DEPARTMENT OF INFORMATICS

TECHNICHAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Solver-Based Learning of Pressure Fields for Eulerian Fluid Simulation

Robert Brand

DEPARTMENT OF INFORMATICS

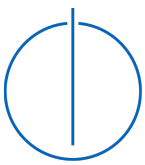
TECHNICHAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Games Engineering

Solver-Based Learning of Pressure Fields for Eulerian Fluid Simulation

Solver-Basiertes Lernen von Druckfeldern für Eulersche Flüssigkeitssimulationen

Author:	Robert Brand
Supervisor:	Prof. Dr. Nils Thuerey
Advisor:	Philipp Holl, M. Sc.
Submission Date:	15.09.2020



I confirm that this Master's Thesis in Informatics: Games Engineering is my own work and I have documented all sources and material used.

Munich, 15.09.2020 Robert Brand

Acknowledgments

I would like to thank Prof. Dr. Nils Thuerey and Philipp Holl for giving me the opportunity to work on this topic, for supporting me with a lot of advice and discussion along the way and for their patience with my somewhat unconventional thesis time schedule. Likewise, thank you to Steffen Wiewel and Kiwon Um for the input they gave me in several discussions.

Of course, I also want to thank my friends and family for supporting me during the creation process, putting up with my occasional distractedness and especially for giving me feedback on ideas and drafts.

Abstract

Efficiently and accurately solving partial differential equations is an important task in many fields of science and engineering. In Eulerian fluid simulation, solving the pressure equation constitutes a large part of the computational footprint of the simulation. We therefore investigate the use of Convolutional Neural Networks (CNN) to accelerate this step. Specifically, we compare different approaches to training a CNN to provide an initial approximate solution to the numerical pressure solver, with the aim of reducing the overall iterations and thereby computational time needed. We find that integrating the solver into the network's training loop, allowing the network to observe the solver's behavior at training time, significantly improves the usefulness of the network's pressure predictions as an initial state for the solver. While previously used physics-informed approaches show better standalone simulation stability, we show that hybrid simulations using our solver-trained models and the numerical solver in conjunction notably outperform them. These solver-based hybrid simulations achieve the same accuracy as traditional simulations, while requiring significantly less computation time for commonly used target accuracies. Additionally, we show that physics-informed and solver-based training approaches can be effectively combined to alleviate the aforementioned stability issues when using our solver-trained models as standalone pressure predictors.

Contents

1 Introduction

Fluid simulations have a vast range of interesting applications. From special effects in movies and video games to modeling liquid and gas flow in various engineering disciplines, to medical applications, the list goes on. At the heart of these simulations lie the Navier-Stokes equations, a set of partial differential equations that describe the physical dynamics of fluid flow. To efficiently simulate a fluid, these equations are commonly split up and solved in parts. One particularly important part is the calculation of the pressure acting within the fluid, which prevents it from being unnaturally compressed. In Eulerian fluid simulation (see section 3.1), calculating the pressure field entails solving a linear system of equations using a numerical solver. As this has to be done for every individual time step, fluid simulations with high accuracy and fidelity still take a long time to compute. Due to this, a recent trend in the fluid simulation community has been to investigate how techniques from machine learning may be used to accelerate traditional simulation methods. Methods approximating the pressure calculation step using neural networks have proven to yield good results [Tom+17], significantly increasing computational performance at the expense of some accuracy. However, directly replacing the numerical solver by a trained network always leads to an approximation with fixed accuracy. By contrast, a numerical solver can keep iterating until it converges to the desired accuracy.

In this work, we therefore investigate how a trained neural network may be used in conjunction with a numerical solver to provide simulation speed-ups while retaining the same accuracy guarantees. A network's pressure prediction can be used as an initial state for the solver to further improve upon. We thus focus on evaluating the solver's iteration behavior given initial pressure guesses by networks trained with different approaches. In particular, we explore the usefulness of training the neural networks together with the numerical solver. We start by summarizing relevant research that has already been performed in this field and discussing similarities and differences to our experiments (see chapter 2). We then provide a review of the theoretical foundations of this thesis, Eulerian fluid simulation (see chapter 3) and Convolutional Neural Networks (see chapter 4). Our general approach is outlined in chapter 5, before we evaluate the results of our experiments (see chapter 6). Lastly, we summarize our main findings (see chapter 7) and give insight into possible follow-up research (see chapter 8).

