcal{U}(0.1, 0.6)$
  draw scaling s from $\mathcal{U}(0.5, 1.5)$
  draw coin-flip c from $\mathcal{U}_p(0, 1)$
  if $c = 1$ **then**
    $s \leftarrow 1/s$
  end if
  $X \leftarrow c_x + s \cdot r\cos(\alpha), \quad Y \leftarrow c_y + s \cdot r\sin(\alpha)$
  if any element of $X \notin [0.6, 3]$ **then**
    skip to next iteration
  end if
  $p_0 \leftarrow$ list of indices where $Y < 0.05, \quad Y[p_0] \leftarrow 0$
  $p_1 \leftarrow$ list of indices where $Y > 0.95, \quad Y[p_1] \leftarrow 1$
  if length(p_0)> 0 **then**
    $i \leftarrow \max(p_0) - \min(p_0)$
    delete elements from X and $\mathbf{Y}$ with indices $\in [p_0 + 1, p_0 + i)$
  end if
  if length(p_1)> 0 **then**
    $i \leftarrow \max(p_1) - \min(p_1)$
    delete elements from X and $\mathbf{Y}$ with indices $\in [p_1 + 1, p_1 + i)$
  end if
  if $1 - (\max Y - \min Y) < d$ **then**
    skip to next iteration
  end if
  update polar angles: $\alpha \leftarrow \text{atan2}(Y/X)$
  sort α in ascending order, apply same sorting to X and Y.
  define polygon by connecting vertices defined by $\{X, Y\}$ in the given order
  if center point not contained in polygon **then**
    skip to next iteration
  end if
  if $\mod(\alpha[i] - \alpha[j], \pi) < 10\,\text{deg}$ for any adjacent vertices i, j **then**
    skip to next iteration
  end if
  admissible $\leftarrow 1$
end while

4.2. Machine Learning Implementation

The artificial neural networks were implemented using the free open source machine learning library TensorFlow [58] in version 2.9. It offers efficient low level tensor computation, differentiable computing and parallelization capabilities.

As a template for building a GAN fluid model, we used the image-to-image translation model pix2pix, first presented in [24]. This GAN model was developed as a general purpose tool, and has been successfully applied across a large variety of tasks. These include semantic segmentation, image inpainting, conversion of sketches or schematics to photos, greyscale to color image conversion, conversion of aerial photographs to street maps, and day to night scene conversion, among many others. The code was made freely available online, and has since enjoyed great popularity in the ML community. The model was adapted to this application, but otherwise largely unchanged In the following we give a summary of the most important architectural features and characteristics of the model that we used for all of the experiments presented in the next chapter (except where modifications are specifically mentioned).

In the pix2pix model, the generator is trained not only on the feedback provided by the discriminator, but also directly on a ground truth loss. In our model this is optional, with the total objective function being

$$V(G,D) = \mathbb{E}_{p(\mathbf{x},\mathbf{c})}\Big[\lambda_{\text{GAN}}\left(\log D(\mathbf{x},\mathbf{c}) + \log\left(1 - D\left(G(\mathbf{c}),\mathbf{c}\right)\right)\right) + \lambda_{\text{L1}}\overline{\|G(\mathbf{c}) - \mathbf{x}\|_1}\Big], \quad (4.2)$$

where $\mathbf{c}$ is the input image, $\mathbf{x}$ is the ground truth and the overline denotes the arithmetic mean taken of the point-wise L1 norm. The parameters λ_{GAN}, λ_{L1} are hyperparameters allowing us the change the relative weight of the discriminator feedback versus the ground truth error measured in the L1 metric. The loss functions for a single input are therefore

$$\mathcal{L}_{\text{gen}} = \lambda_{\text{GAN}} \log D(\mathbf{x},\mathbf{c}) + \lambda_{\text{L1}}\overline{\|G(\mathbf{c}) - \mathbf{x}\|_1} \quad (4.3a)$$

$$\mathcal{L}_{\text{disc}} = \lambda_{\text{GAN}}\left(\log D(\mathbf{x},\mathbf{c}) + \log\left(1 - D\left(G(\mathbf{c}),\mathbf{c}\right)\right)\right). \quad (4.3b)$$

Initially, we focused on comparing a 'pure' GAN model trained using $\lambda_{\text{GAN}} = 1, \lambda_{\text{L1}} = 0$ with a model trained only directly on ground truth, i.e. $\lambda_{\text{GAN}} = 0, \lambda_{\text{L1}} = 1$. The motivation given by the authors for including the ground truth loss term is that it had been shown previously that training based on ground loss alone is already well suited for capturing the deterministic, large-scale features of many image translation tasks. The adversarial training on the other hand can help the generator fill in high-fidelity detail and produce results which appear more realistic. In Section 5.9, we also analyze results from training hybrid models trained on both parts of the objective function.

An important difference of the pix2pix model compared to other GAN models is that there is no latent space, with the authors stating that their initial testing showed the generator would learn to simply ignore it. Moreover, since we are not interested in stochasticity for the sake of it, we omit the dropout feature,[1] which the pix2pix authors used to introduce randomness into the prediction. Therefore, the output of our model is completely deterministic.

[1] With dropout, a certain random percentage of neuron are removed from the network at each evaluation.

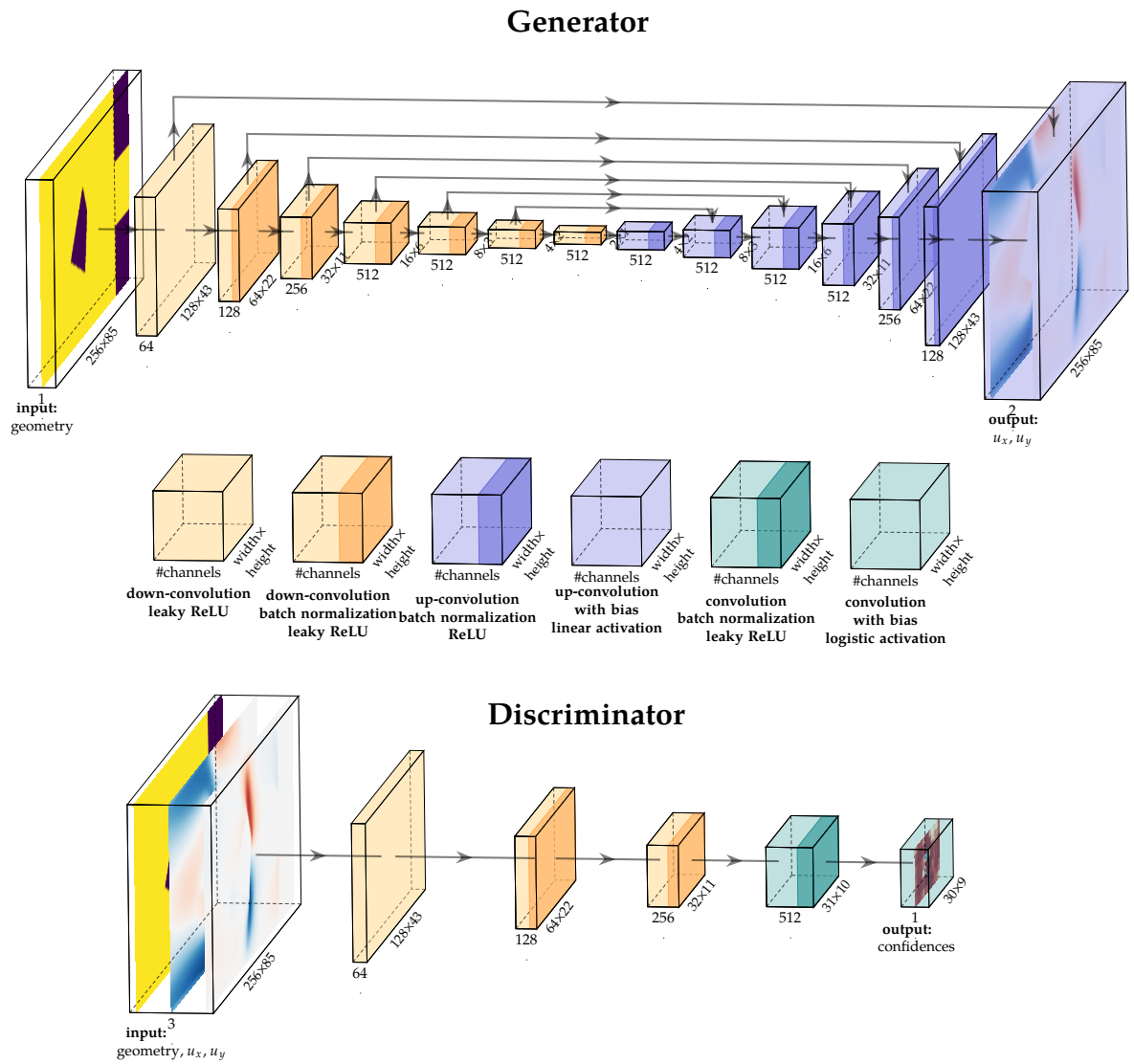


Figure 4.4: Baseline GAN architecture. Boxes indicate layers networks, and arrows symbolize connections between them. Visualization based on [22].

As stated, the architecture of our models is largely identical to that of pix2pix, which in turn implements many of the recommendations from earlier papers discussed in Section 2.3.3. See Figure 4.4 for a detailed sketch of the exact architecture used. In general, both generator and discriminator are purely convolutional without any fully connected layers. Both down- and up-sampling operations are 2-stride convolutions with a 4×4 kernel size. The effect of increasing kernel size in the discriminator was also tested, see Section 5.11 for the results. No bias is used except for at the output layers. All layers, except for input and output, are batch normalized, and all except for output layers use rectified linear activation functions. Specifically, the down-convolutions have leaky ReLU functions with $a = 0.3$, while the up-convolutions are simply ReLU. The total number of trained parameters is 48.6 million for the generator and 2.77 million for the discriminator.

The generator has a U-Net type bottleneck architecture with skip connections at

every level, i.e. each decoding layer operates on the channels of the previous decoder layer as well as those of the encoder layer of the corresponding level (see [51]). There are seven layers each in both encoder and decoder, and cropping is applied between layers as necessary to ensure the symmetry required for the skip connections. The number of channels is increased closer to the bottleneck. Overall, the generator maps from a 256×85 binary encoding of the geometry to an output tensor with dimensions $256 \times 85 \times 2$, with one channel for each velocity component. The bottleneck layer has a dimension of $2 \times 1 \times 512$. After the last layer, the output is multiplied point wise with the input, i.e. any value corresponding to a location outside the domain is set to zero. This comes without loss of general applicability, as the domain is always known a priori, and it is done in order to not unnecessarily complicate the loss landscape during optimization.

The discriminator has three down-convolutional layers, followed up by two unstrided convolutional layers. The effect of increasing the number of down-convolutional layers was also evaluated, see Section 5.11. It maps from an input tensor with dimensions $256 \times 85 \times 3$, where one channel contains the geometry and the others contain the velocity components, to an output tensor of size 30×9 with a range of $(0, 1)$. With this architecture, not every value of the input tensor affects every value of the output. Instead, each output pixel has a receptive field consisting of a $70 \times 70 \times 3$ batch of the input (or smaller due to cropping if we are considering an output pixel close to the boundary). The patches of neighboring output pixels are highly overlapping, and their size depends on the exact architecture (see [23] for a script on how to calculate it provided by one of the pix2pix authors). Thus the image is not discriminated as a whole, but instead individual patches of the image are classified on a real-fake spectrum. The authors of pix2pix termed this discriminator architecture PatchGAN, and it implies that the discriminator cannot identify patterns larger than this patch size. Moreover, since the operation performed is the exact same for all patches, the PatchGAN can be understood as a type of texture or style loss.

