Reduced Order Models for Fluid Flow With Generative Adversarial Networks (GANs)

by

Mirko Kemna

Interim Thesis and Literature Report

Student Number: 5606896

Faculty Electrical Engineering, Mathematics,
and Computer Science

Delft University of Technology

Contents

Chapter 1

Introduction

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. 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, building efficient computational models remains a challenge for many real world applications. Now, new AI 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 sections of reality.

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. Moreover, 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 have 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 computing power necessary. Thus, scientists have been 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 managed to achieve results described as "transformational"[1], handily outclassing the previously leading models.[2]

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

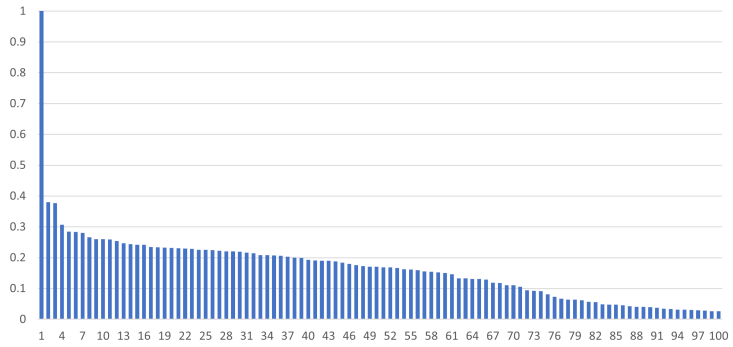

Figure 1.1: Performance of best 100 entries in 14$^{\text{th}}$ CASP protein folding challenge in 2020, relative to the scoring of AlphaFold 2. Data taken from CASP website [3].

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

In this research project, we will investigate training a generative adversarial artificial neural network (GAN) to act as a reduced order model for fluid modeling problems. The field of computational fluid dynamics (CFD) is crucial for a plethora of technical and scientific applications, including energy generation, transportation, geoscience and medicine. If successful, this would be especially useful for design space exploration in engineering contexts, such as shape optimization.

This report is intended to provide three things. First, an overview of the most relevant background, both in terms of fluid modeling and machine learning. Second, a survey of the state of the art and relevant research in the area of interest. And third a presentation of the initial experiments conducted in preparation for the research project, as well as an outline of the research to come.

Chapter 2

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 topic of study in physics and mathematics. Today, computational fluid modeling (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 will present a brief outline of the mathematical theory of fluids. We will 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 will be given in Section 5.1.

2.1 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

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

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 following 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, i.e. the derivative in the Lagrangian reference frame. 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 newtons 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 outside forces acting on 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 $\mathbf{f}$, such as gravity. The divergence of a second order tensor is taken row-wise [4].

To close the 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: 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, in particular water and air (also known as Newtonian fluids). If we assume further that the incompressible fluid has isotropic properties, we arrive at the following relation, written in index notation [5, 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, or more 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 obtain the momentum equation we have been striving for:

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

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

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

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

$$\mu\, \Delta \mathbf{u} - \rho(\mathbf{u} \cdot \nabla)\mathbf{u} = \nabla p. \tag{2.8b}$$

This is a second order, nonlinear system of partial differential equations (PDEs) with elliptic character. 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, the second terms represents the viscous forces, and the right hand side represents an internal source in the form of the pressure gradient. Mathematically, the pressure can be regarded as a Lagrange multiplier enforcing the continuity constraint (2.8a) in the variational formulation of the equations [6].

2.2 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

$$\mathrm{Re} = \frac{\rho U L}{\eta} \tag{2.10}$$

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 [7, page 53-54]. The full dimensionless system of equation 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 completely characterized by the Reynolds number.

2.3 Turbulence Modeling

A particular challenge in computational modeling of fluids is posed by turbulent flows. Turbulence refers to the fleeting chaotic structures that occur on small scales, down to the microscopic (see Kolmogorov microscale [8]), in flows with high Reynolds Numbers. Despite the size, they nevertheless can contribute significantly to transport of quantities such as momentum in 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. 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.

One framework for modeling turbulence are the so-called Reynolds-averaged Navier-Stokes equations (RANS). Reynolds-averaging refers to averaging the equation over a time large enough for the effect turbulent fluctuations to approximately equalize. Applying this to (2.8) gives [9, 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 and $\otimes$ is the outer product. As one can see, the structure of the equation stayed mostly the same, but a new term $\nabla \cdot \tau$ appeared in the momentum equation. The The 2^{nd} order tensor τ is given by

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

where $\mathbf{u}'$ is the instantaneous velocity deviation from the mean, i.e. $\mathbf{u}' = \mathbf{u} - \bar{\mathbf{u}}$. In analogy to the Cauchy stress tensor, it is termed the Reynolds stress tensor.

The goal is to model only the averaged quantities, and thus $\mathbf{u}'$ is an unknown in this equation. This is where a turbulence model is required, which allows us to evaluate the six unknowns of Reynolds stress from the averaged flow (*closure problem*). Early experimental results lead to Boussinesq's hypothesis that net effect of turbulence can be modeled as a flow-dependent increase in viscosity ("eddy viscosity"). Based on this, the typical ansatz for the Reynolds stress tensor is [9, section 10.3.5]:

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

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. The Boussinesq hypothesis reduces the number of unknowns in τ from six down to two turbulence parameters, the eddy-viscosity μ_t and k. Many different closure models have been developed to close the eddy viscosity model. The simplest merely give an algebraic relation between the turbulence parameters and the time averaged flow fields $\bar{\mathbf{u}}$ and $\bar{p}$, but they are too simple to generalize well. 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 $k\epsilon$ model, which introduces the turbulence dissipation rate ϵ and two additional transport equations to close the system. This is the model that will be used for this project, see chapter 5. For more details on this specific model, see [10], and for a comprehensive treatise of the theory of turbulence refer to [8].

Chapter 3

Machine Learning

Enabled by the culmination of sustained exponential growth in computing power over the last decades [11] , the field of machine learning (ML) has made remarkable progress in the last few years. 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) methods are based on some form of machine learning. Moreover, they often rely on 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. [12, page 5]

Ground-breaking results have been achieved across fields such as computer vision, speech and natural language processing as well as synthesis.

Beyond being a very active field of study itself, ML has also been successfully applied to advance scientific frontiers in other areas of research, a discipline referred to as scientific machine learning (SciML). It seeks to incorporate techniques from of machine learning into the field of scientific computing. For a comprehensive overview, see [13]. A survey of works on applying ML to PDE problems will be given in Chapter 4.

Machine learning methods can be separated into two categories, supervised and unsupervised learning. The key difference is that in supervised learning, the training data are labeled, and the goal is generally to map from unseen training 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. [12, page 105]. The focus of this work is on a specific type of machine learning framework known as generative adversarial networks which combines aspects of both paradigms. It will be discussed in the last section of this chapter.

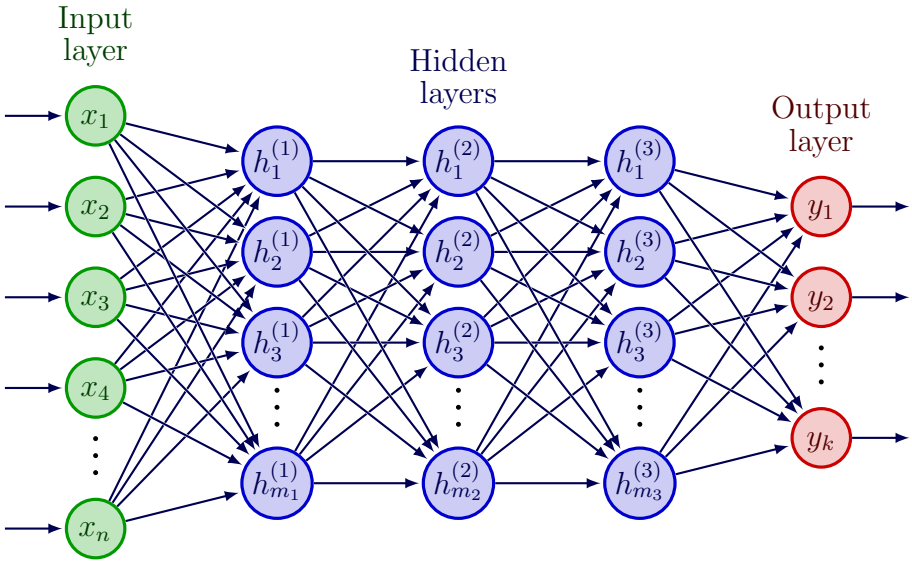

Figure 3.1: A graph visualizing the general structure of an artificial neural network with three hidden layers.

3.1 Artificial Neural Networks

Artificial neuron networks (ANNs, hereafter also simply referred to as neural networks) are 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 artificial neurons, which are arranged in connected layers. An example is visualized in figure 3.1. The most general description of the setting in which neural networks are applied is to model the relation underlying a set of observations (x, y) i.e. to find a way of relating a feature x to a label y. 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. detecting a handwritten digit).