2 Related Work

The application of machine learning techniques to the domain of physical simulation has gained popularity and seen a lot of interesting research in recent years. In relation to this thesis, two sub-areas are particularly relevant. In this chapter, we will therefore first highlight recent findings concerning the application of Artificial Neural Networks (see chapter 4) to the simulation of fluids. Secondly, we will discuss recent research into the use of Deep Learning for solving partial differential equations (PDEs), such as the Poisson problem, which arises during the pressure solving step (see section 3.3) of Eulerian fluid simulation (see chapter 3).

2.1 Neural Networks in Fluid Simulation

Due to their complexity and non-linear nature, fluid dynamics are a challenging problem that is still difficult to efficiently solve. Especially in computer graphics applications, a lot of effort has thus been made to find ways in which data-driven approaches can be used to augment or substitute traditional fluid simulation techniques. In particle-based fluid simulation, models based on regression forests [Lad+15] and most recently graph networks [San+20] have been trained to learn the physical forces acting between fluid particles from pre-computed simulations. Both methods showed impressive capabilities to generalize to new simulation settings, with the method proposed by Ladicky et. al. also showing a noticeable performance benefit.

In Eulerian fluid simulation, neural networks have been utilized to improve simulation accuracy and efficiency. Xie et. al. [Xie+18] used Generative Adversarial Networks [Goo16] to introduce convincing details into low-resolution simulations, making it possible to compute an inexpensive coarser simulation as a base and then upsample it. Wiewel et. al. [WBT19] also showed how neural networks can be used to accelerate simulations. They proposed a Long-Short-Term-Memory (LSTM) network [see Nie15, page 204] to predict temporally coherent pressure sequences, arriving at a neural network based solution with noticeable speed-up compared to a traditional fluid solver. Similarly, Tompson et. al. [Tom+17] trained a Convolutional Neural Network to infer the pressure, although they focused on the pressure for each time step in isolation, leaving time integration to traditional schemes. By employing a loss based on

the physics governing fluid flow, they managed to fully replace the numerical solver by a trained model, yielding simulations that trade some accuracy for a significant computational speed-up.

Our approach likewise aims to accelerate Eulerian fluid simulation by training a Convolutional Neural Network to infer pressure fields, reducing the computational time usually spent on the pressure solving step. However, unlike the highlighted works, our goal is not necessarily to completely replace the numerical solver. Instead, we investigate how a trained pressure predictor can be used in conjunction with the solver. Our method thereby does not sacrifice accuracy for speed, but endeavors to retain the same accuracy guarantees a traditional solver provides.

2.2 Learning Partial Differential Equation Solvers

The pressure equation that is solved during Eulerian fluid simulation is a Poisson problem, a partial differential equation (PDE) of the shape $\Delta\varphi = f$, where φ and f are scalar fields. PDEs arise in many different areas of physics and engineering. Consequently, there has been a lot of research on the application of machine learning techniques to them. Long et. al. used neural networks with convolutional operators to discover the underlying PDE formulations in observed simulations [Lon+18]. Apart from PDE analysis, there have also been several works aiming to train neural networks to solve them [RPK17; SS18; Özb+19]. As in fluid simulation, their aim is generally to train an accurate neural network substitute for a numerical PDE solver. Recently, however, some works have emerged that investigate how to improve a traditional PDE solver's performance using trained models. Um et. al. explored a learned corrector, which, applied after a numerical solver, further improves its accuracy [Um+19]. Hsieh et. al. likewise trained a neural network to modify the updates of an existing solver to further improve them and require less iterations overall [Hsi+19]. They thereby arrive at a 2-3x speed-up while maintaining accuracy and convergence guarantees.

In both of these works the solver is also used directly in the loss function, giving them a strong relation to our investigations in this thesis. However, while they focus on using networks to further improve the solver's output, we instead examine how best to train them to predict a useful initial guess that can be used as input for the solver.

3 Numerical Fluid Simulation

This chapter summarizes the principles of fluid simulation used as a basis for this thesis. We chose to focus on Eulerian fluid simulation, where all relevant quantities are stored in grid form. This regular data structure is very well suited for use with Convolutional Neural Networks (see 4). Additionally, we only look at incompressible flow here, i.e. fluid simulations that attempt to keep a constant density. For a more comprehensive review of numerical fluid simulation techniques, we refer to [BM07].