At the start of training, all weights (i.e. kernels) are randomly initialized to a normal distribution with zero mean and a standard deviation of $\sigma = 0.01$. The models were trained on the main data set described above, where 10% of the 10000 sample was held back as a test set. Since no major hyperparameter optimization was done, no validation set was used. Training was done in mini-batches of size 32 using the adaptive gradient based optimization algorithm Adam [30] with a learning rate of 10^{-4} and initial decay rate parameters of $\beta_1 = 0.5, \beta_2 = 0.999$ for both generator and discriminator. See Section 5.10 for experimental data on the effect of modifying the learning rates.

Computing a prediction took about 0.17 s on an Intel Core i7-6700K CPU at 4.00 GHz (excluding TensorFlow overhead), which is between one and two orders of magnitude faster than the simulation on the same chip (and this does not even take into account potentially improved parallelizability). On the other hand, there is certainly still potential for speeding up the simulation at the cost of reduced accuracy.

4.3. Live-Prediction Tool

As part of this research project, a small program was developed for demonstration and experimentation purposes. Written in Python, it allows the user to draw any polygonal obstacle into the channel geometry, as well as modify the size and position

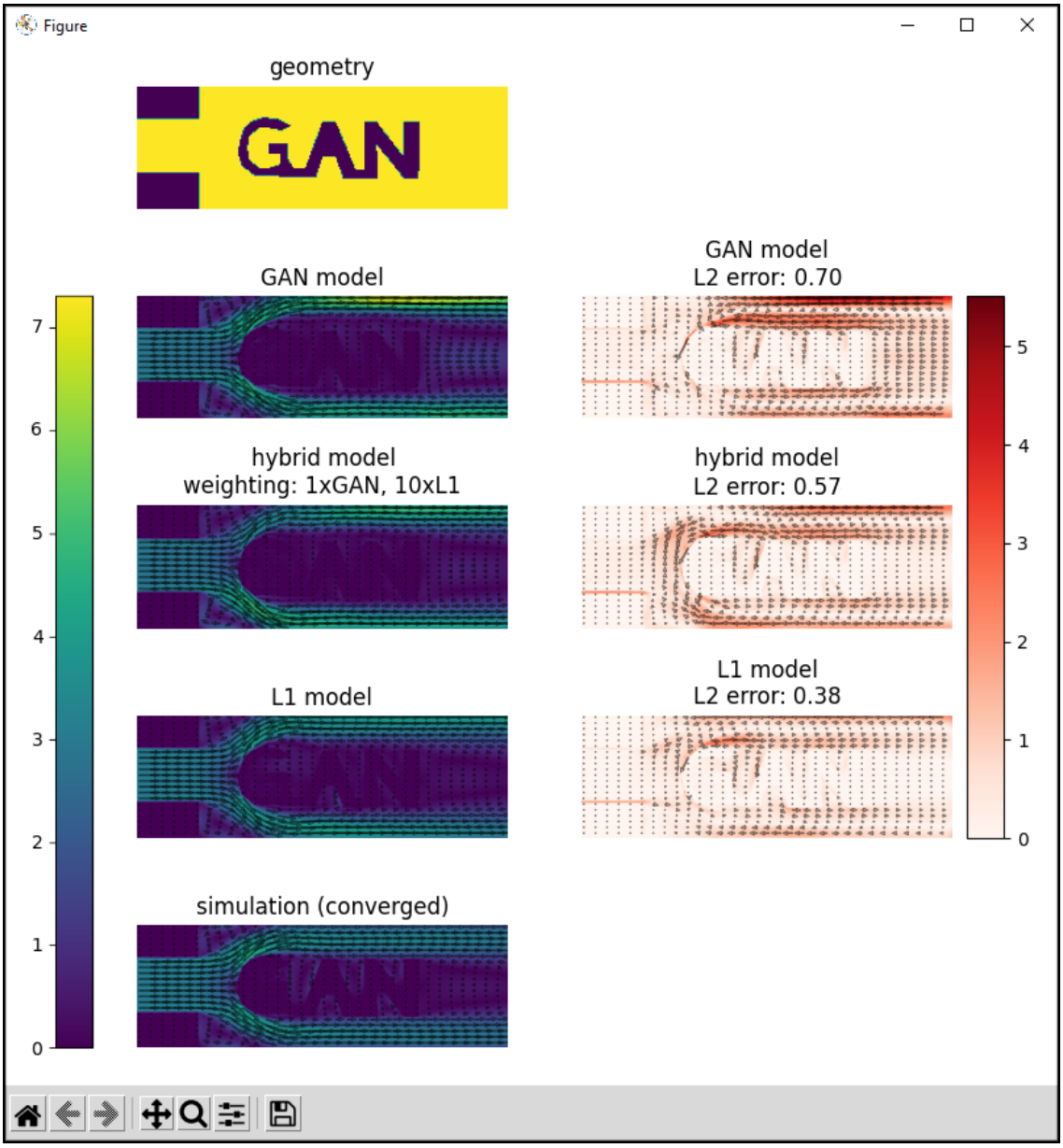


Figure 4.5: Screenshot showing the user interface of the live-prediction tool with three ML models for comparison and vector plots enabled.

of the inlet. When confirmed, the program will (almost instantly) show the flow speed predictions of a number (two or three) of previously selected neural network models (e.g. an adversarially trained model and one trained with an L1 loss) on separate panels. At the same time, an OpenFOAM simulation is started in the background. As the iterative solving progresses, the latest, and eventually final, simulation result is shown in another panel. Moreover, in another column the error of each of the predictions with respect to ground truth is plotted, and the average L2 error (see Section 5.1) is printed. All plots in the same column (showing flow speed and error fields respectively) share the same color scale, and a vector plot overlay can be added optionally. A screenshot of the software is shown in Figure 4.5.

The program gives an impressive demonstration of how much faster the reduced

order neural network prediction can be evaluated compared to the simulation. In this way, it can be seen as a proof-of-concept for the utilization of such machine learning based surrogate models in design space exploration or real-time prediction. At the same time, the ability to freely draw shapes allows the user to easily push the models up to and beyond their limits, and develop an improved intuition for how they behave when confronted with unfamiliar types of geometries, which in turn can be helpful during research.

5

Experimental Results

The main part of the research for in this thesis concerned training GAN models on the fluid simulation dataset, and comparing the results among themselves as well as to regular CNN models trained directly on ground truth. This chapter presents the result of these experiments, split into in multiple parts. The first section presents initial findings and serves as a baseline and reference point for further analysis in later sections. Next, in Section 5.2, the most important qualitative patterns of difference between the two model types are discussed. Subsequent sections extend the comparison in breadth and depth by analyzing generalization performance (Section 5.3), residuals of the continuity equation (Section 5.4), prediction artifacts (Section 5.5), the discriminator output (Section 5.6), the prediction of pressure (Section 5.7) and how the amount of available training data affects the model's performance (5.8). The final sections evaluate the performance of additional trained models differing in their loss functions (Section 5.9), their learning rate (Section 5.10) and architectures (Section 5.11).

5.1. Performance Baseline

Without a doubt, the most import metric for assessing how well a trained neural network performs as a reduced order fluid model is to compare the resulting prediction on the unseen test samples with the ground truth (i.e. simulation result) on a point-by-point basis. Arguably the most sensible way of doing this is to take the difference between the two velocity vector fields, and compute the average vector length on it, i.e. measuring the average difference in the L2 norm. We refer to this as the L2 error.

To start, we compare the L2 error performance of adversarially trained models with those trained directly on ground truth. Specifically, we set $\lambda_{\text{GAN}} = 1$, $\lambda_{\text{L1}} = 0$ and vice versa respectively in their cost functions (see Eq. 4.3). We trained five identical models (differing only by their random parameter initialization) of each type on the training set of 9000 samples. The rationale for training multiple models of identical structure is to get a sense of how sensitive the outcome is to the initialization. Training for 1000 steps took on average 181 s for the pure L1 model and 274 s for the GAN model (51 % longer) on a NVIDIA GeForce GTX 1080 Ti graphics chip. Most models were trained for 50 000 steps, which at a batch size of 32 corresponds to roughly 178 epochs (i.e. the model will have encountered each sample around 178 times during training). To test

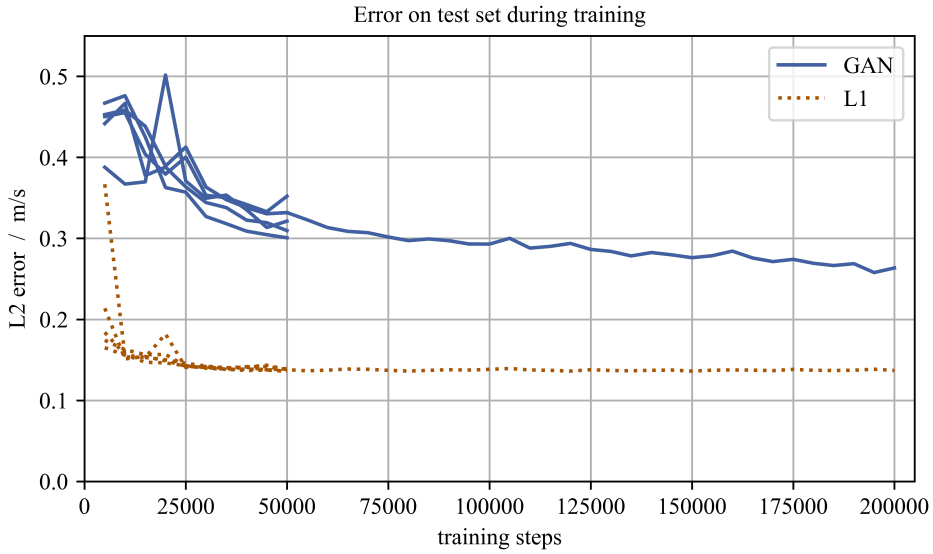


Figure 5.1: Evolution of test set L2 error during training for five identical pure GAN models and five pure L1 models.

how even more training affects outcomes, one model of each type was trained for a total of 200 000 steps, i.e. more than 711 epochs.

The most central finding from this analysis is that the L1 trained models, without exception, perform much better on this accuracy metric. After 50 000 steps the average L2 error of the ground truth trained models is 57 % smaller than that of the GAN models, and between the models trained for 200 000 steps the difference is still 48 %. The data also show that the model spread due to the random initialization is initially quite large for both model types, but decreases quickly for the L1 type, becoming negligible after about 25 000 steps. For the GAN models on the other hand, the spread remains significant, with an estimated standard deviation of the mean L2 error on the test set of 0.02 after 50 000 steps. Moreover, the L1 models' performance also quickly levels off at around 30 000, whereas the GAN model continues to improve throughout the entire evaluated range of steps. This is not surprising, as training convergence in GAN models is a more complicated process subject to the evolving dynamic between generator and discriminator. Nevertheless, looking at the observed trend it appears unlikely that the GAN model would eventually reach the performance of the L1 trained model given more training.