3.1.1 Artificial Neurons

A single neuron can be represented as a mapping from a number of inputs x_i to a single output y. 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 $\mathbf{x}$ and the corresponding weights in a vector $\mathbf{w}$. Together with the bias b we thus write

$$h = f\left(\mathbf{w} \cdot \mathbf{x} + b\right), \tag{3.1}$$

where the resulting output h is called the activation level. The weights and bias parametrize a hyperplane in the input space known as the decision boundary, which is defined by $\mathbf{w} \cdot \mathbf{x} + b = 0$. A geometric interpretation is that the activation function operates on the euclidean distance d of the input point $\mathbf{x}$ to the decision boundary, scaled by the norm of the weight vector:

$$h = f\left(d\,|\mathbf{w}|\right). \tag{3.2}$$

3.1.2 Network Structure

As mentioned before, the neurons in an ANN are arranged in layers. These are typically connected in a sequential order, such that information propagates from input to output, although other variations exist (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 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 size of input and output layer (n and k respectively) is 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 one dimensional structures, then the operation that a layer j performs can be written as

$$\mathbf{z}_j = F\left(\mathbf{W}_j \mathbf{z}_i + \mathbf{b}\right), \tag{3.3}$$

where i is the index of the previous 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 F is the element wise application of an 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 will be explored further in Section 3.2.

The whole network then is essentially a parametrized function $\mathcal{N}$ mapping from an input space X to an output space Y:

$$\begin{aligned} &\mathcal{N} : X \to Y \\ &\hat{\mathbf{y}} = \mathcal{N}(\mathbf{x}; \boldsymbol{\omega}), \end{aligned} \tag{3.4}$$

where $\boldsymbol{\omega}$ represents weights and bias of every neuron in the network, and $\hat{y}$ 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. We will use this in later sections, where we will be dealing with translating from an image of geometry to a flow field image of the same size. We correspondingly take input and output layers as 2-dimensional, i.e.

$$X \subseteq \mathbb{R}^{k \times l}, \quad \subseteq \mathbb{R}^{k \times l}$$

where k and l are the image dimensions. Note that size of input and output layers may also be drastically different; take for instance the task of classifying 256x256 pixel images on the basis of whether or not they depict a dog. In this case, the network will have an input layer with a size of approx. 65 000, but only a single, binary output neuron.

3.1.3 Training

A large neural network can easily have millions of trainable parameters, with the largest containing over 100 billion [14, GPT3 language model]. Our goal is to choose these in such a way that the network can approximate the unknown law underlying our observations. This law could be a simple functional relation, $\mathbf{y}(\mathbf{x})$, but is typically instead modeled in the more general framework of stochastics, 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 $\mathbb{X}$ consists of a number N_T of features $x \in X$ and associated labels $y \in Y$, drawn from the probability distribution $p(\mathbf{x}, \mathbf{y})$. As described in eq. (3.4), the ANN can generate an output $\hat{\mathbf{y}}$ for each feature sample $\mathbf{x}$. Initially, weights are typically semi-randomly initialized based on some heuristic. Obviously, such an untrained network cannot be expected to solve any given problem. 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 $c(\mathbf{y}, \hat{\mathbf{y}})$, which assigns a cost to every combination of y and $\hat{y}$:

$$\begin{aligned} c &: Y \times Y \to \mathbb{R} \\ \text{s.t. } & c(\mathbf{y}, \mathbf{y}) = 0. \end{aligned} \tag{3.5}$$

The goal of the learning process is to minimize the expected cost C with respect to the underlying data distribution. The optimization problem is therefore

$$\min_{\boldsymbol{\omega}} C = \min_{\boldsymbol{\omega}} \mathbb{E}_{p(\mathbf{x},\mathbf{y})} \left[c(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) \right]. \tag{3.6}$$

However, we generally do not know $p(\mathbf{x}, \mathbf{y})$ exactly. Therefore we approximate C by the average training error C_{T} on our training set

$$\min_{\boldsymbol{\omega}} C_{\mathrm{T}} = \min_{\boldsymbol{\omega}} \left(\frac{1}{N_{\mathrm{T}}} \sum_{\mathbb{X}} c(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) \right). \tag{3.7}$$

This optimization problem is typically highly non-convex, and solutions are by no means unique. In fact, just by permutation of neurons, a single fully connected layers of size L multiplicatively contributes $L!$ equivalent solutions. At any rate, using direct methods 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 (w.r.t. the parameters). 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

$$\Delta \boldsymbol{\omega}_i = -\eta \left. \frac{\partial C_{\mathrm{T}}}{\partial \boldsymbol{\omega}_i} \right|_{\boldsymbol{\omega}_0} = -\eta \left. \left(\frac{\partial C_{\mathrm{T}}}{\partial \hat{\mathbf{y}}_j} \frac{\partial \mathcal{N}_j}{\partial \boldsymbol{\omega}_i} \right) \right|_{\boldsymbol{\omega}_0}, \tag{3.8}$$

where $\boldsymbol{\omega}_0$ 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, the initialization deciding over which. For shallow minima, this can be overcome (literally) by adding "momentum" terms to the descent kinetics (see [12, section 8.3]), but the core problem remains. Nevertheless, gradient descent has in practice proven to be a very successful workhorse of machine learning. On this issue, the authors remark in [12, 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 still works well in practice despite this is down to the fact that we are not actually interested in finding a global or even local minima of (3.7), as this solution would most likely not correspond to a good solution of the original problem (3.6) (overfitting, see [12, 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 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 at 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 depend 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. It ensures that across a batch (or mini-batch), the distribution of each activation has zero mean and unit standard deviation [15].

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 [12, page 174]. This is because they avoid the problem of a stretching of the loss landscape w.r.t. shallow layers, which is caused by the vanishing gradient of sigmoid functions for large inputs, while retaining a non-linearity. The general formula for the rectified linear unit is

$$f(z) = \arg\max(z, az),\ a \in [0, 1], \tag{3.9}$$

which is a modification of the original rectified linear unit (ReLU, [16]) with $a = 0$. If $0 < a << 1$, it is also referred to as leaky ReLU [17].

For more information on optimization techniques in neural networks, refer to [12, chapter 8]

3.2 Convolutional Networks

Much of the data encountered in real world problems has quasi-spatial structure, such images or time series like audio recordings. A common way to reflect this in the structure of an ANN is through use of convolutional layers, which can greatly improve the performance of ANN models.

3.2.1 Convolutional Layers

Traditional 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 imageor signal (in the following we will focus on application to images). At each point, the data in the kernel range is weighted by the corresponding kernel elements and summed up. The resulting grid is called a feature map, as it indicates the presence of the feature in the original image. A simple example is edge detection.

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 3.2. However, we typically apply not just one but many kernels on a single layer, and thus we get multiple feature maps, also known as channels, as output. For the

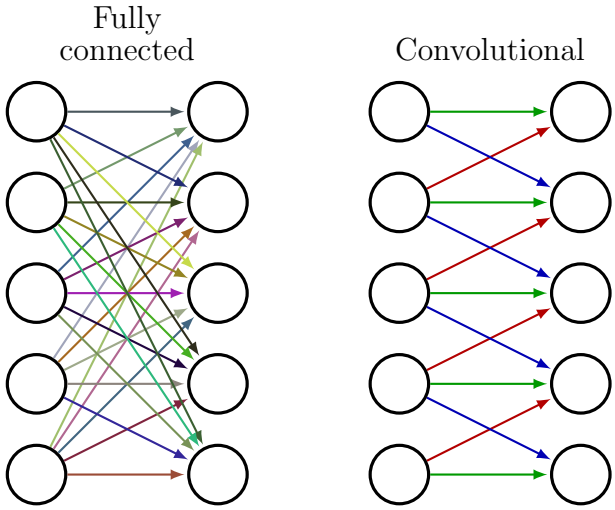

Figure 3.2: Comparison between fully connected and convolutional layer. Same colors indicate shared weights; the fully connected layer has 25 independent weights, while the convolutional layer has only 3.

case of a 2D image, we can therefore represent a convolutional layer as a 3D block, where each sublayer in the block represents the response to a different filter. 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 input and output of vectors, we can write the operation as

$$\mathbf{y}_i = \sum_j \mathbf{C}_{ij}\mathbf{x}_j, \tag{3.10}$$

where $\{\mathbf{x}_i\}_{i=1\ldots I}$ are the I input channels, $\{\mathbf{y}_i\}_{i=1\ldots J}$ 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, with each row containing the elements of the respective kernel. Not by coincidence, this structure is very similar to that of matrices resulting from finite element discretizations of PDE problems, for instance (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 row from $\mathbf{C}_{ij}$, and effectively downsamples the input. For example, a stride of two in x and y direction reduces the rows in $\mathbf{C}_{ij}$ and thereby the length of $\mathbf{y}_i$ by a factor of four. 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 [18]. 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 more details on convolutional layers and their use, see and [12, Chapter 9].

3.2.2 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 down-sampling 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 lower level ones from 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 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. Sometimes fully connected layers are placed at the end of the down-convolutional pipeline to perform nonlinear operations on the detected features.

Consider the example of classifying images based on whether they contain a dog. 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 dog.