3.1 The Eulerian Viewpoint

To simulate a continuous quantity (such as a fluid) on a computer, it must be split up into discrete elements. An image, for example, is usually processed as a grid of pixels. The resolution may be high enough so as not to be able to distinguish the individual pixels, but current computers cannot simulate a truly continuous color field.
When simulating a fluid, therefore, it must also be discretized. In Eulerian fluid simulation, this is done by splitting the simulation domain, i.e. space, into evenly sized grid cells. Every such grid cell measures the fluid's quantities (e.g. velocity, pressure, density, color, etc.) at that fixed point in space. The flow of the fluid is achieved by shifting the quantities between those grid cells. For example, as the fluid flows upwards, lower cells' density will decrease while the density of upper grid cells increases.

3.2 The Navier-Stokes Equations

The Navier-Stokes equations are the physical foundation of fluid simulation. They describe the motion of a fluid, such as water or smoke, in the form of two partial differential equations that must hold throughout the fluid for it to behave believably.
The overall goal of the simulation is therefore to ensure that the fluid's velocity field u changes in such a way that the Navier-Stokes equations remain satisfied.

3.2.1 Momentum Equation

The momentum equation describes how the fluid moves over time as a result of the internal and external forces acting on it.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \Delta u + g \tag{3.1}$$

Mathematically, this means it expresses the change of the velocity field u over time t, i.e. $\frac{\partial u}{\partial t}$, as a sum of four terms: advection, pressure, viscosity and external forces.

Advection As a fluid moves, it carries its attributes with it. For example, if a part of the fluid were to be colorized with a splash of ink, this color would move along with the fluid's velocity field u. This phenomenon is known as advection. A quantity q's advection may be expressed as $\frac{\partial q}{\partial t} + (\nabla q)u$, i.e. the sum of its change over time and its change in space. Somewhat unintuitively, the velocity u itself should also be advected. If a portion of the fluid moves due to having velocity, then it should conserve that velocity, carrying it to its new location. The advection term of the momentum equation therefore describes how the velocity field u changes due to its own movement.

Pressure Areas of a fluid that have high pressure p should move towards low pressure areas. This force should therefore point in the direction where the pressure decreases the most. This can be expressed as the negative gradient of the pressure field $-\nabla p$. As higher concentrations of fluid require more force to be pushed, the term is multiplied by the inverse density $\frac{1}{\rho}$. In the context of incompressible flow, the pressure p can be seen as "whatever it takes to keep the fluid from being compressed". Therefore, p is a helper quantity that has to be chosen such that the incompressibility constraint (see 3.2.2) holds true.

Viscosity Different fluids have varying degrees of viscosity. Viscous fluids such as honey resist deformation. This means there is an internal force resisting large spatial differences between the velocity in one place and the immediately surrounding area. This can be modeled by the Laplace operator Δ, which measures how far a quantity is from the average around it. The kinematic viscosity ν is a parameter that can be chosen to adjust how viscous the fluid should be. Viscosity is mostly relevant for very specific effects, such as simulation of honey. Furthermore, numerical errors (see 3.3) lead to a similar effect without the need to explicitly simulate viscosity [see BM07, page 9]. We therefore disregard the viscosity term in the following.

External Forces The external forces term g is simply a sum of all external influences on the fluid. The most important part of this is gravity, but e.g. forces resulting from user interaction would also be included here.

3.2.2 Incompressibility Constraint

For an incompressible fluid, the overall density of the fluid should remain constant throughout the simulation. Thus, the density must not globally decrease or increase when advected using the fluid's velocity field u. For this to hold, u must be divergence-free.

$$\nabla u = 0 \tag{3.2}$$

The divergence ∇u describes how much the velocity vectors point inwards or outwards in any given point. If they all point towards the same location, the fluid flows into this position without flowing back out again, creating a sink. A velocity field with zero divergence therefore has no sources or sinks. Hence, quantities advected by it are shifted around without globally increasing or decreasing.