For a more detailed analysis, we compare the distribution of L2 errors on the 1000 test samples rather than just looking at the mean. Figure 5.2 visualizes the distributions of both models trained for 200000 training steps in a histogram. One can see that the two error distributions share similarities, with a steep rise and a long tail. However, the GAN model's distribution is significantly stretched towards larger values. The median L2 errors are 0.214 and 0.104 for GAN and L1 respectively, with the former outperforming the latter in 8.4 % of cases.

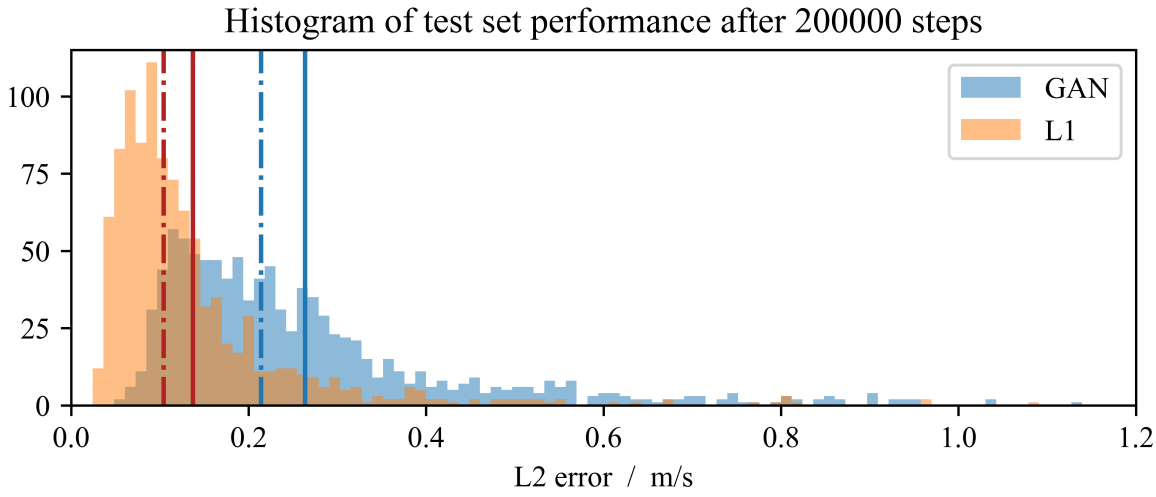


Figure 5.2: Histogram visualizing the distribution of L2 errors on the test set for both models after 200 000 steps. The dashed and solid lines in red and blue show median and mean for L1 and GAN trained models, respectively.

5.2. Characteristic Patterns

In order to form a theory to explain the statistical results shown in the previous section, we need to take a closer look at the individual model predictions. In the following we will present illustrative examples that help us identify and describe the most important systematic qualitative differences between predictions generated by the two types of models.

As mentioned in Section 4.1, flow through a sudden expansion tends to attach to whichever wall is closer to the inlet (at least in this Reynolds regime). In case the inlet is roughly centered however, the flow condition can be characterized as an unstable equilibrium and the outcome is essentially random. By analyzing how the two different types of ML models handle such a bifurcation, we can demonstrate important differences between them.

Figure 5.3 shows the prediction outcomes of the two model types on two such bifurcation geometries. In both, the inlet is approx. centered, but the flow attaches to the upper wall in one, and to the lower wall in the other example. In both cases, the GAN model predicts almost exactly the same result. In fact, further tests showed that whenever the outlet is approx. centered, the GAN will predict flow attachment to the same side. The L1 model on the other hand produces predictions in both cases that look akin to a superposition of the two possible bifurcation outcomes. This difference in behavior is rooted in the different training methodologies, and in order to understand it we need to consider how the different cost functions would act under this type of scenario.

If the dataset is large enough, we can expect to encounter a roughly equal number of both bifurcation outcomes during training. In such a case there is no unique optimum in terms of L1 loss, in fact every convex combination of the two solutions will, in terms of expected value, be assigned an equal loss. This type of combination is exactly what we see in the L1 prediction of Figure 5.3. In contrast, a GAN-generator would most likely incur a large cost if it produced such a superposition, since it is an unphysical

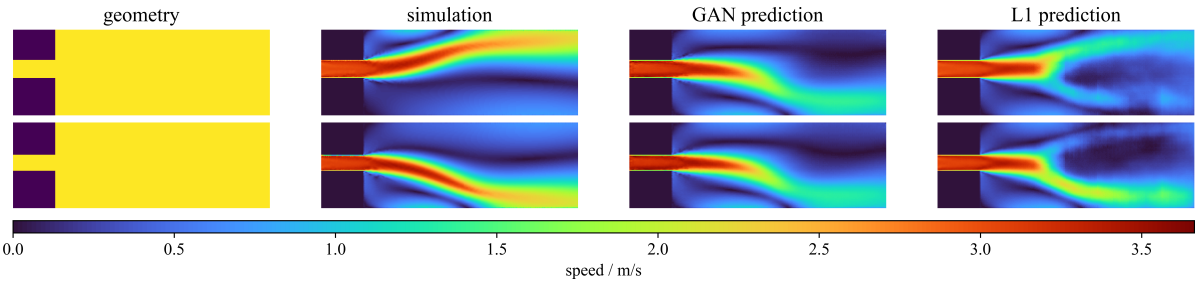


Figure 5.3: Flow speed simulation & prediction for two bifurcation examples. Models were trained for 50 000 steps.

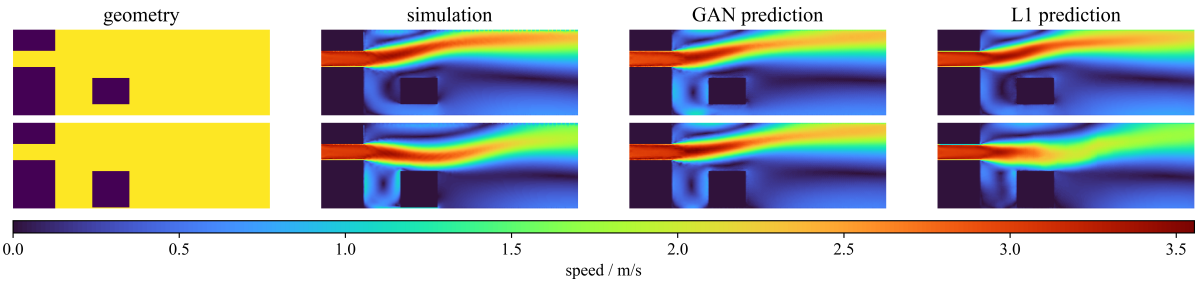


Figure 5.4: Flow speed simulation & prediction for two geometries specifically designed to show the effect that blocking back-flow has on flow attachment. The models were trained for 200 000 steps.

flow condition that never occurs in the training data and could therefore easily be detected as artificial by a well-trained discriminator. Loosely speaking, the generator learns to force clear-cut decisions in the face of uncertainty, even if, as in the example presented in the figure, it is a wrong decision. On the other hand, the discriminator does not penalize that the GAN always predicts attachment to the same side when in doubt (even though this is not truthful to the original distribution), since each prediction taken by itself appears realistic.

For the next example, we consider another aspect of the wall attachment phenomenon. The effect is caused by eddies that form naturally besides the main flow and impinge back on it. The eddy is stronger on whichever side more space is available, giving rise to a net force towards the closer of the two walls. However, by the deliberate placement of an obstacle, we can interfere with the phenomenon. Specifically, if we place an obstacle touching the wall opposite of where the flow would normally attach, the eddy back-flow is blocked and the flow attachment mechanism is disturbed. An example of this effect is shown in Figure 5.4: when there is space between the obstacle and the opposite wall, attachment occurs as normal. However, when the obstacle is extended such that it touches the wall, flow attachment is shifted significantly downstream.

As displayed in the figure, if the obstacle is not touching the wall, both models produce very similar predictions, which are moreover very close to ground truth. For the case where the obstacle is extended up to the wall however, significant differences between the two predictions are observed: the GAN prediction is hardly different compared to the previous case. On the other hand, the L1 model apparently learned an better internal representation of this phenomenon, and produces an output that more closely resembles ground truth. Here again we can also observe that the L1 model

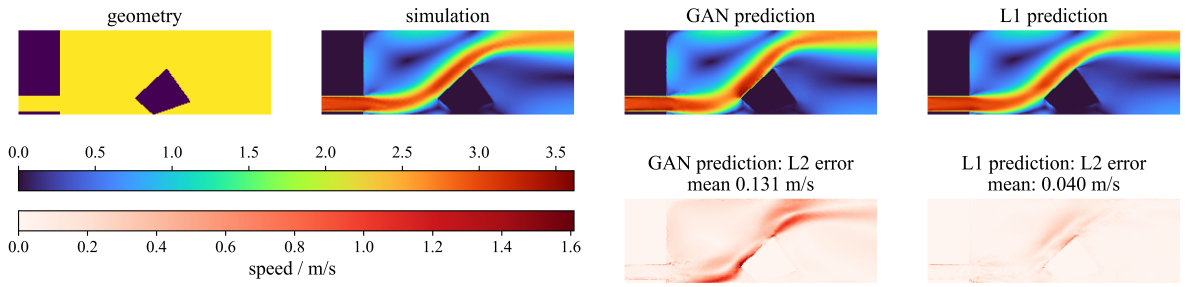


Figure 5.5: Flow speed simulation & prediction for a representative sample from the test dataset. The second row of images plot the L2 error of the predictions. The models were trained for 200 000 steps.

represents uncertainty by superposition, as evidenced by the widening of the flow. The uncertainty in this case stems not from intrinsic stochasticity as with the bifurcation example, but from the fact that the exact formation of this blocked back-flow scenario is not trivial to predict, and there were only few relevant samples in the training set. Overall, the GAN produces a flow field that appears much more realistic to an uninitiated observer, but is much further from the ground truth in terms of L2 error. This is an important pattern seen throughout the dataset.

Thus far in this section we have analyzed geometries that were specifically designed to highlight differences between the two model types. However, to understand why the GAN model on average performs so much worse on the test set, it is helpful to look at an 'average' example from the test set. An appropriate definition of average in this case is the median of the distribution of the difference in L2 error between the two models. Ground truth and predictions for that particular sample are shown in Figure 5.5. To a casual observer, all three velocity fields appear very similar. However, looking at the L2 error plot it becomes clear that indeed the L1 prediction outperforms the GAN model by a wide margin. This is representative of most samples in the test set; both models produce results that look plausible on first glance, but the L1 outcome is simply much closer to ground truth. In some sense, this behavior is not too surprising. After all, the adversarially trained generator fundamentally is only trained to produce outputs that 'look right' to the discriminator, rather than actually being close to ground truth. This is likely the reason that the GAN outcomes ultimately fall short in terms of accuracy compared to the L1 trained model in most metrics.

5.3. Generalization Performance

In the first section of this chapter it is established that the GAN model was not able to reach the performance of the L1 model when tested on unseen samples from the training distribution. However for building a reduced order model, this is not the only relevant criterion, we are also concerned with how the model performs on completely different types of geometries that are not drawn from the training distribution. This is because we cannot necessarily expect that every type of problem to which our model is later applied would have been well represented within the training data set. To test this ability of the model to generalize to new problem settings, we have created two generalization data sets as described in Section 4.1. It would be conceivable that despite being less exact in its predictions on the test set, the adversarially trained model

has built up more robust internal representations of the flow physics that help it on these more challenging problems. However, this is not what the data show. See Table 5.1 for a comparison of the most important characteristics of the L2 error distributions.