For tasks where both input and output have tensorial structure, CNNs commonly use a bottleneck architecture, similar to an autoencoder, as visualized in Figure 3.2.2. It consists of an encoding pipeline, which "featureizes" data, a bottleneck combining the high level features in latent space, and a decoder part that synthesizes the output using transpose convolutions or up-convolutions (sometimes misleadingly referred to as deconvolutions). Transpose convolutions are convolutions where striding is applied at the output instead of the input. The name stems from the fact that we can transform the down-convolution defined in (3.10) into an up-convolution by simply ex-

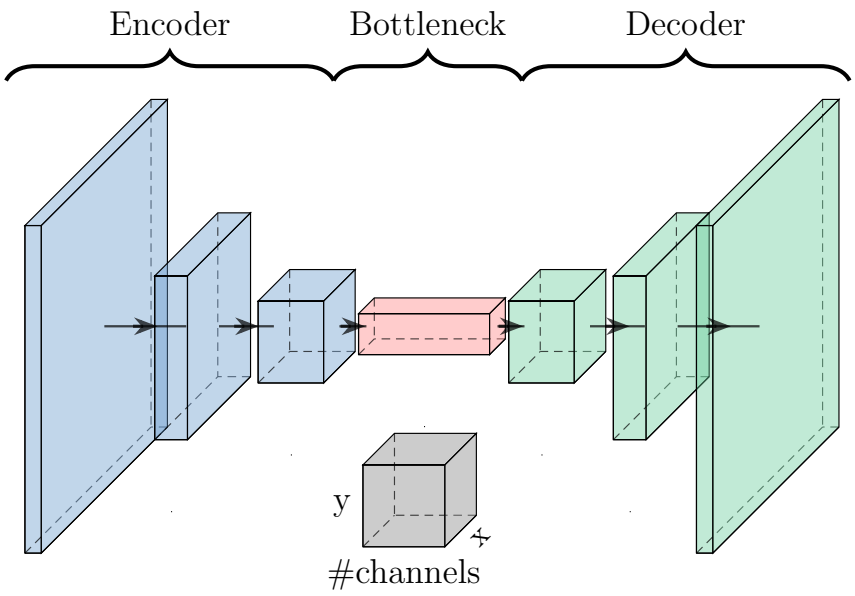

Figure 3.3: Convolutional bottleneck architecture (schematically). Visualization using [19].

changing input with output and transposing the convolution matrix (see also [18]). It should be noted here that kernel weights of the up-convolutions are not inferred from those in the down convolution, but are learned during training.

Some architectures, such as the "U-Net" introduced by Ronneberger, Fischer, and Brox in [20] also introduce direct, so-called skip connections between down-convolution and up-convolution layers on the same level. This can help if some structure is shared exactly between input and output, as is the case in predicting a flow field from geometry. It also bear resemblance to the multigrid method from scientific computing, as explored in [21].

3.3 Generative Adversarial Networks

As deep learning had already been achieving impressive results for discriminative problems, 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 2015 by Goodfellow et al. [22]. 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 [23, `thispersondoesnotexist.com`], text-to-image translation [24] and image-to-image translation [25, 26] as well as combinations multimodal of these [27], 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 [28], flow-based models [29] and diffusion models [30].

3.3.1 Motivation and Concepts

Before a 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 dog. 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. A much more difficult task, however, is to *generate* realistic images of dogs. The key difference is that it is difficult to define a cost function that properly accounts for the target distribution.

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 real data set or were output by the first network (known as the generator). Generator and discriminator are trained together in an adversarial zero sum game, where the former is trying to "fool" the latter, while the latter is trying to expose the "forgeries" of the former.

3.3.2 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 $\mathbf{x} \in X$. 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 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 $\in [0, 1]$ 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. We will call these pluralistically generative. 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 (e.g. image upscaling). An example would be generating a city map from satellite imagery [25]. In this case, the latent input is superfluous and can be omitted. We will call these translatively generative.

In this work, we will train a GAN to perform a mapping from images encoding geometry and boundary conditions to an image showing the resulting flow field. While the underlying PDE is deterministic, both the numerical solution and the scaling of the input image introduce stochasticity into the mapping. This effect is typically small, but can become large in bifurcation settings (cf. Chapter 6). In the initial experiments, no latent space is introduced, but we will discuss potential benefits for doing so in Chapter 7.

3.3.3 Fundamentals and Training

As mentioned above, a GAN can be understood as a game where the two "players" (generator and discriminator ANN) are competing against one another. 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 x^* together with the corresponding conditional, and outputs a value corresponding to its confidence in the input being "real", i.e. from the training set as opposed to created from the generator. Figure 3.3.2 illustrates the flow of information through the network. Mathematically, we have

$$G(\mathbf{z}, \mathbf{c}; \boldsymbol{\omega}_G) \in \mathbb{X} \tag{3.11a}$$

$$D(\mathbf{x}^*, \mathbf{c}; \boldsymbol{\omega}_D) \in [0, 1], \tag{3.11b}$$

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

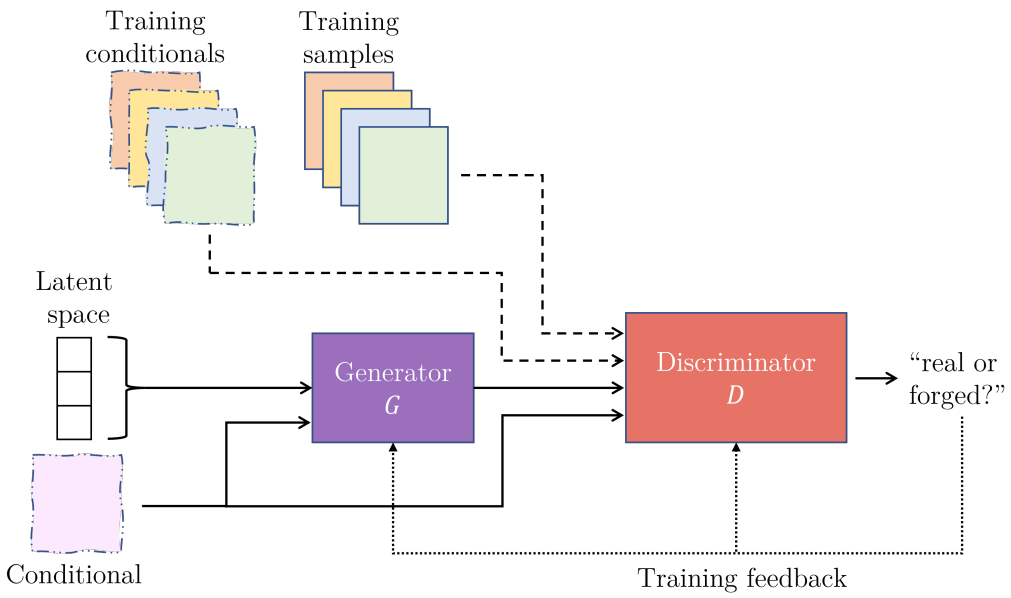

Figure 3.4: Chart schematically illustrating the information flow through a conditional GAN during generator training. The dashed lines represent the data used to train the discriminator.

optimization problem

$$\arg\min_G \max_D V(G, D), \tag{3.12}$$

of the objective function V given by

$$V(G, D) = \mathbb{E}_{p(\mathbf{x},\mathbf{c})} \left[\log D(\mathbf{x}, \mathbf{c})\right] + \mathbb{E}_{p(\mathbf{z},\mathbf{c})} \left[\log \left(1 - D\left(G(\mathbf{z}, \mathbf{c})\right)\right)\right], \tag{3.13}$$

where we have used cross entropy as the measure of uncertainty. For the generator then, the cost function is $C = V$, whereas for the generator we have $C = -V$. Using these cost functions, the model can be optimized by updating generator and discriminator in alternation. For this such an algorithm, it was shown in the original paper [22] that given enough model capacity and training, G will eventually learn to perfectly mimic the underlying data distribution, i.e

$$p\left(G(\mathbf{z}, \mathbf{c}), \mathbf{c}\right) = p(\mathbf{x}, \mathbf{c}). \tag{3.14}$$

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 (3.12), and a Nash equilibrium of the two-player game.

3.3.4 GAN Architectures & Challenges

While the GANs are conceptually compelling, they have proven to be notoriously unstable during training. The learning progress of generator and discriminator has to be delicately balanced in order to avoid undesirable effects such as mode collapse. This describes a phenomenon where the generator becomes stuck in a state of producing only a very narrow range of outputs [31].

In particular, there were initial difficulties making the GANs concept work with deep 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) [32]. 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:

- Replacing all pooling functions with strides in the convolution.
- Removing fully connected layers.
- Batch normalization.

Moreover, the authors found the ReLU output 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 [33], including modifications to the cost functions and methods to help the discriminator identify unwanted model collapse.