3.2.3 Numerical Solution

To solve complex partial differential equations such as the Navier-Stokes equations numerically, a widely used method is splitting [see BM07, pages 12-14]. Consider an example partial differential equation (PDE):

$$\frac{dq}{dt} = f(q) + g(q) \tag{3.3}$$

Discretizing the time t into steps of size h, we can approximate this as:

$$\begin{aligned} \tilde{q} &= q^n + h \cdot f(q^n) \\ q^{n+1} &= \tilde{q} + h \cdot g(\tilde{q}) \end{aligned} \tag{3.4}$$

This splitting introduces an error on the order of h, but it enables specialized methods to be used to solve every component individually. Applying splitting to the momentum equation (3.2.1) yields the following algorithm for simulating the fluid's velocity field u for one time step:

1. Advect the velocity field u
2. Apply external forces g to u
3. Use pressure p to make u divergence-free
   a) Compute ∇u
   b) Use a numerical solver to obtain p (see 3.3)
   c) Correct u by subtracting $h \cdot \frac{1}{\rho} \nabla p$ (see 3.2.1)

Advection is usually performed using the Semi-Lagrangian method [Sta99]. Here, for each grid point, an imaginary particle is traced back using the negative velocity. The values at the traced position are those that will end up at the grid point due to advection. External forces can simply be added per $\tilde{u} = u + h \cdot g$ (see 3.2.3).
Lastly, the velocity field needs to be made divergence-free (so as to satisfy 3.2.2), before it can be advected again. Since this correction is performed using $-h \cdot \frac{1}{\rho} \nabla p$, it follows that p must be found such that $\nabla(\tilde{u} - h \cdot \frac{1}{\rho} \nabla p) = 0$. In other word, p must be determined so that u is divergence-free after being corrected with the gradient of p.

3.3 Pressure Solvers

To make the velocity divergence free, p has to be determined so that:

$$\begin{aligned} \nabla(\tilde{u} - h \cdot \frac{1}{\rho} \nabla p) &= \nabla \tilde{u} - \frac{h}{\rho} \Delta p \\ \nabla \tilde{u} &= \frac{h}{\rho} \Delta p \end{aligned} \tag{3.5}$$

As the simulation is performed on a grid, the differential operators can be discretized using finite differences (with δx and δy denoting the horizontal and vertical distance between grid points respectively):

$$\begin{aligned} \nabla u &\approx \Big(\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{u_{i,j+1} - u_{i,j-1}}{2\delta y} \Big) \\ \Delta p &\approx \Big(\frac{p_{i+1,j} - 2p_{i,j} - p_{i-1,j}}{2\delta x} + \frac{p_{i,j+1} - 2p_{i,j} - p_{i,j-1}}{2\delta y} \Big) \end{aligned} \tag{3.6}$$

This yields a linear equation for every grid cell (i, j). Equation 3.5 therefore can be expressed as $Ap = d$, where d is the pre-computed divergence vector and A is a coefficient matrix defining the linear system of equations.

These equations can be further modified to include boundary conditions, e.g. to ensure that the pressure at a wall is always equal to that of its neighboring fluid cells, thereby making sure that no velocity into walls occurs.
There are several well established methods of numerically solving a linear system of equations (e.g. Jacobi, Gauss-Seidel, etc. [see Saa03, pages 105 ff.]). For this thesis, we focus on the conjugate gradient method.

3.3.1 The Conjugate Gradient Method

This section will briefly summarize the Conjugate Gradient (CG) algorithm. For an in-depth explanation, we refer to [She+94]. The CG algorithm can be used to solve a linear system of equations

$$Ax = b \tag{3.7}$$

as long as A is a symmetric, positive-definite (i.e. $qAq > 0$ for any vector $q \neq 0$) matrix. Under these conditions, solving $Ax = b$ becomes equivalent to finding the minimum of the function [see She+94, page 54]:

$$f(x) = \frac{1}{2}xAx - bx + c \tag{3.8}$$

Due to A being positive-definite, $f(x)$ takes the shape of a paraboloid bowl. The CG algorithm starts at an initial guess x_0 somewhere on this solution paraboloid. It then takes a series of steps $x_0, x_1, ..., x_n$ until it is sufficiently close to the solution x, i.e. the minimum of the paraboloid. Every such step is defined by a step size α and a search direction.

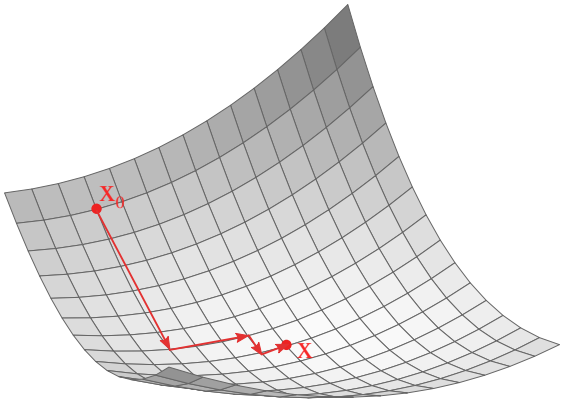


Figure 3.1: Iteratively getting closer to the minimum of the solution paraboloid (figure adapted from [She+94]).