Table 5.1: Characteristic quantities of the L2 error distributions on the test data sets. Models were trained for 200 000 steps.

dataset	quantity	L1	GAN
test	mean / $\mathrm{m\,s^{-1}}$	**0.137**	0.263
	median / $\mathrm{m\,s^{-1}}$	**0.104**	0.214
	share won / %	**91.6**	8.4
"curved"	mean / $\mathrm{m\,s^{-1}}$	**0.298**	0.438
	median / $\mathrm{m\,s^{-1}}$	**0.277**	0.405
	share won / %	**85.9**	14.1
"double"	mean / $\mathrm{m\,s^{-1}}$	**0.368**	0.495
	median / $\mathrm{m\,s^{-1}}$	**0.345**	0.488
	share won / %	**90.0**	10.0

As expected, both models perform much worse on these datasets compared to the previously considered test set. Nevertheless, in many samples from the "curved" set, both models still manage to produce a decent prediction such as in the example shown in Figure 5.6. In comparison however, the GAN model predictions are on average again much less accurate on this set.

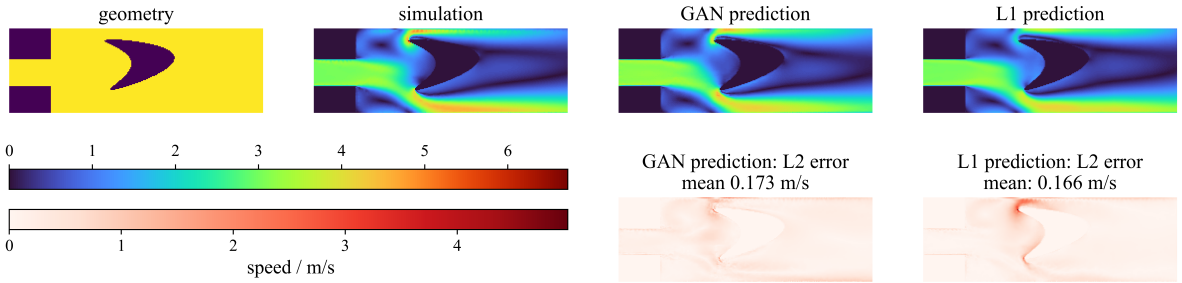


Figure 5.6: Flow speed simulation & prediction of a sample from the "curved" data set. The second row of images plot the L2 error of the predictions. The models were trained for 200 000 steps.

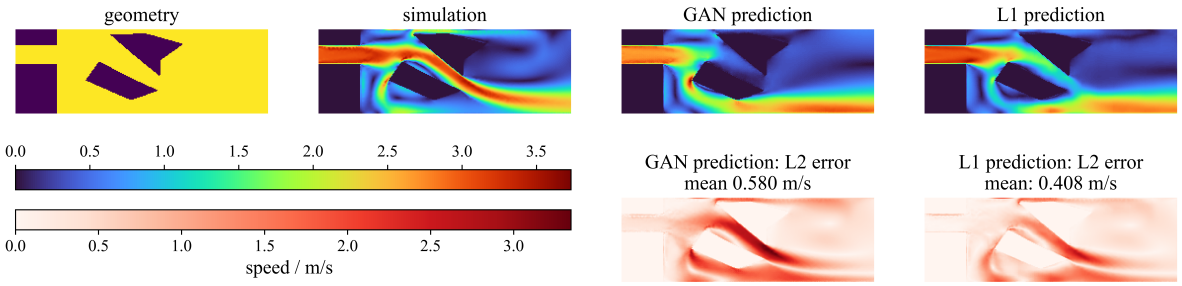


Figure 5.7: Flow speed simulation & prediction of a sample from the "double" data set. The second row of images plot the L2 error of the predictions. The models were trained for 200 000 steps.

Next we take a closer look at how well the models can predict flow if a second obstacle is added to the problem. In cases where only one obstacle is significantly interfering with the main flow or the obstacles effectively act as one (e.g. because one obstacle is shielding the other), predictions are usually fairly accurate, qualitatively speaking. Thus we can conclude that the models are robust enough not to be completely useless in case of a second obstacle, as long as the resulting flow dynamics are not too different from that of the single obstacle case. If however the main flow is channeled through a gap between the two obstacles, both models fail for the most part to provide reasonable predictions. That being, said, the L1 trained model does handle such cases significantly better than the GAN model. The latter appears to completely ignore smaller gaps between obstacles, whereas the former often correctly predicts a certain amount of flow through the gap, if with a magnitude that is much too small. A good example of this is shown in Figure 5.7.

5.4. Continuity Residuals

Beyond looking at the error to ground truth, another relevant performance metric in the context of solving differential equations is the residual. This term refers to the difference between the two sides of the equation at each point of the approximated solution. Here we will restrict ourselves to the residual of the continuity equation (2.12a), since evaluating the RANS momentum equation (2.12b) is much more involved. To calculate the continuity residual, we need to evaluate the divergence of the velocity field. We compute the necessary derivatives by a central difference scheme. Thus we get

$$R_{i,j} = \nabla \cdot \mathbf{u}\,|_{i,j} = \frac{v_{i+1,j} - v_{i-1,j}}{2h} + \frac{w_{i,j+1} - w_{i-1,j-1}}{2h}, \tag{5.1}$$

where R is the residual, v and w are the two velocity components, the grid coordinates are $\{i, j\}$ and h is the grid spacing. Given that the height of the domain is 1 m and there are 85 grid points in vertical direction, we have that $h = 1/84\,\text{m}$. In order to avoid boundary effects, we only consider grid points that do not directly neighbor a boundary point.

When applying this formula directly to the simulated training examples, we noticed that in areas of strong gradients, large high frequency residuals appeared. These are not a direct result of the simulation, but artifacts introduced by interpolating to the coarse image grid. In order to exclude this effect as best as possible, we apply a smoothing operation by filtering five times with a 3×3
enquoteotebox blur kernel. This way the average residual on the training samples was reduced by a factor of approx. 4, which helps to identify the real, as in physically meaningful, residuals in the predictions.

The distribution resulting from applying this procedure on the test set predictions of both model types is visualized in Figure 5.8. On this metric, the GAN model performs significantly better than the model trained in ground truth, which produces on average a mean residual approx. 70 % greater. This again can be traced back to the properties of the different training methodologies. As noted in Section 4.2, the discriminator feedback can be understood as a kind of learned texture loss. For the discriminator, learning the exact mapping between a geometry and a flow field is hard, but learning the typical 'texture' of training images (which is one that has a small

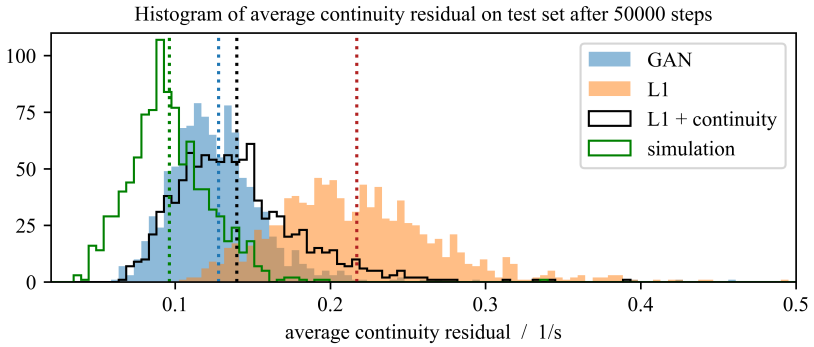


Figure 5.8: Histogram visualizing the distribution of the smoothed continuity residual on the test set predictions of both models after 50 000 steps. The dotted lines show the mean values of the corresponding distributions.

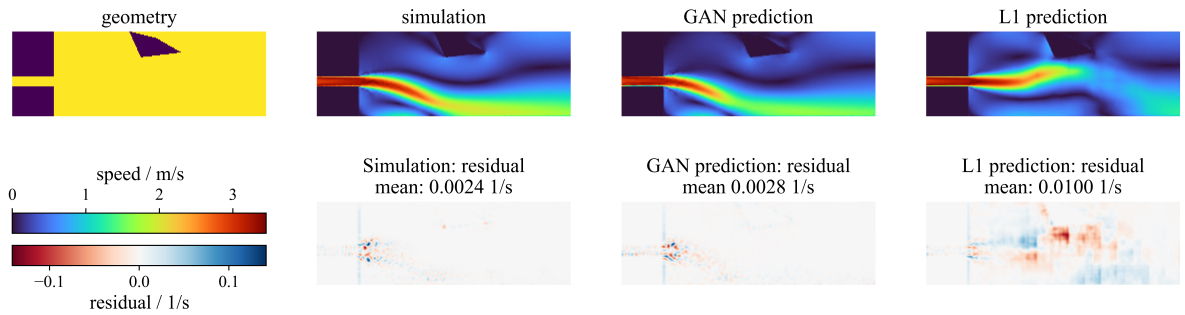


Figure 5.9: Flow speed simulation & prediction of a sample from the test set. The second row of images plot the smoothed continuity residual. The models were trained for 200 000 steps.

continuity residual) is much easier. This is abetted by the particular discriminator architecture ("PatchGAN"), but would likely hold also for other types of discriminators.

A good example of the L1 model predicting an 'unphysical' flow field with a large residual is shown in Figure 5.9. In the L1 residual plot, sharp vertical line artifacts are visible, which are only blurred by the smoothing operation. These are a common feature of L1 generated predictions though their prominence varies from sample to sample.

The question that naturally arises from this observation is whether using adversarial loss is strictly necessary to achieve this residuals performance, or whether adding a specific residuals loss term to the L1 cost function can achieve the same. In this case, the total generator loss would be

$$\mathcal{L}_{\text{gen}} = \lambda_{\text{L1}} \overline{\|G(\mathbf{c}) - \mathbf{x}\|_1} + \lambda_{\text{resid}} \overline{\|R\|_1}, \tag{5.2}$$

where $\overline{\|R\|_1}$ is the average residual magnitude. The distribution from a model trained this way with parameters $\lambda_{\text{L1}} = \lambda_{\text{resid}} = 1$ is shown alongside the previously discussed in Figure 5.8. It shows that adding such a physics-based loss is indeed suitable for reducing the residual, although in this experiment the effect fell short of the level achieved by the GAN by about 9 %. It should be noted here that the addition of the continuity loss did not have a significant effect on the L2 error distribution.

5.5. Prediction Artifacts

While the GAN model tended to produce results that appear more convincing at first glance, there are some cases where the model fails gravely and outputs an anomalous artifact. This behavior was observed with each of the five identical models trained for 50 000 steps, but only on few samples, usually less than 1 % of the test set. On the contrary, none of the L1 trained models suffered from the issue. Often, the specific pattern that appeared was almost identical between multiple faulty samples produced by the same model. Moreover, each pattern appears to be related to a specific geometric configuration. However, these conditions were different for each of the patterns and model, and there were no samples on which multiple models produced artifacts. Figure 5.10 displays some of the observed examples.