A more principled approach of modifying the objective function was taken by Arjovsky et al. in [34]. The authors analyzed sources of instability, as well as possible remedies from a theoretical perspective. Based on this work, the same authors later proposed a new framework for adversarial learning, the so-named Wasserstein GAN or WGAN [35]. To understand it, we first revisit the objective function in eq. (3.13). It turns out that for an optimal discriminator D^*, the value function becomes (except for a constant factor)

the Jensen–Shannon divergence (JSD) between distribution of training and that of generated data:

$$V(G, D^*) = -\log 4 + 2 \cdot \mathrm{JSD}(p_{\mathrm{data}} || p_g), \tag{3.15}$$

where p_g is the distribution defined by $G(\mathbf{z}, \mathbf{c})$ if $(\mathbf{z}, \mathbf{c}) \sim p(\mathbf{z}, \mathbf{c})$. Therefore we may say that the discriminator is trained to estimate the Jensen–Shannon divergence between real and generated data, which it is then using to as "feedback" to the generator. As the name suggests, in the Wasserstein GAN this is replaced with the Wasserstein metric, also known as the earth mover's distance. This metric, the authors illustrate, has more favorable properties for the application, in particular it gives a useful measure even when p_{data} and p_g have disjoint supports (as is in practice usually the case during training). This, the authors note, should among other things help prevent mode collapse. The objective is then given by

$$\arg \min_G \max_D W(G, D), \tag{3.16a}$$

$$W(G, D) = \mathbb{E}_{p(\mathbf{x}, \mathbf{c})} \left[D(\mathbf{x}, \mathbf{c}) \right] - \mathbb{E}_{p(\mathbf{z}, \mathbf{c})} \left[D\left(G(\mathbf{z}, \mathbf{c}) \right) \right], \tag{3.16b}$$

which can be used as cost functions for training generator and discriminator (in the WGAN context more aptly called "critic").

Other important work has focused on analyzing and improving the optimization process [36, 37, 31].

An important family of GAN architectures are the so-named StyleGANs [38, 39, 23]. Whereas normally, the latent space is supplied to the generator as the input layer, which is then transformed to an image from the desired space of training data through learned operations. Instead, with StyleGAN the latent space is first mapped through a fully connected ANN to an intermediate "style" space, and the input layer is a learned constant set of feature maps. This input is then iteratively upsampled like in a normal convolutional generator, however at each step an operation called adaptive instance normalization (AdaIN) is applied. This aligns mean and variance of the input feature maps with those from the current style space. This method shows truly remarkable results, and can be used not just for generating images from latent space but also for interpolating realistically between them.

Chapter 4

Learning Fluid Dynamics

4.1 Machine Learning for Physical Modeling

The use of neural networks in finding approximative solutions to problems with an underlying partial differential equation (PDE) is an emerging field at the intersection of machine learning and scientific computing. Research activity in this area can be roughly split into at least three groups.

One group contains methods that seek to approximate the PDE solution directly using the parametrized function space given by the neural network, as already proposed by Lagaris et al. in 1997 [40]. 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 [41, 42]. In this framework, the solution $u(x)$ is directly approximated by the neural network $\mathcal{N}(x;\omega)$.

The second 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 [43], learning based preconditioning for domain decomposition methods [44] and use of neural networks for obtaining closure terms in turbulence modeling [45].

The method presented here belongs to a third group of using neural networks as reduced order surrogate models. Different to the first group, these operate on existing discretizations. The network is typically trained to map from a raster image of the domain, together with information about boundary / initial conditions as well as additional non-dimensional parameters, onto one or multiple images showing the approximated 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 approximate steady state flow [46, 47]

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 examined model range directly from data. This could potentially reduce the need for hand-crafted domain specific models using assumptions based on expert knowledge. To draw an analogy, the learning process could be akin to how humans can build up a physical intuition without ever studying physics. Reduced order models can be very useful in many applications. One potential application is design space exploration, where the it could enable scanning a large number of configurations for optimality, or even providing design feedback to an engineer in close to real-time.

4.2 Use of GANs in Physical Modeling

Since the GAN framework was first proposed in 2015 [22], numerous studies have looked at applying it to modeling problems. Most of these 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 [48]. The authors trained GANs to solve 2D boundary value problems, specifically Laplace's equation and the incompressible steady-state Navier-Stokes equations (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 a L1 loss with respect to the ground truth. For the case of Navier-Stokes, there are three input and output channels, one for each of the variables (velocity components u,v and pressure p). The PatchGAN architecture was used for the discriminator. The authors were able to obtain high accuracy on a test set with a relative mean absolute error (MAE) of less than 1%, and show 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.

A number of 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 [49]. The method was found to provide fast and accurate solutions for the analyzed test cases. In another recent pub-

lication, 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 [50]. 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 [51]. In their publication, the authors supply separate images showing shape, loads and boundary conditions in the domain as input and extract a single output image of the domain showing von Mises stress. The generator uses a regular bottleneck architecture. The authors found the GAN to consistently outperform a previous model using a regular CNN architecture.

While all the work mentioned so far relies on a purely data-driven approach, other authors have investigated the effects of incorporating physics constraints into the training process. In one publication, a cGAN was trained to predict how a given flow field around a cylinder would evolve one time step into the future [52]. The authors compared four variants: Regular CNNs and GANs, 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 pix2pix GAN implementation to predict wind speeds in urban settings based on a building height map [53]. Again they trained the generator both on the discriminator and on the ground truth. Generalization performance was mixed, but the authors did not benchmark it against a comparable reduced order model.

Chapter 5

Methodology

As laid out in the introduction, the goal of this project is to investigate the potential for training GANs as reduced order models for predicting fluid flow. The networks are trained to map from an image of geometry (and perhaps boundary conditions) to a flow field image. The training data is generated using numerical simulations. In this chapter, we will briefly describe the methodology for generating the training data, as well give an introduction to the neural network architecture used as a starting point for the project. It should be noted that the methodology will likely be subject to change as the research progresses.

5.1 Flowfield Generation

To solve the Navier-Stokes equations presented in Chapter 2, we will use the popular free open source toolbox OpenFOAM [54]. It includes a vast range of tools and solvers for continuum mechanics problems, with a clear focus on CFD. Most solvers, written in C++, implement the so-called finite volume method (FVM) of discretizing PDEs. This method is often favored for the solution of PDEs with underlying conservation laws, as it is inherently conservative (i.e. the PDEs conservation properties are upheld by the numerical scheme).

5.1.1 Finite Volume Method

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 which the PDE is integrated. Thus solution quantities are the averages of the field quantities in each cell. Divergence terms in the volume integral are transformed to flux terms integrated across the cell boundaries using Gauss's theorem. These fluxes are central to the method, and they are evaluated in a zero-sum way, i.e. the flux leaving one cell is always entering another (except at the domain boundary). This is

the reason for the conservative property. 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 with the average values taken as defined at the centers of each cell, but other approaches have also been developed, such as Godunov's scheme [55]. For a more detailed treatise of the finite volume method, see [56].

Besides the conservation property, another major reason for the popularity of FVM for practical use is the ease with which it can incorporate almost arbitrary discretization meshes. For the initial testing, we used the OpenFOAM utility blockMesh [57] to generate the computational mesh. The tool can generate regular parametric meshes including grading and curved edges.

5.1.2 Solver

The solver used for generating the training data is called simpleFOAM [58]. As per its name, it implements a fractional step method known as "Semi-Implicit Method for Pressure-Linked Equations" (SIMPLE), first proposed in [59]. It is an iterative algorithm for solving the incompressible steady state Navier-Stokes equations. Fractional step refers to the fact that the two field quantities $\mathbf{u}$ and p are solved for in alternation. We are using the RANS-equations with the k-ε closure model, as discussed in Section 2.3.

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 process bears many similarities to time-stepping in a transient problem. For steady-state solving, implicit update methods are typically used, as they allow for larger step sizes without loosing stability. In every iteration of the SIMPLE algorithm, first the momentum equation is solved (i.e.) using the pressure field from the previous iteration 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 [9, section 7.2] or the source code [58].

5.2 ML Implementation

The artificial neural networks will be implemented using the free open source machine learning library TensorFlow [60], version 2.8. It offers efficient low level tensor computation, differentiable computing and parallelization capabilities.

As template and starting point for building a reduce order GAN for fluid prediction, we will use the image-to-image translation model pix2pix first presented in [25]. This GAN model was developed general purpose tool, and has been successfully applied across a large number of tasks. These include semantic segmentation, image inpainting, conversion of sketches or schematics to photos, greyscale to color images, aerial photographs to street maps, and day to night scenes, among many others. The code was made freely available online, and has since enjoyed great popularity in the ML community. The following will give an overview of the most important architectural features and characteristics.

An important property of pix2pix is that the model is not a "pure GAN". This is because the generator is trained using feedback from the generator *as well as* the ground truth image. Thus the total objective function is

$$V(G,D) = \mathbb{E}_{p(\mathbf{x},\mathbf{c})}\Big[\log D(\mathbf{x},\mathbf{c}) + \log\left(1 - D\left(G(\mathbf{c})\right)\right) + \lambda\|G(\mathbf{c}) - \mathbf{x}\|_1\Big], \quad (5.1)$$

where $\mathbf{c}$ is the input image, and $\lambda \in \mathbb{R}_+$ is a hyperparameter weighing the L1-distance to ground truth $\mathbf{x}$. The motivation given by the authors for including the ground loss term is that it has been shown previously that training based on ground loss alone can already sufficiently capture the deterministic, large-scale features. The adversarial training on the other hand can help the generator to fill in high-fidelity detail and produce realistic results.

Another important characteristics of pix2pix is that the authors completely omitted a latent space input, as it did not show any effects in their initial testing. Some stochasticity is introduced through a method known as dropout [1] , which they apply not just during training but also during testing, however as the authors point out the effect is small. The generator uses a U-Net architecture, with strided convolutions, batch norm and ReLU units (leaky ReLU in the decoder), following some of the recommendations from earlier papers discussed above.