Steepest Descent An obvious approach to choosing the search direction is to use the negative gradient $-f'(x_i)$, as it represents the direction in which $f(x)$ decreases the most. Deriving equation 3.8 and taking into account A's symmetry, we arrive at:

$$\begin{aligned} f'(x) &= \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b \\ -f'(x_i) &= b - Ax_i \end{aligned} \tag{3.9}$$

Intuitively, $b - Ax_i$ also describes how far away the current guess x_i is from satisfying $Ax_i = b$. It is therefore also called the residual $r_i = b - Ax_i$. Taking the residual r_i as the search direction, the step size α, i.e. how far along the negative gradient to go, must be determined next.
The optimal step size α minimizes $f(x)$ along the search direction, since the eventual goal is to find the global minimum of $f(x)$. Therefore, the minimum of the parabola resulting from the intersection of the search direction with the solution paraboloid must be found.

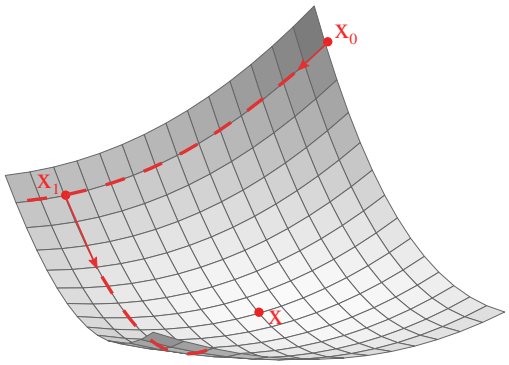


Figure 3.2: Finding the minimum along the intersection parabola of the search direction and solution paraboloid (figure adapted from [She+94]).

This minimum can be found at the point along the search line where $r_{i+1} \cdot r_i = 0$, i.e. where the previous residual and the new one are orthogonal to each other [see She+94, pages 6-7]. From $r_{i+1} \cdot r_i = 0$ it can be derived [see She+94, page 6] that:

$$\alpha = \frac{r_i r_i}{r_i A r_i} \tag{3.10}$$

Using the calculation rules for α, r_i and $x_{i+1} = x_i + \alpha r_i$, the Steepest Descent algorithm can be formulated as in Algorithm 1. The calculation rule for r has been adapted for performance reasons [see She+94, page 8].

$x = x_0$; $r = b - Ax$;
$i = 0$;
while $max(|r|) > tolerance \wedge i < i_{max}$ **do**
  $\alpha = dot(r, r) / dot(r, Ar)$;
  $x = x + \alpha \cdot r$;
  $r = r - \alpha \cdot Ar$;
  $i = i + 1$;
end

Algorithm 1: The Steepest Descent Algorithm

Conjugate Search Directions Ideally, the search directions should be chosen so that the algorithm never has to take more than one step in each direction. In this case, the initial error $e_0 = x_0 - x$ can be expressed as a linear combination of the step vectors:

$$e_0 = -\sum_{j=0}^{n-1} \alpha_j d_j \tag{3.11}$$

Geometrically, e_0 is the vector from the target to the initial guess and the search directions d_j are the base vectors it is made up of. Every step should thus remove one term of the linear combination, making e_i become a simpler linear combination with every iteration. For this to work, the next error e_{i+1} after a step i has to be orthogonal to the search direction d_i used in that step. If e_{i+1} is orthogonal to d_i, no part of e_{i+1} points in the same direction as d_i. Therefore, d_i is no longer a base vector making up e_{i+1}. From $d_i \cdot e_{i+1} = 0$, it follows that $\alpha = -\frac{d_i e_i}{d_i d_i}$[see She+94, page 22]. Calculating this, however, requires knowledge of e_i, which can only be calculated if the true solution x is already known.

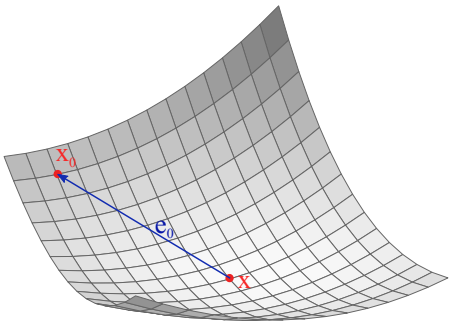


Figure 3.3: Geometrical representation of the initial error vector e_0. (figure adapted from [She+94]).