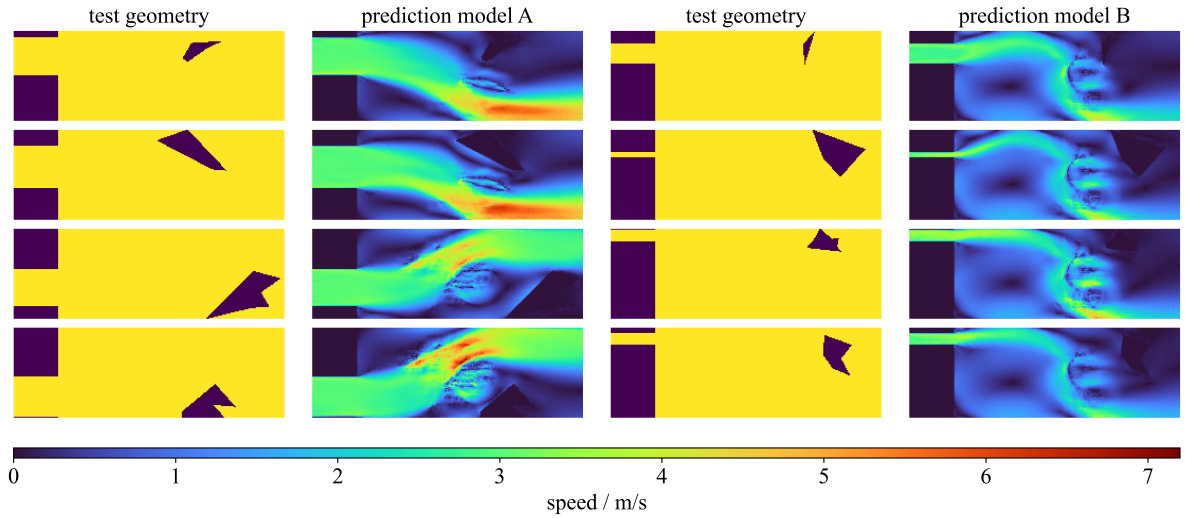


Figure 5.10: Examples of the artifacts occasionally produced by GAN models after 50 000 steps of training. The models (A and B) are identical GAN models differing only by their random initialization. The geometries are taken from the test set.

Fortunately after continuing training until 200 000 steps, these artifacts all but disappeared, with only a single, faint case identified among the 1000 test samples.

5.6. Discriminator Output

Thus far we have only looked at the performance of the generator, however, of course a GAN is made up of not one but two neural networks. In order to study the behavior of the trained discriminator, we can compare the output it produces on real (simulated) and forged (generated) flow fields of the training set. By taking the average of the 30×9 output, we obtain a probability $\in (0, 1)$ indicating the discriminator's belief that the particular input is from the training distribution (i.e. a simulation result). The histogram in Figure 5.11 shows that the discriminator indeed performs very well at discriminating the generated samples from the simulated ones. Barring just two samples out of 1000 , the two sets could be perfectly separated based on the discriminator output by choosing the appropriate threshold value. This is an indication that the discriminator dominates the generator, and the model could likely benefit from improved balance during training. This is investigated further in Section

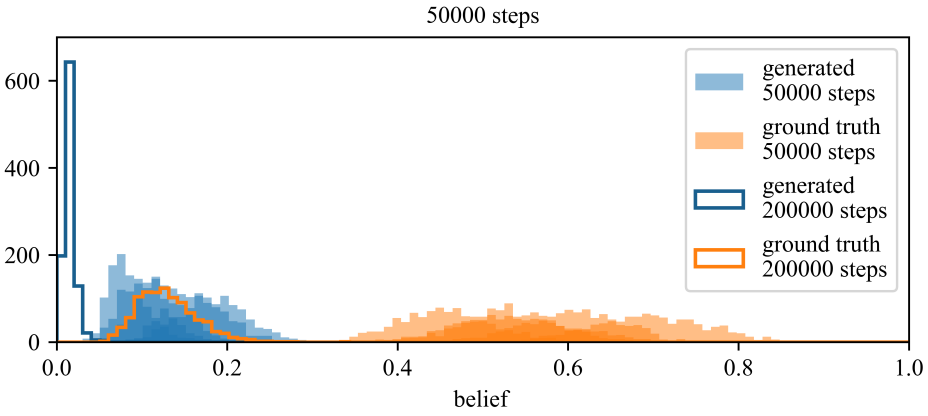
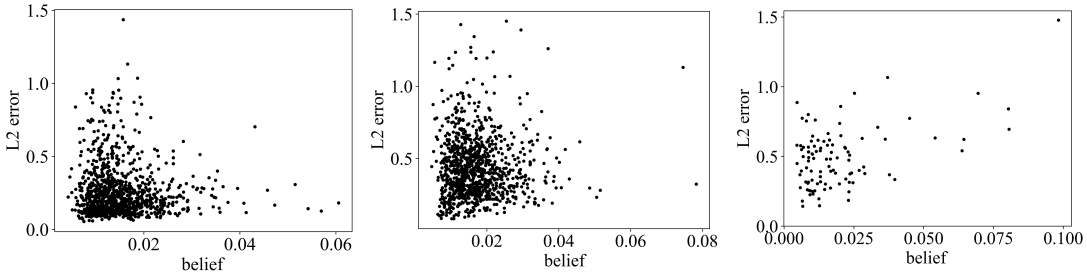


Figure 5.11: Histogram visualizing the distribution of mean discriminator output on real (simulated) and forged (generated) images from the test set. The filled bars correspond to the five identical GAN models trained for 50 000 steps, each plotted half-transparent to show how they overlap. The contoured bars show results after 200 000 steps.



(a) test set
correlation: −0.058

(b) "curved" set
correlation: 0.028

(c) "double" set
correlation: 0.491

Figure 5.12: Scatter plot showing L2 error plotted over mean discriminator output for every sample from the different data sets. The model was trained for 200 000 steps.

5.10. It is also surprising that the discriminator assigns such low belief to both groups especially after 200 000 training steps. After all the binary cross-entropy loss function would assign much lower cost if the whole distribution was just shifted to the center of the interval.

An interesting question arising from this analysis is whether there is a correlation between the spread of belief outputs of generated samples and their L2 error. If this was the case, the discriminator would provide us useful additional information on the probable quality of a generated sample, which something that a regular CNN cannot do.

If we calculate the Pearson correlation coefficient between the two quantities averaged over the five identical GAN models after 50 000 steps of training, we obtain values of −0.040, −0.032 and −0.384 for test set, "curved" set & "double" set respectively. The first two values are negligibly small, but there appears to be a significant negative correlation on the two-obstacle dataset. This translates a to higher belief corresponding to a lower L2 error on average, which is the type of relationship one might intuitively expect.

Next, let us focus on the GAN trained for 200 000 steps. we see the correlations

change from −0.079, −0.131, −0.416 after 50 000 steps to −0.058, 0.028, 0.491 after 200 000. Thus, surprisingly the correlation of the two-obstacle set switched sign over the course of further training. At first glance, the fact that there could be a positive correlation at all might appear very surprising, as one might expect that the cases where the discriminator assigns a greater belief to the generated flow field being a simulation result would be those which more closely resemble the training distribution, and therefore have a smaller L2 error. However, after more careful consideration the opposite appears just as plausible: cases where the generated sample successfully 'fools' the discriminator could be indicative of a type of input for which the discriminator is particularly 'gullible'. Thus, for samples of such type the generator has had less of an incentive to improve, leading to a higher L2 error. Perhaps a balance between these two effects is the reason for the absence of a strong correlation in both of the single obstacle datasets.

5.7. Predicting Pressure

Thus far we have focused exclusively on predicting flow velocities from geometry. However, as discussed in Section 2.1 there is of course another unknown field quantity in fluid problems, the pressure p. However this is not as negligent as it might at first glance appear, for the simple reason that the pressure may be derived from the velocity field using the governing Navier-Stokes equations. Specifically, we can evaluate the left side of the momentum equation (2.8b) to obtain the pressure gradient, which is the only information about the pressure field holding any physical meaning in an incompressible setting. However, as noted we are using a RANS approach for the simulation, wherefore this method of obtaining the pressure field comes at the cost of an error due to neglecting Reynolds stress (which the neural network does not predict). If solving for pressure is important, it might thus be worth considering to extend the ML problem by adding a channel for pressure to generator output and discriminator input respectively (as well as to the L1 loss function). The rest of the architecture is left unchanged, resulting in the total number of trained parameters in each model remaining roughly the same (increase of less than 1 per mil). In the following we will compare these calculated and predicted pressure fields, to conclude whether there is a significant benefit to the latter approach.

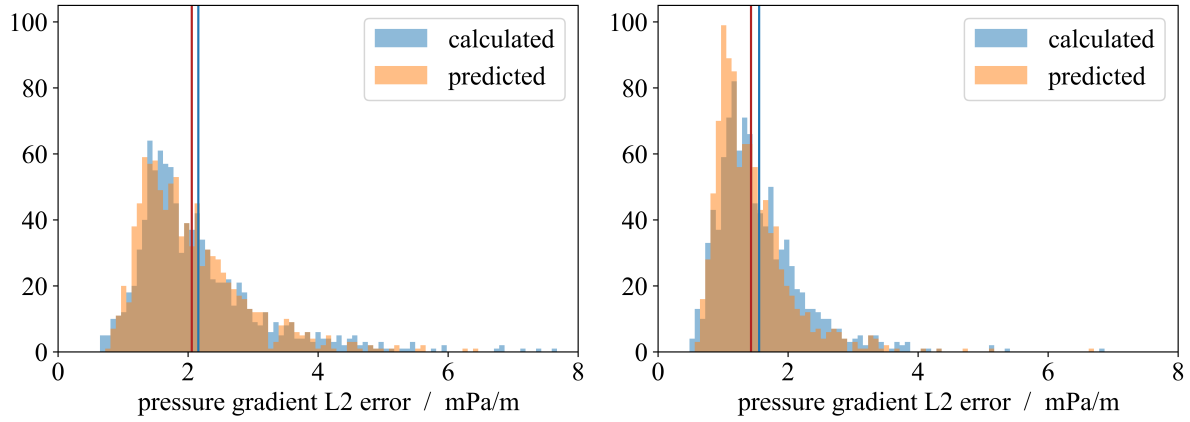
To evaluate the momentum equation, we again discretize the derivatives using a central difference, however this time the expression is slightly more involved. We get

$$\begin{aligned}\frac{1}{\rho}\nabla p =& \frac{\nu}{h^2}\begin{pmatrix} v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{i,j} \\ w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1} - 4w_{i,j} \end{pmatrix} \\ & -\frac{1}{2h}\begin{pmatrix} v_{i,j}(v_{i+1,j} - v_{i-1,j}) + w_{i,j}(v_{i,j+1} - v_{i,j-1}) \\ v_{i,j}(w_{i+1,j} - w_{i-1,j}) + w_{i,j}(w_{i,j+1} - w_{i,j-1}) \end{pmatrix}.\end{aligned} \tag{5.3}$$

For obtaining the gradient of an existing pressure field, the calculation is, of course, much simpler:

$$\nabla p = \frac{1}{2h}\begin{pmatrix} p_{i+1,j} - p_{i-1,j} \\ p_{i,j+1} - p_{i-1,j-1} \end{pmatrix}. \tag{5.4}$$

Again, we avoid boundary effects by setting all points either on our in direct neighborhood to the boundary to zero. To evaluate the accuracy of a gradient field, we compare



(a) GAN **(b)** L1

Figure 5.13: Histograms of the average L2 error of the gradient pressure field w.r.t. the simulation data on the test set. The plots compare the results of calculating pressure from the velocity field prediction versus training the network to explicitly output a pressure prediction. All models were trained for 50000 steps.

it to that obtained from the simulated pressure field. Here again we measure in the euclidean norm, i.e. the L2 error, and average over the whole domain.