[1] With dropout, a certain random percentage of neuron are removed from the network at each evaluation, see [12, section 7.12] for more details.

The authors used specific type of discriminator architecture which they term PatchGAN. Instead of discriminating an image as a whole, it essentially classifies individual patches of the image on a real-fake spectrum. The patches are overlapping, and their size can be easily adjusted by modifying the architecture. As the authors note, "such a discriminator effectively models the image as a Markov random field, assuming independence between pixels separated by more than a patch diameter". The operation performed is the exact same for all patches, and so the PatchGAN can be understood as a type of texture or style loss.

Chapter 6

Initial Experiments

6.1 Training Data

As our initial toy problem we use a 2D channel flow through a sudden expansion. A sketch of the domain is shown in 6.1. Within a specific range of Reynolds numbers, it gives rise to a bifurcation. This behavior was documented experimentally in [61], and it is reproduced in the simulation as shown below. Specifically, as the incoming flow leaves the small inlet channel, it will attach to the upper or lower wall of the large channel. Doing that, it will choose the wall that is closer to the inlet. If the inlet is at the center of the main channel, the outcome is essentially random.

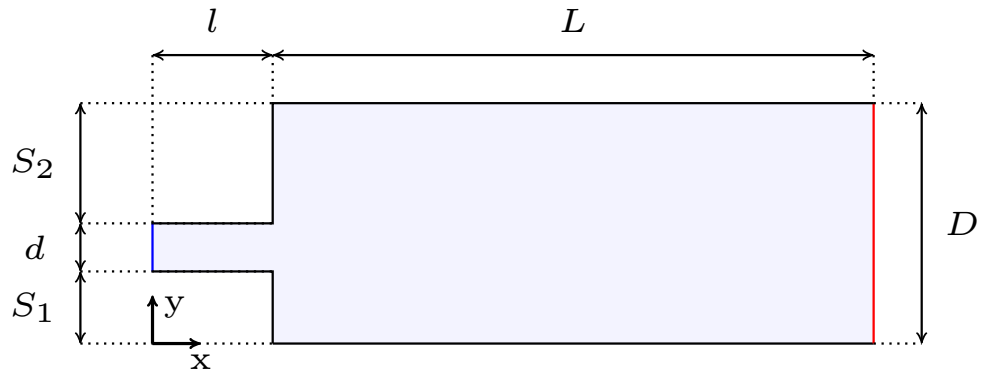

Figure 6.1: Illustration of the domain showing all relevant geometrical parameters. The different boundary types are color coded as inlet (blue), outlet (red) and wall (black).

The reason for choosing this particular configuration is that it exhibits interesting behavior while being very simple geometrically. In particular, the bifurcation might be instructive for understanding the behavior of the GAN model and could help illuminate the differences between generators trained on discriminative vs. L1 losses. Particularly interesting in this regard is the

situation where the inlet flow is very close to the center of the large channel. By examining how the generator deals with the bifurcation, we might be able to infer something about how the reduced order model handles uncertainty more generally. An early hypothesis, for instance, was that a generator trained on an L1 loss might produce something like a superposition of states is the outcome is uncertain, whereas a GAN trained generator would perhaps be more decisive and choose a particular state. We will revisit this hypothesis at the end of the chapter.

The domain is parametrized by just two ratios, the ratio between small and large channel width as well as the ratio between the lower and upper shoulder lengths of the inlet:

$$r_D = \frac{d}{D}, \quad r_S = \frac{S_1}{S_2}. \tag{6.1}$$

Together with the fixed parameters $D = 1\text{m}$, $L = 3\text{m}$, $l = 0.5\text{m}$, these fully constrain the geometry. For the velocity we impose Dirichlet boundary conditions at the inlet ($\mathbf{u} = u_{\text{in}}\mathbf{e}_x$) and walls ($\mathbf{u} = \mathbf{0}$), as well as a homogeneous Neumann condition at the outlet ($\partial_x \mathbf{U} = \mathbf{0}$). As inlet velocity we choose $u_{\text{in}} = 10\text{ms}^{-1}$.

Regardless of parameter choice, the domain is discretized into a computational mesh consisting of 19200 rectangles. The grading functionality of blockMesh was used to refine the mesh near the channel walls, since that is where the largest gradients are generally expected. Moreover, it was ensured that changes in cell geometry are gradual, which almost fully constrains the rest of the mesh construction. An example of a full mesh shown in Figure 6.2.

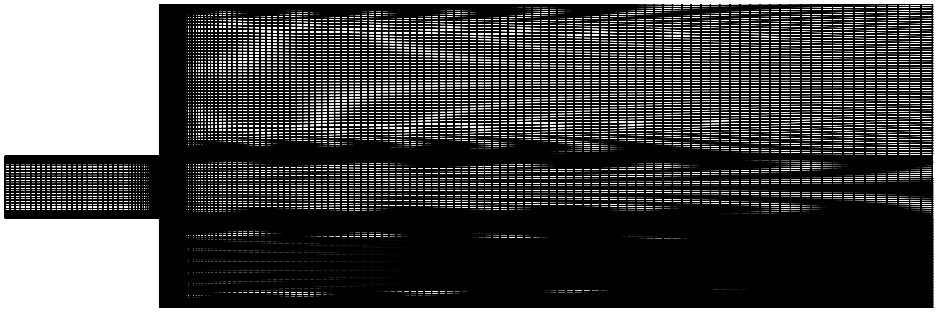

Figure 6.2: Full computational mesh for the case of $r_D = 0.2$, $r_S = 0.6$.

The training data were created in two parameter sweeps. The first includes all combinations of

$$r_D \in \{0.1, 0.2, 0.3\} \text{ with } r_S \in \{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$$

as well as the inverse of the latter, for a total of 51 combinations. The second, a more narrow sweep around the center position, consists of all 60 combinations of

$$r_D \in \{0.12, 0.14, 0.16, 0.18\} \quad \text{with}$$
$$r_S \in \{0.7, 0.75, 0.8, 0.84, 0.88, 0.92, 0.96, 1, 1.04, 1.08, 1.12, 1.16, 1.2, 1.25, 1.3\}.$$

In total thus there are 111 configurations in the initial training set. Each one was solved to convergence using the methods described in Section 5.1. Subsequently, the three resulting fields u_x, u_y and p were interpolated to a regular 256 by 83 grid to be used input for a CNN. For points outside of the domain, all values were set to zero. The geometry was encoded similarly using binary, with each point being assigned a one if it lays inside the domain, and zero otherwise. To give an impression, Figure 6.3 shows a random selection of images showing the magnitude of the velocity. In order to remove potential

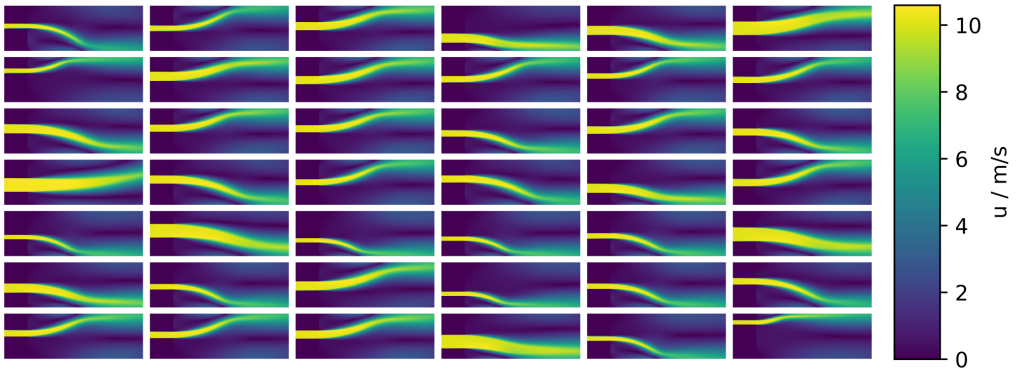

Figure 6.3: Random selection of flow fields showing absolute value of velocity. Regions outside the domain are shown as zero.

bias originating in the numerical algorithm, we additionally add a mirror version (along the x-axis) of each data sample to the training set.

6.2 Training Procedure

As laid out in Section 5.2, we are using the pix2 architecture as a starting point for building the reduced order fluid model. This network is designed for image-to-image translation, and therefore uses three input and output channels (`rgb`). Our input data uses only a single channel image for the binary geometry encoding. The output uses three channels (u_x,u_y,p), however while the `rgb` data is from the bounded interval $[0, 1]$, this is not the case for our data. Therefore, we had to change the activation function of the output layer from sigmoid to linear. we also use three output channels but only a single input channel for the geometry. The boundary conditions are kept fixed for these initial experiments, so we do not need to provide any additional information on that to the generator. As mentioned previously, the generator uses a U-Net architecture, while the discriminator uses the PatchGAN architecture. Both are convolutional architectures, and all use a 4×4 kernel. The total number of trainable parameters is 48 million for the generator, and 2.8 million for the discriminator. See appendix Figure A.1 for a detailed visualization of the architectures. The output of PatchGAN is size 14×30, each pixel having a receptive field of 70×70 (see [62] for calculation). We also did some initial testing using a global discriminator, but did not find it to improve results.