The CG algorithm gets around this issue by requiring A-orthogonality, or conjugacy, instead of orthogonality:

$$Conjugate(d_j, d_i) \iff d_j A d_i = 0 \tag{3.12}$$

With A-orthogonality as the requirement and $r_i = -Ae_i$, the formula for the step size α becomes:

$$\alpha = -\frac{d_i A e_i}{d_i A d_i} = \frac{d_i r_i}{d_i A d_i} \tag{3.13}$$

The only remaining issue is thus obtaining search directions conjugate to e_{i+1}. This is usually done through a process called Gram-Schmidt-Conjugation [see She+94, pages 25-26]. It requires a set of linearly independent base vectors u_0, u_1, ..., u_i in order to obtain the search directions d_0, d_1, ..., d_i. Normally, all previous d would need to be kept in memory to obtain d_{i+1}. However, the CG algorithm uses the residuals as the base vectors for Gram-Schmidt-Conjugation, greatly simplifying the process. The reason for this is that r_{i+1} is automatically A-orthogonal to all search directions except d_i [see She+94, pages 30-31]. As such, r_{i+1} can be used as the new search direction d_{i+1} after the part that is not A-orthogonal to d_i has been subtracted [see H+52, page 411]:

$$d_{i+1} = r_{i+1} - \frac{r_{i+1} A d_i}{d_i A d_i} \cdot d_i \tag{3.14}$$

As d_{i+1} only depends on r_{i+1} and d_i, CG's space and memory complexity is reduced significantly compared to using Gram-Schmidt-Conjugation with normal base vectors (such as the coordinate axes).
With the update schemes for step size α and search direction d defined, the complete Conjugate Gradient algorithm can be expressed as in Algorithm 2.

$x = x_0$; $r = b - Ax$;
$d = r$; $i = 0$;
while $max(|r|) > tolerance \wedge i < i_{max}$ **do**
  $\alpha = dot(d, r)/dot(d, Ad)$;
  $x = x + \alpha \cdot d$;
  $r = r - \alpha \cdot Ad$;
  $d = r - d \cdot dot(r, Ad)/dot(d, Ad)$;
  $i = i + 1$;
end

Algorithm 2: The Conjugate Gradient Algorithm

3.3.2 Convergence

The Conjugate Gradient algorithm can solve a linear system of equations defined by a $n \times n$ matrix in n steps, if round-off error is disregarded [see She+94, pages 32-33]. Depending on the spectrum, i.e. Eigenvalues of that matrix A, it can also be much faster. This can be measured by means of the condition number

$$\kappa(A) = \frac{|\lambda_{max}|}{|\lambda_{min}|}, \tag{3.15}$$

with λ_{max} and λ_{min} referring to the maximum and minimum Eigenvalues of A respectively. A low condition number κ leads to faster convergence, a larger κ implies more iterations will be needed.
In Eulerian fluid simulation, A arises from the discretized Navier-Stokes equations and the simulation domain. Thus, $\kappa(A)$ is fixed and cannot be optimized to speed up convergence. Pre-conditioning [see BM07, page 32] can be used if $\kappa(A)$ is large. This involves finding a matrix M that is a good approximate of A^{-1}. This way, $MAx = Mb$ can be solved instead of $Ax = b$, which is simpler due to $MA \approx I$.
Besides A, the other important factor in the convergence of CG is the initial guess x_0. We consider two criteria for the quality of an initial guess x_0:

1. The initial residual $r_0 = b - Ax_0$.
   This measures how far x_0 is away from fulfilling $Ax_0 = b$.

2. The iterations needed by CG to reach a desired accuracy when using x_0 as the starting point.

Finding a good initial guess is often dependent on the specific problem being solved. For incompressible fluid flow, the two most common approaches are choosing the solution of the previous time-step as the initial state, or simply using an all-zero initial guess [see BM07, page 33]. Using the previous time-step is effective for resting or slow-moving fluids, but does not help much for more turbulent fluids. Their pressure fields can change drastically between time steps, making this approach not suitable.

In this work, we therefore investigate a different strategy: Training an artificial neural network to predict a pressure field which can then be used as an initial guess for the CG solver.