The error distribution obtained this way on the test set is shown in Figure 5.13, for both model types. The data clearly show that the difference in accuracy gained from including the pressure in the training is minimal. Moreover we see that the L1 model again performs better in this metric, which is unsurprising given given the results from Section 5.1.

While there appears to be little benefit of including pressure in the training, it should be noted that it did not appear to have any significant downsides, either in terms of accuracy in predicting velocity or in terms of computational effort. Therefore we can conclude that both methods of calculating pressure are equally as viable. This also shows that the error made due to neglecting the Reynolds stress is small.

5.8. Training Data Utilization

Another relevant question to inform a judgment on which type of model is superior for this application is, which can learn better from a limited number of training examples. Besides interest in the comparison, this is also crucial information if one set out to train this type of reduced order model for an end-user application. For this purpose, we trained baseline GAN and L1 models on datasets with the following numbers of samples: $\{250, 500, 1000, 3000, 6000, 9000\}$. The results are plotted in Figure 5.14. The models were trained for 50 000 steps. The results show that for a small dataset, the difference in the average test set L2 error is relatively small. However, the GAN already plateaus after 6000 steps, while the L1 trained model continues to benefit from an increased training set size. The fact that this convergence behavior is opposite of that seen in Figure 5.1 in Section 5.1 is most likely no coincidence: we can safely assume that the number of training steps needed until the learning curve plateaus increases with the size of the training set (in other words, the larger the training set the more and the longer can be learned from it). Since it takes more steps to reach this

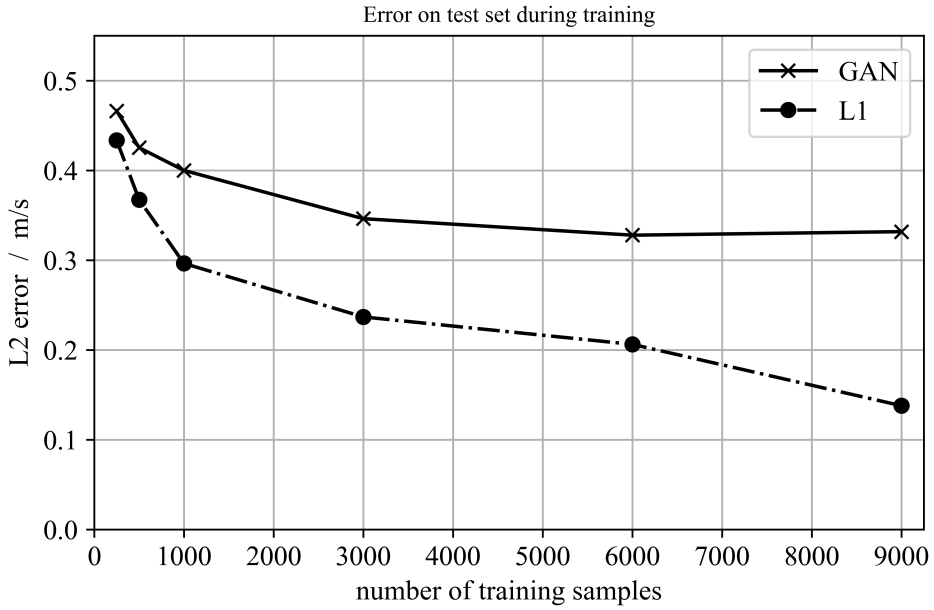


Figure 5.14: Plot showing L2 error performance on test set as a function of the number of samples in the training set. Models trained for 50 000 steps.

Table 5.2: Performance statistics (L2 error and mean continuity residual) on the three data sets for five models with different weighting in the objective function. All models trained for 50 000 steps. Boldface values correspond to best performance in each metric (column).

		L2 error			continuity residual		
λ_{GAN}	λ_{L1}	test	"curved"	"double"	test	"curved"	"double"
1	0	0.33	0.52	0.53	0.13	0.15	0.23
1	1	0.22	0.37	0.51	**0.12**	**0.13**	**0.20**
1	10	0.16	0.30	0.41	0.12	0.14	0.23
1	100	0.14	**0.28**	0.39	0.13	0.14	0.22
0	1	**0.14**	0.30	**0.37**	0.22	0.24	0.46

convergence with adversarial training, the GAN model should be expected to benefit less from increasing the number of training samples if the number of training steps is kept constant, which is exactly what the data show.

5.9. Combined Training

Up to this point we have only compared models that were either trained only adversarially or exclusively on ground truth. However as noted in Section 4.2, the pix2pix framework allows for training with a combination of both types of losses. In this section, we will examine the performance of three models trained in this manner with parameters $\lambda_{\text{L1}} \in \{1, 10, 100\}$ respectively, while λ_{GAN} is kept constant at 1. The particular choice of these parameters was informed by checking the average loss during training the previously discussed GAN & L1 models, the former of which was about one order of magnitude larger than the latter.

Table 5.2 shows the average L2 error and continuity residual for the three hybrid as well as the two baseline models. The data show that indeed, this combined training

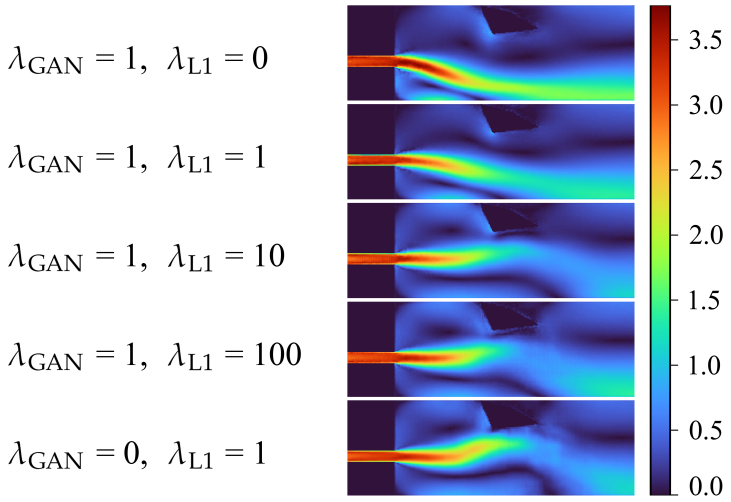


Figure 5.15: Flow speed predictions of different models on a test set sample. The models differ by the weighting of terms in their objective function. All models were trained for 50 000 steps. See Figure 5.9 for geometry & simulation result.

regime can, to some degree, combine the advantages of both model types. In particular the models trained with $\lambda_{L1} = 10$ and $\lambda_{L1} = 100$ produce L2 errors similar to the pure L1 model while retaining the low continuity residual of the pure GAN model.

A qualitative look at edge case examples confirms the hypothesis that the weighted combination of the different loss terms can be used to in some sense 'interpolate' between the models. Figure 5.15 shows the outputs of all five models on a difficult geometry from the test set, the same as plotted in Figure 5.9. It is difficult because it is unclear whether or not the small obstacle at the upper wall is enough to prevent flow attachment by blocking the back-flow. There is a clear trend recognizable in the series, with samples towards the left (lower λ_{L1}) having an increasingly confident and smooth appearance that does not take into account potentially blocked back-flow, similar to the behavior shown in Figure 5.4. Furthermore it appears, that the greater the relative weight of the L1 loss, the stronger the model takes into account the possibility of a lack of wall attachment.

5.10. Modified Learning Rates

The data discussed in Section 5.6 indicated an imbalance in the two-player game that is the GAN training. In all analyzed cases, the data showed that the discriminator was able to clearly distinguish between trained and generated. As discussed in Section 2.3.3, such an imbalance usually leads to sub-optimal results due to slow training convergence. One potential remedy is increasing the learning rate of the generator relative to that of the discriminator. To test this approach, three additional experiments were performed, with generator learning rates twice, four times or eight times greater than that of the discriminator. Table 5.3 compares between these three as well as the baseline by listing the most important performance metrics on the three test datasets.

For most of the data, there is no clear trend discernible, making it difficult to draw strong conclusions. For example, as expected, the 2× model and the 8× model both significantly outcompete the baseline in terms of L2 error on all three data sets, with

Table 5.3: Performance comparison between GAN models trained with different generator learning rates, measured relative to the discriminator learning rate. Metrics include: L2 error, mean continuity residuals, Pearson correlation between discriminator output and L2 error as well as share of samples that could not be correctly classified based on discriminator output. Best results in each category marked in boldface. For correlation, the largest value in terms of magnitude is taken as 'best'. For belief overlap, the largest value is taken as best, since it is closer to equilibrium. The models were trained for 50 000 steps.

	relative generator learning rate	1	2	4	8
L2 error / $\mathrm{m\,s^{-1}}$	test set	0.332	0.241	0.448	**0.221**
	"curved" set	0.521	0.407	0.530	**0.394**
	"double" set	0.531	0.503	0.549	**0.495**
residuals / $\mathrm{s^{-1}}$	test set	0.128	**0.117**	0.172	0.119
	"curved" set	0.148	**0.131**	0.172	0.134
	"double" set	0.225	**0.193**	0.209	0.194
correlation	test set	-0.079	0.181	**-0.452**	0.040
	"curved" set	**-0.131**	-0.046	-0.053	-0.114
	"double" set	-0.416	**-0.552**	-0.422	-0.595
belief overlap / %	test set	0.00	**12.1**	6.10	9.80
	"curved" set	0.70	**40.5**	20.3	18.5
	"double" set	0.00	0.80	0.90	**1.30**

8× producing the best results. However unexpectedly the 4× model actually performs worse. Nevertheless, it is a remarkable result, with the 8× model improving the test set result by 33 % (for comparison, the L1 result was 58 % smaller compared to the baseline, see Section 5.1). Looking at the residual data, we see a similar pattern, with the 2× and 8× models outperforming the baseline, but the 4× performing significantly worse than baseline, at least on the two larger datasets. The data on correlation between the L2 error and the discriminator output holds even more peculiar results. On the two-obstacle set all models exhibit a strong negative correlation, but on the test set only the 4× model does. On the "curved" set in turn, the same model shows almost no correlation, much less than the baseline.

The last bracket includes a measure quantifying how well the discriminator can distinguish between simulated and generated samples. For this purpose, we divide the interval $[0, 1]$ into 100 equal sized bins just like for plotting a histogram, and calculate the number of overlapping samples between the distributions. I.e., if there are ten samples from one distribution and three from the other in the same bin, it counts as three towards the total overlap count. We refer to this quantity as belief overlap and a larger value should correspond to an improved balance between generator and discriminator. As expected, we see that in each case, the increased generator learning rate results in elevated numbers of overlapping samples, however again there is no clear trend beyond that observation.