In a preprocessing step, we pad the arrays with zeros such that both dimensions are powers of two, such that the final dimensions of the input images are 256×128. This simplifies the treatment using strided convolutional layers in the U-Net architecture. In order to impart a certain level of invariance onto the GAN w.r.t. to the location of the data in the image, we added every image twice to the dataset, once padding at the top and once at the bottom. A more optimal method of randomly shifting data during training is yet to be explored. The data set then consists of a total of 444 samples, each containing four 256×128 images. It was split up into test and training set according to the typical 20% – 80% ratio.

Training is done one a single sample basis (SGD) using the gradient based *Adam* optimizer [63]. The learning parameters are the same for both generator and discriminator. At each step, a random sample is chosen and its geometry is fed as input into the generator. The data produced by the generator is multiplied by the geometry map, i.e. anything outside the domain is not penalized. Subsequently, the discriminator is called twice, once on the

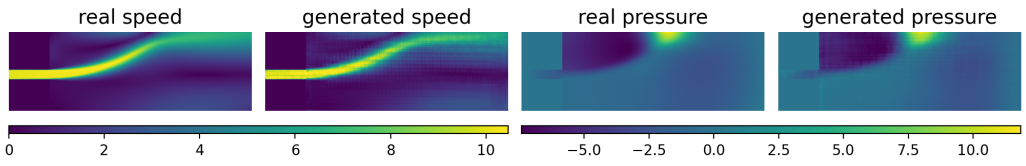

Figure 6.4: Sample output of trained generator. Shown are the velocity (magnitude) and pressure field, as well as their ground truth for comparison.

real and once on the generated data. The geometry is provided as conditional both times. Then, both generator and discriminator can be updated based on the cost function described in Section 3.3.3. The cross entropy of the patchGAN output is taken w.r.t. an array of ones or zeros for the real and generated data respectively.

6.3 Initial Results

In the initial of the initial results, we will focus on qualitatively assessment. A more complete analysis including quantitative measures will part of the final thesis.

After training for around 25000 steps, the generator was able to produce realistic looking flow fields on many of the unseen inputs from the test set. Figure B.1 in the appendix visualizes training progression by showing generated flow fields after various numbers of steps. Figure 6.3 shows a randomly selected output generated by the fully trained generator, and the ground truth for comparison. A larger batch of randomly selected samples is shown in Figure C.1 of the appendix.

During almost all experiments, we saw the discriminator eventually "winning", i.e. reliably distinguishing generated from real samples with high accuracy (cf. Figure B.1). This means that the theoretical optimum state is not reached (cf. Section 3.3.3). An overpowering discriminator can also be problematic, as it can lead to vanishing gradients on the generator which makes further training progress difficult (cf. [35]).

Our hypothesis that a generator trained purely on L1 loss would generate superpositions or averages of equally likely results (i.e. when the inflow channel is centered) were not confirmed by initial testing. Still, we expect differences in how adversarial generators handle uncertainty compared to those trained exclusively on L1 loss. This requires further investigation, as

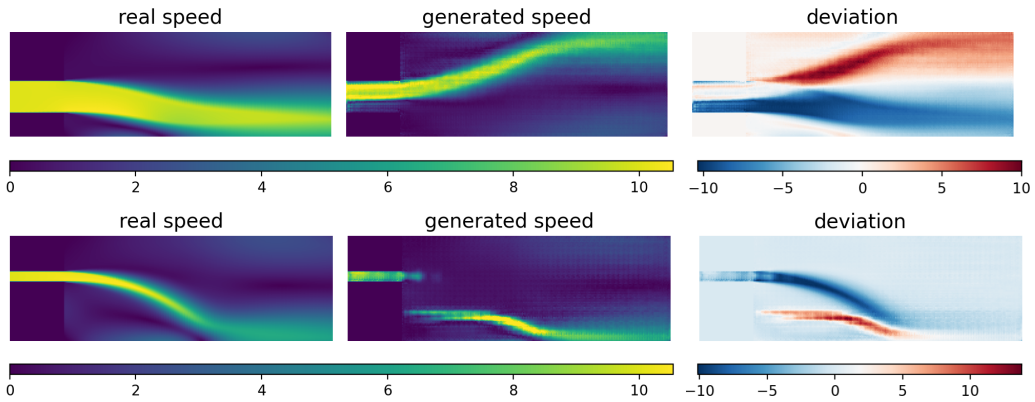

Figure 6.5: Examples representative of common failure modes. Shown are ground truth and generated magnitude velocity field on samples from the test set, as well as the difference between both. Top image shows two forms of mode collapse, failure to correctly take into account the large inflow diameter as well as its position. Lower image shows a generated discontinuity.

discussed in the last chapter..

Nevertheless, the generator manages to predict well the overall structure of flow fields on the test set. When it comes to small scale structures however, tiling artifacts are clearly visible in the generated images even after training for tens of thousands of steps. These were also observed in the original pix2pix publication [25], however for a different PatchGAN architecture.

Beyond that, we can identify a few large scale failure modes in the data that are worth discussing. The first revolves around the bifurcation. For $r_S \approx 1$, the side at which the flow attaches is essentially random (ensured also by the fact that all simulation data are also provided in a mirrored form as training data). Therefore, obviously the generator is expected to make the wrong choice about half of the time, which does not constitute a flaw in the model. However, we have often observed a form of mode collapse, where the generator develops a strong bias towards one of the sides, and wrongly attaches the flow to that side even when it should be clearly the opposite based on the training. This occasional behavior warrants further investigation.

Another form of mode collapse that was identified appears for large inflow widths r_D. The generator appears to have difficulties correctly predicting these high width inflows, and will often simply treats these geometries as if the inflow was thinner. Initial attempts at correcting this by artificially

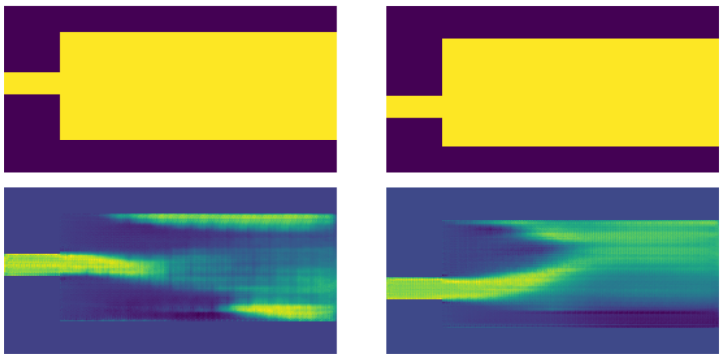

Figure 6.6: Two representative examples of generator failing to generalize to a vertically shifted domain. Shown are geometry inputs (top) and magnitude velocity fields generated (bottom).

increasing the proportion of samples with large inlet diameters in the training set were not successful. Further investigation is necessary. An example from the test set showing both forms of mode collapse is given in Figure 6.3. Perhaps the most extreme mode collapse observed is also shown in the same figure, where the network generator predicts a discontinuous flow.

As stated at the beginning of the chapter, in all training samples the domain sits either at the top or at the bottom of the images used for training. In order to test the generalization power of the trained generator, we test it on conditionals showing a domain that is shifted towards the center of the image. Spatial invariance is a core principle of physics, and is also reflected in the equations underlying the simulation (given that no inhomogeneous volumetric force term is present). Therefore this is a good litmus test to probe how close to representations learned by the generator are to the underlying physical laws. Figure 6.3 shows two representative examples of a shifted domain and the resulting generated velocity field. As is clearly visible, the generator fails to generalize to this setting. Further work is required to remedy this and bring the generator closer to physical representations that generalize well.

Chapter 7

Project Outlook & Scope

This final chapter will lay out the most important research questions, insofar as they are already evident, which are intended to be studied in this research project. The focus and scope of the work will also be discussed. As shown in the previous Chapter, the initial experiments provided some promising results and a proof of concept, but they also raised many interesting questions.

The main goal of the project is to investigate the use of GANs as reduced order fluid models. While the methods will be mostly empirical, the clear intention is to focus on understanding the strengths and weaknesses of the GAN approach for this purpose, assessing potential and limitations, as well as highlighting pathways and methods for further improvement. In this sense, optimizing the methodology to obtain better results is not an end, but a means to further understanding. One objective in particular is to compare adversarially trained generators to those trained partially or inclusively on pixel-based loss (L1 or others). We will attempt to use the bifurcation setting to probe how these generators deal with uncertainty.

A few obvious shortcomings were already identified in the initial methodology and results described in the previous chapter. In order to impart shift invariance and improve generalization, we will augment the training dataset by randomly shifting the domain vertically inside the image. We will also investigate the causes of tiling artifacts and possible remedies. Moreover, we will search for modifications to allow for arbitrary domain sizes and aspect ratios (currently limited to powers of two due to the U-Net architecture).

An important step towards realistic application will be the incorporation of boundary conditions into the conditional input. This will potentially allow the generator to learn representations that capture significant aspects of the underlying laws. For training such a generator we will be able to make use of more general data augmentation such as rotation and mirroring. Moreover, we can also vary boundary conditions in the simulations during data generation. However, we expect training on such a much more general dataset to be very challenging and the outcome is unclear.