4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialized type of Artificial Neural Network (ANN) that is often used in image recognition and processing. As Eulerian fluid simulation utilizes a grid, i.e. a regular, image-like data structure, CNNs are very well suited to the task of learning pressure fields for these simulations. In this chapter, we briefly review the core concepts and terminology related to CNNs. A more thorough, in-depth explanation may be found in [Nie15].

4.1 Artificial Neural Networks

A neural network is a collection of units called neurons. Loosely based on the anatomy of the brain, these neurons are interconnected and can transmit signals from one to another. Each neuron weights all signals it receives as input, sums them up and itself produces an activation or output signal. Mathematically, this can be expressed as:

$$a = \sigma(z) = \sigma(w \cdot x + b) \tag{4.1}$$

Here w, b, a and x are the weights, bias, activation and inputs of the neuron, respectively. Before the weighted sum z is transmitted as output, the activation function σ is applied. As σ, a variety of functions can be chosen. The important part is that it is a non-linear function, so that the output of the neuron is not merely a linear combination of its inputs.

These neurons are chained together, often in a layer structure, to form the neural network. Such an ANN starts with an input layer and, by consecutively evaluating its layers, produces a corresponding output. In this way, an ANN is theoretically able to calculate any function [see Nie15, page 127ff.]. Depending on which function should be learned by the network, the right values for the weights w and biases b must be found. These parameters are the trainable part of the network and are obtained through optimization, i.e. they are randomly initialized and gradually adjusted throughout training. During the training process, the network is given input data samples from a training dataset. The network's output for this data is calculated and, depending on it, the weights and biases are adjusted.

For this, it is necessary to define when the network's output is considered good or bad.

This is done by means of a loss function. In supervised learning, this loss function compares the network's output to the pre-computed solution for each training data sample, e.g. via a difference. In unsupervised learning, no pre-computed solution exists and the loss is instead determined by a mathematical measure based on the network's output itself. In either case the network's training goal is to minimize the loss function by adjusting its parameters.

4.1.1 Optimization

To minimize the loss function, we must derive it with respect to the network's weights and biases. This derivative indicates how the loss changes depending on the parameters. This in turn tells us how they should be adjusted to minimize the loss. As the loss function $\mathcal{L}$ is a scalar function depending on multiple trainable parameters W, this derivative is the gradient:

$$\nabla \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{W_0} \\ ... \\ \frac{\partial \mathcal{L}}{W_n} \end{pmatrix} \tag{4.2}$$

The intuition behind a gradient is that it is the vector pointing in the direction where the function increases the most. It then becomes clear that iteratively following the negative gradient leads us to minimize the loss. This is the fundamental idea behind the Stochastic Gradient Descent algorithm that is used to train neural networks. Of course, there are many more advanced variations of it. At their core, though, they all depend on the gradient. Unfortunately, the gradient of the loss function is not straightforward to compute efficiently. The backpropagation algorithm, which was popularized in the field of neural networks in [RHW86], was therefore a critical discovery.

Backpropagation The backpropagation algorithm is what enables training of neural networks as it is done today by providing an efficient way of approximating the gradient of the loss function. The algorithm starts computing the gradient at the output layer of the network. This is then backpropagated through the network, layer by layer, until the first layer is reached and the gradient has been determined for each weight. As an example, consider a neuron in the output layer l.

$$a^{(l)} = \sigma(z^{(l)}) = \sigma(w^{(l)} \cdot a^{(l-1)} + b^{(l)}) \tag{4.3}$$

For a given training input, we can compute the network's output and thus this neuron's activation $a^{(l)}$ and the current loss $\mathcal{L}_0$. Because $\mathcal{L}_0$ is directly based on the activations of the output neurons $a^{(l)}$, it is possible to calculate $\frac{\partial \mathcal{L}_0}{\partial a^{(l)}}$, i.e. how the loss changes

depending on the activation. Using this and the chain rule from calculus, the gradient for the output neuron can be rephrased:

$$\frac{\partial \mathcal{L}_0}{\partial w^{(l)}} = \frac{\partial z^{(l)}}{\partial w^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial \mathcal{L}_0}{\partial a^{(l)}} \tag{4.4}$$

All terms on the right side can be calculated by deriving $\mathcal{L}$ and Equation 4.3. In this manner, we can therefore calculate the gradient for every neuron in the output layer. For the next layer, $l-1$, the same procedure can be used if $\frac{\partial \mathcal{L}_0}{\partial a^{(l-1)}}$ is known. This, too, can be obtained by using the chain rule:

$$\frac{\partial \mathcal{L}_0}{\partial a^{(l-1)}} = \frac{\partial z^{(l)}}{\partial a^{(l-1)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial \mathcal{L}_0}{\partial a^{(l)}} \tag{4.5}$$