Table 5.4: Performance comparison between GAN models with modified discriminator architectures, including: L2 error, mean continuity residuals, Pearson correlation between discriminator output and L2 error as well as share of samples that could not be correctly classified based on discriminator output. Best results in each category marked in boldface. For correlation, the largest value in terms of magnitude is taken as 'best'. For belief overlap, the largest value is taken as best, since it is closer to equilibrium. The models were trained for 200 000 steps.

	dataset	baseline	2x kernel size	+2 layers
L2 error / $\mathrm{m\,s^{-1}}$	test	0.263	0.325	**0.251**
	"curved"	0.438	0.408	**0.381**
	"double"	**0.495**	0.531	0.507
residuals / $\mathrm{s^{-1}}$	test	**0.110**	0.190	0.145
	"curved"	**0.130**	0.184	0.161
	"double"	**0.185**	0.214	0.225
correlation	test	-0.058	**-0.311**	-0.005
	"curved"	0.028	**0.046**	-0.028
	"double"	**0.491**	-0.437	-0.265
belief overlap / %	test	0.2	**100.0**	58.2
	"curved"	1.1	**100.0**	82.4
	"double"	0.7	**10.0**	5.3

5.11. Modified Discriminator

While a large parameter study is left for future work, we did test the effects of modifying two important parameter in the discriminator architecture. The first is the kernel size for the convolutional operations, which was increased from 4×4 to 8×8, increasing the total number of trained parameters approximately fourfold. The second is the number of down-convolutional layers, which was increased from three to five. Here the pattern of doubling the number of channels at every layer is continued, i.e. the last hidden layer has 2048 channels. This comes with a dramatic increase in the number of trained parameters by a factor of approx. 16, resulting in a total of 44.7 million, which is similar to the generator's 48.6 million. Moreover, it increases the theoretical receptive field to 286×286, wherefore each value in the output array of dimensions 6×1 is causally linked to each pixel in the input data.

Similar to the previous section, the most important performance statistics are listed in Table 5.4. For this comparison, the models were trained for 200 000 steps, on the rationale that additional layers might slow down convergence. Again, the results are rather ambiguous. The deeper discriminator (additional layers) appears to result in slightly more accuracy on the single obstacle datasets, especially the "curved" set. The discriminator with larger kernels on the other hand results in significantly worse accuracy on the test set. Both modified discriminators lead to larger continuity residuals compared to the baseline on all sets. Interestingly, both modifications also lead to larger ambiguity in the discriminator output, as the data on overlap samples show (see previous section on how these are calculated). This is surprising as in theory, it would indicate a worse relative performance of the discriminator, despite the large increase in model size. Moreover, from a theoretical perspective one might expect the larger belief overlap to correspond with improved accuracy, since at equilibrium (i.e. the optimum) the discriminator would show complete ambiguity. However, this is

not borne out by the data, especially for the discriminator with increased kernel size. The data also do not show a clear pattern in terms of correlation between L2 error and discriminator belief. Therefore we cannot conclude that the tested modifications to the discriminator architecture provide a clear improvement.

6

Conclusions

GAN models, as the name implies, were originally invented to tackle so-called generative problems. The goal with this type of problem is generally to train the neural network to match a target distribution.As explained in Section 2.3.3, generative problems are difficult to solve with regular neural networks, because formulating a suitable cost function for measuring performance is highly non-trivial. The GAN framework overcomes this problem by essentially optimizing this cost function as part of the training. which is realized by training an observer (discriminator) alongside the generator. This explains their particular strength at generative task. Given that the entire proposition of GANs is a change in cost function, whether or not adversarial training can be expected to provide superior results thus ultimately depends on what concept of error is used. GAN training is focused on producing outputs with a 'realistic' appearance (in the sense of the training set), rather than the best possible accuracy. In other words, adversarial training teaches the generator to produce outputs that 'look right' to an observer familiar with the dataset.

Indeed, this is exactly what our experimental data shows. In terms of accuracy to the ground truth velocity field (in an L2 sense), none of the GAN models reached the performance of an equivalent generator trained directly on ground truth (using an L1 loss). This is true on unseen data from the training distribution, as well as on two additional data sets designed to test generalization to unfamiliar types of geometries (see Sections 5.1, 5.3). The average L2 error rate of the GAN model was typically between 30 % and 100 % larger than that of the equivalent L1 trained model (depending on the specific comparison), only producing a superior result on about 10 % of the samples. Qualitatively speaking, the L1 model appeared much more successful at reproducing the finer details of flow mechanics, such as exactly where flow around an obstacle detaches.

On the other hand, the data also show that the velocity fields produced by GAN generators exhibit significantly lower average continuity residuals, by approx. $40-50\%$, compared to L1 trained models (see 5.4). This again is exactly the type of distribution-matching behavior expected from GANs. After all, if the residuals would be large the discriminator could easily learn to identify generated samples based on it. In order to test whether a similar result can be achieved without adversarial training, we also experimented with adding a cost term specifically penalizing the continuity residual

(i.e. a physics-based loss) to the L1 training, which helped improve the residual almost to the level achieved by the GAN model.

Different to typical generative problems, in flow problems there is normally only a single unique solution. However, the training set for this study was specifically designed to include cases where the opposite holds, i.e. there is more than one possible solution for a single geometry. These scenarios are of particular interest for the comparison, because they illustrate the differences between adversarial and regular training much more clearly. Specifically, we analyzed a flow bifurcation, where the flow would randomly attach to the upper or lower part of the domain. As expected, the GAN essentially 'picked a side', whereas the L1 trained model produced a sort of superposition of the two possible outcomes (see Section 5.2). In such cases, none of the two outcomes is obviously superior; it completely depends on how performance is measured. The L1 trained prediction could be considered worse since it is unphysical and would never occur in the data set. On the other hand however, it represents the uncertainty between the two possibilities, which could be very useful in many applications.

Overall, the L1 trained model was also shown to be more attentive to subtle flow dynamical effects, such as an obstacle blocking back-flow leading to a change in flow attachment (see again Section 5.2). Another problem with the GAN models was that the generator would occasionally produce a large artifact, as detailed in Section 5.5. However, the issue occurred infrequently ($< 1\%$ of samples) and disappeared almost entirely given more training. Nevertheless it is an interesting observation that warrants further study.

Beyond comparing the two different types of models, we also investigated combining them by training on a sum of both cost functions. The results show that this strategy can be used to successfully interpolate between the types, both in terms of qualitative outcomes as well as statistical performance (see Section 5.9). In this way, it is to some degree possible to combine both model's advantages (in particular: low L2 error and low residual).

An analysis of the discriminator output showed, that in almost every case it allowed for a correct classification of a sample as simulated or generated (see Section 5.6). The fact that the generator appeared unable to 'fool' the discriminator hinted at an imbalance in the adversarial training, with a dominating discriminator. Based on this result, we tested increasing the learning rate of the generator for improved balance. This did indeed produce some better performing models, although the results were ambiguous without a clear trend discernible (see Section 5.10). Moreover, we investigated whether the discriminator can be used to quantify the uncertainty of the prediction. To test this, we calculated the statistical correlation between L2 error and discriminator output. However, the results did not show a clear trend, with sometimes positive, sometimes negative, and most of the time no significant correlation (see Section 5.6, 5.9).

While most of the work was focused on predicting the velocity field, we also looked into predicting pressure as well. Based on the collected experimental data, we were able to conclude that despite the coarse image grid resolution, it is sufficient to only predict the velocity field and use that to calculate the pressure field by virtue of the Navier-Stokes momentum equation (neglecting Reynolds stress). Directly predicting

pressure is also feasible without much extra effort, but only showed a negligible improvement (see sections 5.7).

Another important experimental observation is, that it generally took much more training steps for GAN models to reach convergence during training. This can be attributed to the more complicated training process involving the evolving dynamics between generator and discriminator (see Section 5.1). Similarly, with GANs the data show a larger spread in model outcomes due to the randomly initialized parameters. Moreover, for a fixed number of training steps, the L1 trained model was able to benefit more from increasing the size of the training set (see Section 5.8).

In summary, model outcomes are consistent with how they were trained, or more specifically, they excel at precisely what the cost function they were trained on measures. Whether or not a GAN is the right choice for building a reduced order ML fluid model ultimately depends on how success is to be measured. For example, if one is interested in creating visual effects for film, then a GAN model is a promising candidate. If instead the application is scientific in nature, then most likely the lower average error of an L1 trained model will be desirable. Combined training can also be a good option, with results showing that adding a small GAN contribution to an L1 cost function can act as a sort of regularization, increasing apparent realism significantly without compromising on accuracy. However, the increase in computational time of more than 50 % in our tests must also be taken into account, especially given that there are alternatives with potentially similar effects such as adding physics-based loss as regularization.

In light of these results, the choice of using GANs in some earlier published work on NN-based fluid flow prediction, such as in [54] for predicting pedestrian level wind speeds in urban design, could be due for reconsideration. Moreover, it suggests that the improvement seen by the authors of [26] when using a GAN over their baseline model was not due to the effect of adversarial training, but rather to using a different generator architecture.

In terms of further work, the scope of this thesis did not include a large optimization study to find the best hyperparameter choices, which would certainly be a relevant object of study. As the data presented in Section 5.10 show, the balance between discriminator and generator could be of particular interest for further investigation. However, the tests that were conducted on modified discriminator architectures show that increasing model capacity does not necessarily improve results, and that results are often not consistent with theoretical expectations (see Section 5.11). It could also be interesting to further investigate the behavior of GANs during hybrid training for use as a type of regularization.

Looking ahead, moving from proof of concept towards application will require a number of important changes and extension to the models considered here. For many types of applications, the addition of information on boundary conditions as input to the model (such as in [26]) would be crucial for making the model more generally applicable. This also enables and necessitates the expansion of the training set to include a much more varied set of flow examples (here it will be helpful to consider the data on training performance as a function of dataset size shown in Section 5.8). This should also inevitably instill the proper invariances inherent to the underlying equations (translational, rotational etc.) into the network's representations. In order

to enable the model to also work in different Reynolds regimes, adding inputs for flow-relevant global parameters, such as scale and viscosity, could be an interesting option for further study. Other modifications could include increasing resolution or a extending the model to work with 3D-problems.

In order properly evaluate the true utility of the types of machine learning models we considered as reduced order fluid models, further research is required. For this, it will be necessary to compare outcomes and computational effort as fairly as possible with existing, conventional reduced order models.

Bibliography

[1] Md Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: (Mar. 2018). arXiv: 1803.01164 [cs.CV] (cit. on p. 9).

[2] Nathan Baker et al. *Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence*. Tech. rep. Feb. 2019. DOI: 10.2172/1478744 (cit. on p. 10).

[3] Andrea Beck, David Flad, and Claus-Dieter Munz. "Deep neural networks for data-driven LES closure models". In: *Journal of Computational Physics* 398 (Dec. 2019), p. 108910. DOI: 10.1016/j.jcp.2019.108910 (cit. on p. 20).

[4] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: (May 2020). arXiv: 2005.14165 [cs.CL] (cit. on p. 12).

[5] Ewen Callaway. "'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures". In: *Nature* 588.7837 (Nov. 2020), pp. 203–204. DOI: 10.1038/d41586-020-03348-4 (cit. on p. 1).

[6] Prafulla Dhariwal and Alex Nichol. "Diffusion Models Beat GANs on Image Synthesis". In: (May 2021). arXiv: 2105.05233 [cs.LG] (cit. on p. 16).

[7] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: (Mar. 2016). arXiv: 1603.07285 [stat.ML] (cit. on pp. 15, 16).

[8] Ricard Durall et al. "Combating Mode Collapse in GAN training: An Empirical Analysis using Hessian Eigenvalues". In: (Dec. 2020). arXiv: 2012.09673 [cs.LG] (cit. on p. 19).

[9] Matthias Eichinger, Alexander Heinlein, and Axel Klawonn. "Surrogate convolutional neural network models for steady computational fluid dynamics simulations". In: *Electronic Transactions on Numerical Analysis (ETNA)* 56 (2022), pp. 235–255. DOI: 10.1553/etna_vol56s235. URL: https://epub.oeaw.ac.at/?arp=0x003d4c21 (cit. on p. 20).