In order to potentially improve convergence and reduce mode collapse,

we will further study recent publications on GANs addressing these, some of which were discussed in Section 3.3.4 (e.g. Wasserstein GAN). If we can identify potential improvements to the pix2pix architecture and training process, we will implement and test them.

If we can establish a training procedure that produces satisfactory results on the simple, we will test it by applying it to more complex geometry, e.g. by adding an obstacle to the domain.

Another interesting research question is probing to what degree we can discern "physical knownledge" in the generator, and whether we are able to recover properties of the underlying equations. A related topic of study is to investigate the effects of adding physics-based cost functions on training and results.

Lastly we will also take a look at performance. In this regard, we may study how much the size of generator and discriminator can be reduced and still produce useful results. Most importantly, we will attempt to compare the computational cost of generating a prediction to running a comparable numerical simulation

Appendices

Appendix A

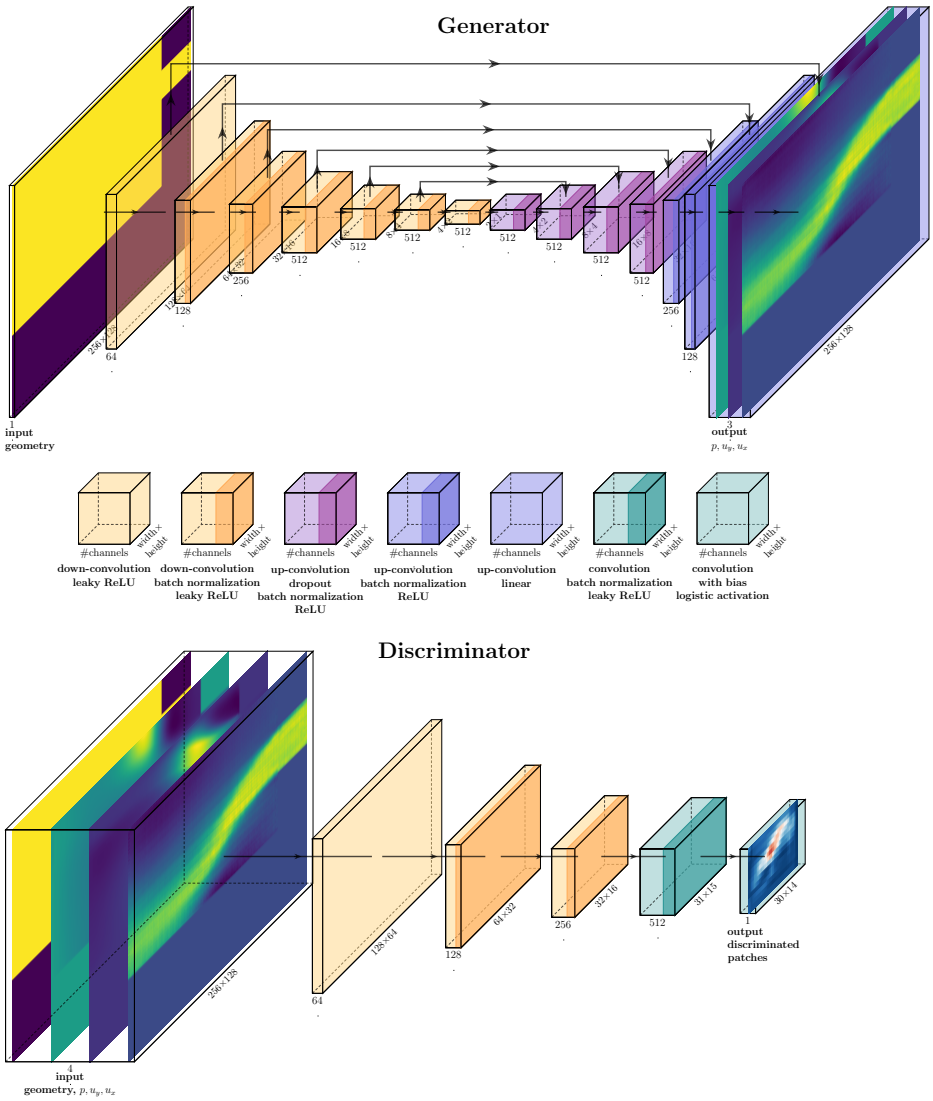

Figure A.1: Architecture of pix2pix GAN used for initial experiments.

Appendix B

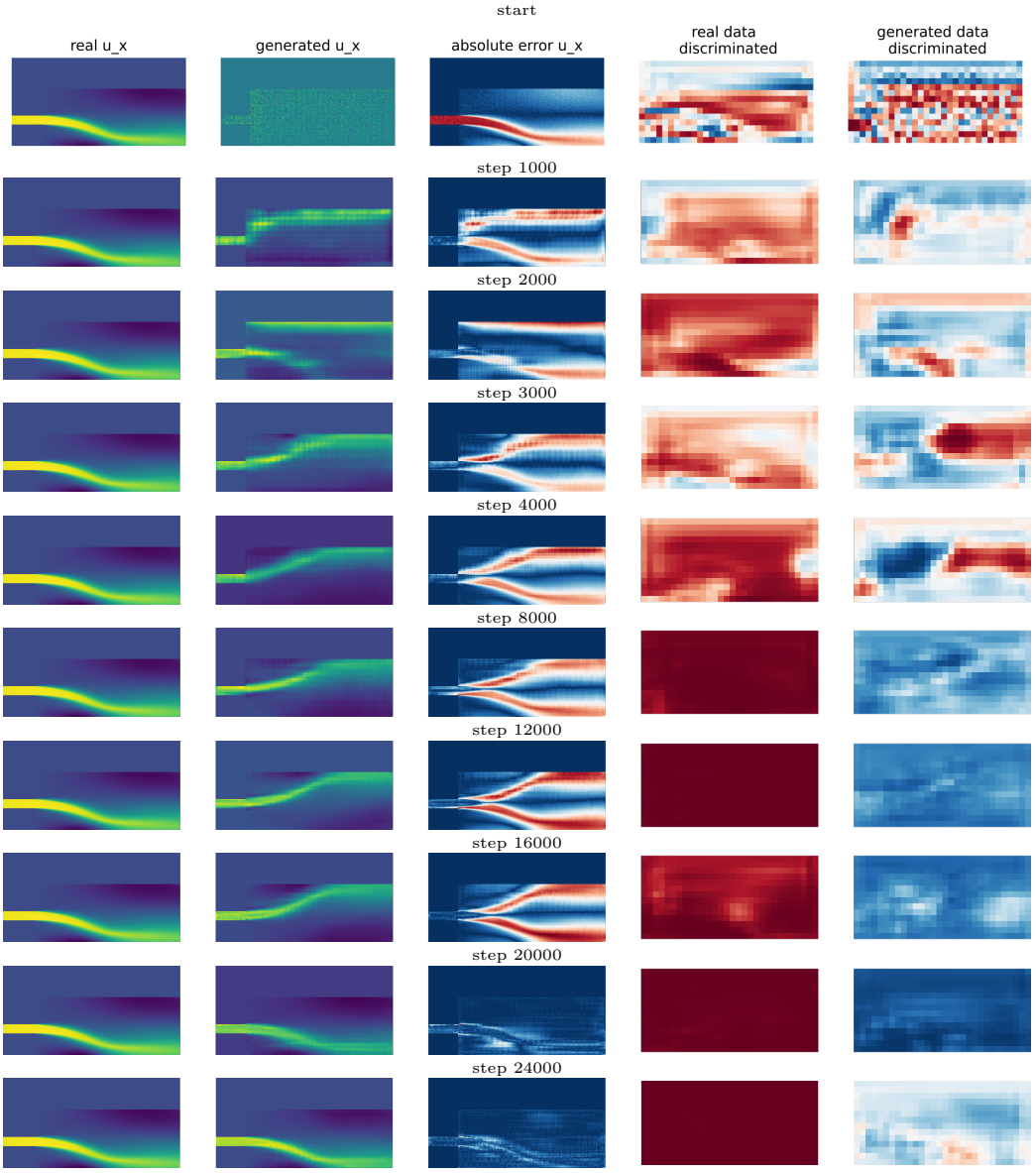

Figure B.1: Training progress visualized using generated fields of u_x after different numbers of training steps.