[10] Amir Barati Farimani, Joseph Gomes, and Vijay S. Pande. "Deep Learning the Physics of Transport Phenomena". In: (Sept. 2017). arXiv: 1709.02432 [cs.LG] (cit. on pp. 2, 21).

[11] Joel H. Ferziger, Milovan Perić, and Robert L. Street. *Computational Methods for Fluid Dynamics*. Springer International Publishing, 2020. DOI: 10.1007/978-3-319-99693-6 (cit. on pp. 7, 9).

[12] William K. George. *Lectures in Turbulence for the 21st Century*. 2013. URL: http://www.turbulence-online.com/Publications/Lecture_Notes/Turbulence_Lille/TB_16January2013.pdf (cit. on pp. 6, 8).

[13] C. Geuzaine and J.-F. Remacle. "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79(11) (2009), pp. 1309–1331 (cit. on p. 25).

[14] Ian Goodfellow, Joshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press Ltd, 2016. 800 pp. (cit. on pp. 9, 10, 12–15).

[15] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014) (cit. on pp. 16, 18, 21).

[16] Xiaoxiao Guo, Wei Li, and Francesco Iorio. "Convolutional Neural Networks for Steady Flow Approximation". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Aug. 2016. DOI: 10.1145/2939672.2939738 (cit. on p. 20).

[17] W. Malalasekera H. Versteeg. *An Introduction to Computational Fluid Dynamics*. Second Edition. Pearson Education (US), Feb. 2007. 520 pp. ISBN: 0131274988 (cit. on pp. 8, 9).

[18] Alexander Heinlein et al. "Machine Learning in Adaptive Domain Decomposition Methods - Predicting the Geometric Location of Constraints". In: *SIAM Journal on Scientific Computing* 41.6 (Jan. 2019), A3887–A3912. DOI: 10.1137/18m1205364 (cit. on p. 20).

[19] Kai Hu et al. "Text to Image Generation with Semantic-Spatial Aware GAN". In: (Apr. 2021). arXiv: 2104.00567 [cs.CV] (cit. on p. 16).

[20] Xun Huang et al. "Multimodal Conditional Image Synthesis with Product-of-Experts GANs". In: (Dec. 2021). arXiv: 2112.05130 [cs.CV] (cit. on p. 16).

[21] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (2015) (cit. on p. 13).

[22] Haris Iqbal. *HarisIqbal88/PlotNeuralNet v1.0.0*. 2018. DOI: 10.5281/ZENODO.2526396 (cit. on pp. 16, 29).

[23] Phillip Isola. *Adding script to calculate netD receptive field sizes*. GitHub. URL: https://github.com/phillipi/pix2pix/blob/master/scripts/receptive_field_sizes.m (cit. on p. 30).

[24] Phillip Isola et al. "Image-to-Image Translation with Conditional Adversarial Networks". In: (Nov. 2016). arXiv: 1611.07004 [cs.CV] (cit. on pp. 16, 17, 28).

[25] Changlin Jiang and Amir Barati Farimani. "Deep Learning Convective Flow Using Conditional Generative Adversarial Networks". In: (May 2020). arXiv: 2005.06422 [physics.flu-dyn] (cit. on pp. 2, 21).

[26] Haoliang Jiang et al. "StressGAN: A Generative Deep Learning Model for Two-Dimensional Stress Distribution Prediction". In: *Journal of Applied Mechanics* 88.5 (Feb. 2021). DOI: 10.1115/1.4049805 (cit. on pp. 2, 21, 52).

[27] C. S. Jog. *Fluid Mechanics*. Cambridge University Press, 2015. DOI: 10.1017/cbo9781316134030 (cit. on p. 6).

[28] Tero Karras et al. "Alias-Free Generative Adversarial Networks". In: (June 2021). arXiv: 2106.12423 [cs.CV] (cit. on p. 16).

[29] Diederik P Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *ICLR*. Dec. 2014. arXiv: 1312.6114 [stat.ML] (cit. on p. 16).

[30] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 2014). arXiv: 1412.6980 [cs.LG] (cit. on p. 30).

[31] I. E. Lagaris, A. Likas, and D. I. Fotiadis. "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations". In: (May 1997). DOI: 10.1109/72.712178. arXiv: physics/9705023 [physics.comp-ph] (cit. on p. 20).

[32] Quang Tuyen Le and Chin Chun Ooi. "Surrogate Modeling of Fluid Dynamics with a Multigrid Inspired Neural Network Architecture". In: (May 2021). arXiv: 2105.03854 [physics.flu-dyn] (cit. on p. 16).

[33] Leonid P Lebedev and Michael J Cloud. *Tensor Analysis*. WORLD SCIENTIFIC, Apr. 2003. DOI: 10.1142/5265 (cit. on p. 5).

[34] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: (Sept. 2016). arXiv: 1609.04802 [cs.CV] (cit. on p. 17).

[35] Sangseung Lee and Donghyun You. "Data-driven prediction of unsteady flow over a circular cylinder using deep learning". In: *Journal of Fluid Mechanics* 879 (Sept. 2019), pp. 217–254. DOI: 10.1017/jfm.2019.700 (cit. on p. 22).

[36] OpenCFD Ltd. *User Guide 7.4: Sampling data*. URL: https://www.openfoam.com/documentation/user-guide/7-post-processing/7.4-sampling-data (cit. on p. 26).

[37] Nils Margenberg et al. "A neural network multigrid solver for the Navier-Stokes equations". In: (Aug. 2020). DOI: 10.1016/j.jcp.2022.110983. arXiv: 2008.11520 [physics.comp-ph] (cit. on p. 20).

[38] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-16874-6 (cit. on p. 9).

[39] T. Mullin et al. "Bifurcation phenomena in the flow through a sudden expansion in a circular pipe". In: *Physics of Fluids* 21.1 (Jan. 2009). DOI: 10.1063/1.3065482 (cit. on p. 23).

[40] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077 (cit. on p. 14).

[41] OpenCFD Ltd. *simpleFOAM documentation*. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-applications-solvers-incompressible-simpleFoam.html (cit. on p. 25).

[42] Wojciech Ożański. "The Lagrange multiplier and the stationary Stokes equations". In: *Journal of Applied Analysis* 23.2 (Dec. 2017). DOI: 10.1515/jaa-2017-0017 (cit. on p. 6).

[43] Ronald L. Panton. *Incompressible Flow*. John Wiley & Sons, Inc., July 2013. DOI: 10.1002/9781118713075 (cit. on p. 5).

[44] S.V Patankar and D.B Spalding. "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows". In: *International Journal of Heat and Mass Transfer* 15.10 (1972), pp. 1787–1806. ISSN: 0017-9310. DOI: https://doi.org/10.1016/0017-9310(72)90054-3. URL: https://www.sciencedirect.com/science/article/pii/0017931072900543 (cit. on p. 9).

[45] Protein Structure Prediction Center, University of California, Davis. URL: https://predictioncenter.org/casp14/zscores_final.cgi (cit. on p. 2).

[46] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: (Nov. 2015). arXiv: 1511.06434 [cs.LG] (cit. on p. 19).

[47] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045 (cit. on p. 20).

[48] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations". In: (Nov. 2017). arXiv: 1711.10566 [cs.AI] (cit. on p. 20).

[49] openfoam repository. *backwardFacingStep2D tutorial*. URL: https://develop.openfoam.com/Development/openfoam/-/tree/master/tutorials/incompressible/simpleFoam/backwardFacingStep2D (cit. on p. 25).

[50] Danilo Jimenez Rezende and Shakir Mohamed. "Variational Inference with Normalizing Flows". In: *ICML*. May 2015. arXiv: 1505.05770 [stat.ML] (cit. on p. 16).

[51] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28 (cit. on pp. 16, 30).

[52] Karl Rupp. *42 Years of Microprocessor Trend Data*. Feb. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ (cit. on p. 9).

[53] Tim Salimans et al. "Improved Techniques for Training GANs". In: (June 2016). eprint: 1606.03498 (cs.LG) (cit. on p. 19).

[54] Aleksandra Sojka Sarah Mokhtar and Carlos Cerezo Davila. "Conditional Generative Adversarial Networks for Pedestrian Wind Flow Approximation". In: *Proceedings of SimAUD*. 2020, pp. 25–27 (cit. on pp. 2, 22, 52).

[55] I. Bohachevsky Sergei Godunov. "Finite difference method for numerical computation of discontinuoussolutions of the equations of fluid dynamics". In: *Matematičeskij sbornik* 47 (89).3 (1959), pp. 271–306 (cit. on p. 9).

[56] Robert Service. "'The game has changed.' AI triumphs at solving protein structures". In: *Science* (Nov. 2020). DOI: 10.1126/science.abf9367 (cit. on p. 1).

[57] B. Siddani et al. “Machine learning for physics-informed generation of dispersed multiphase flow using generative adversarial networks”. In: *Theoretical and Computational Fluid Dynamics* 35.6 (Oct. 2021), pp. 807–830. DOI: 10.1007/s00162-021-00593-9 (cit. on pp. 2, 21).

[58] TensorFlow Developers. *TensorFlow*. 2022. DOI: 10.5281/ZENODO.4724125 (cit. on p. 28).

[59] The OpenFOAM Foundation. URL: https://openfoam.org/ (cit. on p. 25).

[60] Bing Xu et al. “Empirical Evaluation of Rectified Activations in Convolutional Network”. In: (May 2015). arXiv: 1505.00853 [cs.LG] (cit. on p. 14).

[61] Jun-Yan Zhu et al. “Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”. In: (Mar. 2017). arXiv: 1703.10593 [cs.CV] (cit. on p. 16).

A

OpenFOAM settings

Schemes (fvSchemes-file)

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  9
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
format      ascii;
class       dictionary;
location    "system";
object      fvSchemes;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

ddtSchemes
{
default         steadyState;
}

gradSchemes
{
default         Gauss linear;
}

divSchemes
{
default         none;
div(phi,U)      bounded Gauss linearUpwind grad(U);
div(phi,k)      bounded Gauss limitedLinear 1;
```

```
div(phi,epsilon) bounded Gauss limitedLinear 1;
div(phi,omega)  bounded Gauss limitedLinear 1;
div(phi,v2)     bounded Gauss limitedLinear 1;
div((nuEff*dev2(T(grad(U))))) Gauss linear;
div(nonlinearStress) Gauss linear;
}

laplacianSchemes
{
default         Gauss linear corrected;
}

interpolationSchemes
{
default         linear;
}

snGradSchemes
{
default         corrected;
}

wallDist
{
method meshWave;
}


// ************************************************************************* //
```

Solvers (fvSolution-file)

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  9
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
format      ascii;
class       dictionary;
location    "system";
object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
p
{
solver          GAMG;
tolerance       1e-07;
relTol          0.1;
smoother        GaussSeidel;
}

"(U|k|epsilon|omega|f|v2)"
{
solver          smoothSolver;
smoother        symGaussSeidel;
tolerance       1e-06;
relTol          0.1;
}
}

SIMPLE
{
nNonOrthogonalCorrectors 0;
consistent      yes;

residualControl
{
p               1e-2;
U               2e-4;
"(k|epsilon|omega|f|v2)" 2e-3;
```

```
}
}

relaxationFactors
{
equations
{
U               0.5;
".*"            0.5;
}
}


// ************************************************************************* //
```