Appendix C

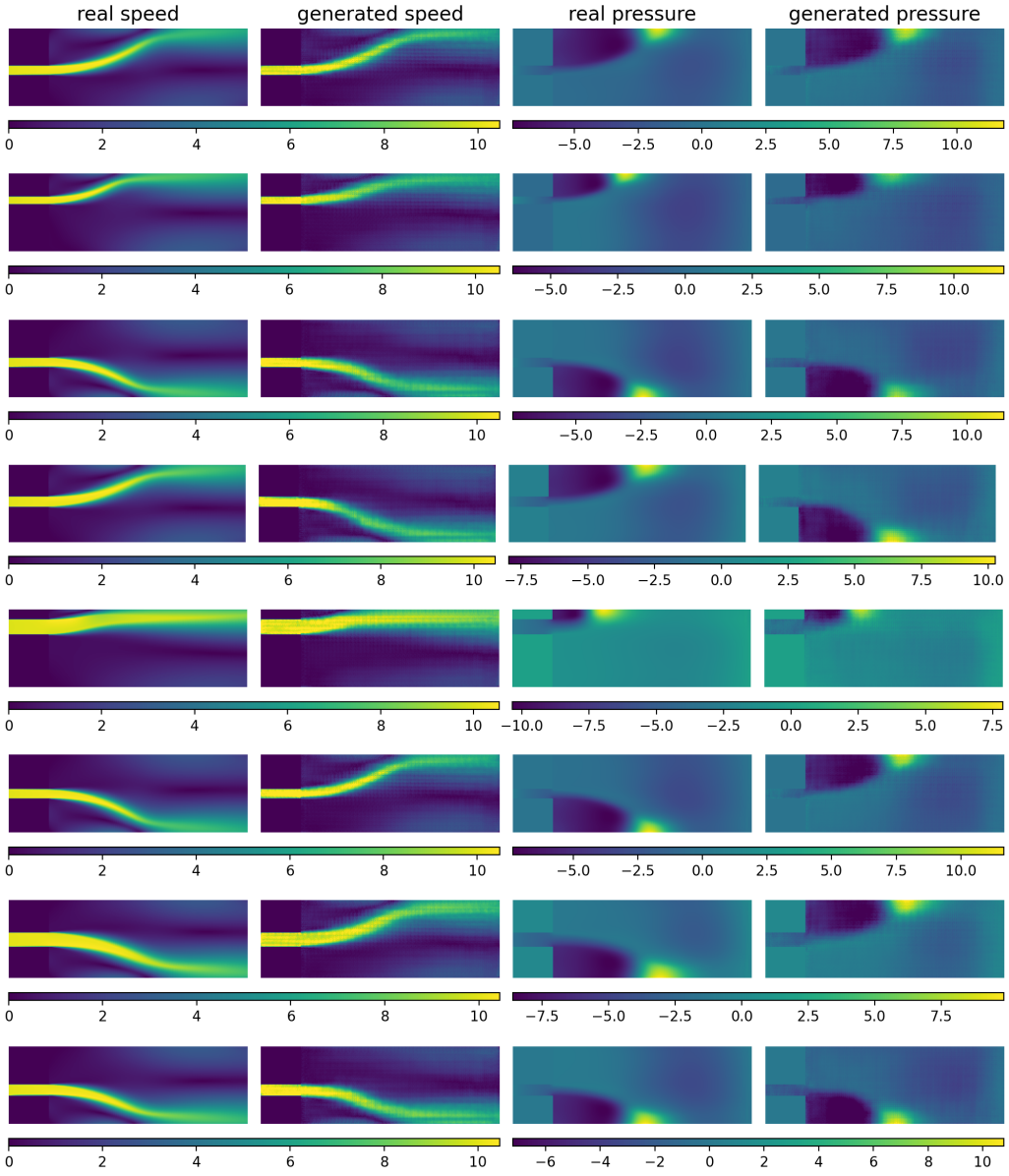

Figure C.1: Sample results after 25000 training steps.

Bibliography

[1] 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. 3).

[2] Robert Service. "'The game has changed.' AI triumphs at solving protein structures". In: *Science* (Nov. 2020). DOI: 10.1126/science.abf9367 (cit. on p. 3).

[3] Protein Structure Prediction Center, University of California, Davis. URL: https://predictioncenter.org/casp14/zscores_final.cgi (cit. on p. 4).

[4] Leonid P Lebedev and Michael J Cloud. *Tensor Analysis*. WORLD SCIENTIFIC, Apr. 2003. DOI: 10.1142/5265 (cit. on p. 6).

[5] Ronald L. Panton. *Incompressible Flow*. John Wiley & Sons, Inc., July 2013. DOI: 10.1002/9781118713075 (cit. on p. 6).

[6] 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. 7).

[7] C. S. Jog. *Fluid Mechanics*. Cambridge University Press, 2015. DOI: 10.1017/cbo9781316134030 (cit. on p. 8).

[8] 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. 8, 9).

[9] 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. 8, 9, 29).

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

[11] 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. 10).

[12] Ian Goodfellow, Joshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press Ltd, 2016. 800 pp. (cit. on pp. 10, 15, 16, 18, 30).

[13] 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).

[14] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: (May 2020). arXiv: 2005.14165 [cs.CL] (cit. on p. 13).

[15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (2015) (cit. on p. 15).

[16] 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. 16).

[17] Bing Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network". In: (May 2015). arXiv: 1505.00853 [cs.LG] (cit. on p. 16).

[18] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: (Mar. 2016). arXiv: 1603.07285 [stat.ML] (cit. on pp. 18, 19).

[19] Haris Iqbal. *HarisIqbal88/PlotNeuralNet v1.0.0*. 2018. DOI: 10.5281/ZENODO.2526396 (cit. on p. 19).

[20] 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 p. 19).

[21] 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. 19).

[22] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems* 27 (2014) (cit. on pp. 19, 22, 26).

[23] Tero Karras et al. "Alias-Free Generative Adversarial Networks". In: (June 2021). arXiv: 2106.12423 [cs.CV] (cit. on pp. 20, 24).

[24] Kai Hu et al. “Text to Image Generation with Semantic-Spatial Aware GAN”. In: (Apr. 2021). arXiv: 2104.00567 [cs.CV] (cit. on p. 20).

[25] Phillip Isola et al. “Image-to-Image Translation with Conditional Adversarial Networks”. In: (Nov. 2016). arXiv: 1611.07004 [cs.CV] (cit. on pp. 20, 21, 30, 37).

[26] 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. 20).

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

[28] Diederik P Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *ICLR*. Dec. 2014. arXiv: 1312.6114 [stat.ML] (cit. on p. 20).

[29] Danilo Jimenez Rezende and Shakir Mohamed. “Variational Inference with Normalizing Flows”. In: *ICML*. May 2015. arXiv: 1505.05770 [stat.ML] (cit. on p. 20).

[30] Prafulla Dhariwal and Alex Nichol. “Diffusion Models Beat GANs on Image Synthesis”. In: (May 2021). arXiv: 2105.05233 [cs.LG] (cit. on p. 20).

[31] 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 pp. 23, 24).

[32] 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. 23).

[33] Tim Salimans et al. “Improved Techniques for Training GANs”. In: (June 2016). eprint: 1606.03498 (cs.LG) (cit. on p. 23).

[34] Martin Arjovsky and Léon Bottou. “Towards Principled Methods for Training Generative Adversarial Networks”. In: (Jan. 2017). arXiv: 1701.04862 [stat.ML] (cit. on p. 23).

[35] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein GAN”. In: (Jan. 2017). arXiv: 1701.07875 [stat.ML] (cit. on pp. 23, 36).

[36] Vaishnavh Nagarajan and J. Zico Kolter. "Gradient descent GAN optimization is locally stable". In: (June 2017). arXiv: 1706.04156 [cs.LG] (cit. on p. 24).

[37] Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. "The Numerics of GANs". In: (May 2017). arXiv: 1705.10461 [cs.LG] (cit. on p. 24).

[38] Tero Karras, Samuli Laine, and Timo Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". In: (Dec. 2018). arXiv: 1812.04948 [cs.NE] (cit. on p. 24).

[39] Tero Karras et al. "Analyzing and Improving the Image Quality of StyleGAN". In: (Dec. 2019). arXiv: 1912.04958 [cs.CV] (cit. on p. 24).

[40] 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. 25).

[41] 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. 25).

[42] 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. 25).

[43] 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. 25).

[44] 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. 25).

[45] 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. 25).

[46] 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. 25).

[47] Axel Klawonn Matthias Eichinger Alexander Heinlein. *Surrogate Convolutional Neural Network Models for Steady Computational Fluid Dynamics Simulations*. Tech. rep. University of Cologne, 2020 (cit. on p. 25).

[48] 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 p. 26).

[49] 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 p. 26).

[50] 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 p. 27).

[51] 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 p. 27).

[52] 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. 27).

[53] Aleksandra Sojka Sarah Mokhtar and Carlos Cerezo Davila. "Conditional Generative Adversarial Networks forPedestrian Wind Flow Approximation". In: *Proceedings of SimAUD*. 2020, pp. 25–27 (cit. on p. 27).

[54] The OpenFOAM Foundation. URL: https://openfoam.org/ (cit. on p. 28).

[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. 29).

[56] 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. 29).

[57] OpenCFD Ltd. *blockMesh user guide*. URL: https://www.openfoam.com/documentation/user-guide/4-mesh-generation-and-conversion/4.3-mesh-generation-with-the-blockmesh-utility (cit. on p. 29).

[58] OpenCFD Ltd. *simpleFOAM documentation*. URL: https://www.openfoam.com/documentation/guides/latest/doc/guide-applications-solvers-incompressible-simpleFoam.html (cit. on p. 29).

[59] 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. 29).

[60] TensorFlow Developers. *TensorFlow*. 2022. DOI: 10.5281/ZENODO.4724125 (cit. on p. 30).

[61] 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. 32).

[62] 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. 35).

[63] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 2014). arXiv: 1412.6980 [cs.LG] (cit. on p. 35).