Using $\frac{\partial \mathcal{L}_0}{\partial a^{(l-1)}}$, the gradient of layer $l-1$ can then be computed as in Equation 4.4. This process can then be repeated for every layer, until the full gradient of $\mathcal{L}$ with respect to every parameter is known. Of course, in practice, these equations become somewhat more complicated, since the influences in Equation 4.4 and Equation 4.5 need to be summed up for all connected neurons. For a more thorough and complete explanation of the backpropagation algorithm, we therefore refer to [Nie15, page 39ff.].

4.2 Convolutions

Non-specialized ANNs are often arranged in layers, with the output of every neuron in a layer acting as an input to every neuron in the next. This network architecture is often referred to as fully-connected and comes with some disadvantages. It leads to a high number of trainable parameters for even relatively shallow networks and small layer sizes. As a result, fully-connected networks are prone to overfitting to the training dataset. This means that the network fine tunes its parameters specifically to perform well on the training dataset without learning to generalize to unknown data. Additionally, an unnecessarily high parameter count slows down training drastically. Particularly for input data with spatial structure, such as the grids in Eulerian fluid simulation, fully connected networks are not optimal. For example, they treat grid cells that are far apart in the same way as ones that are close together [see Nie15, page 169].

Convolutional Neural Networks (CNN), popularized in [LeC+98], are a variant of neural network that specifically takes advantage of the spatial structure of the input and thereby alleviates the aforementioned issues. The namesake of these networks is the convolution operation. Instead of connecting every neuron in a subsequent layer to all neurons of the previous layer, a local receptive field is defined. This is a

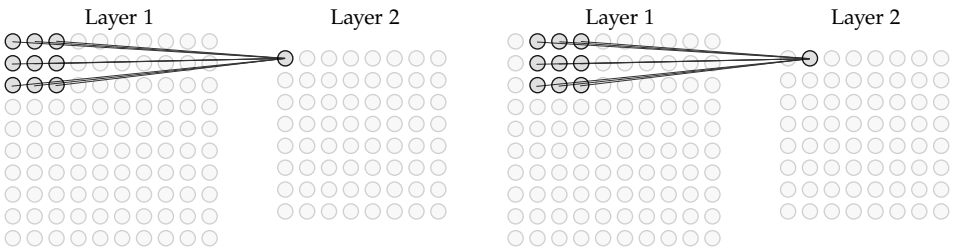


Figure 4.1: Local receptive fields in Convolutional Neural Networks.

small window into the input layer that is also referred to as kernel or filter. During convolution, this window is slid across the entire input layer (see Figure 4.1). The increments in which it is moved are called the stride of the convolution. At each position, the neurons within the kernel are connected to the corresponding neuron in the next layer. In the example in Figure 4.1, every neuron in layer 2 therefore receives the activations of 3×3 neurons from the previous layer as input. Like in section 4.1, the neuron then multiplies each input value by a weight, adds a bias and finally applies the activation function. Unlike an ANN, the weights and bias are shared across all the neurons in the second layer. One particular filter thus only has $s_k \times s_k + 1$ trainable parameters, with s_k referring to the kernel size. Therefore, every filter learns to apply one operation, but at different positions throughout the input. As CNNs are heavily used in image recognition, common terminology is that every filter detects one feature of the input it is applied to [see Nie15, page 172].

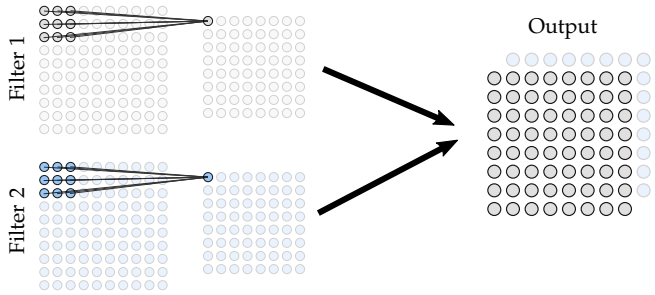


Figure 4.2: Stacking feature maps in Convolutional Neural Networks.

Most interesting operations on images and other grid-like data consist of many such features, though. Therefore, in every convolutional layer, multiple features are usually trained. Each filter, when applied to the input, produces a one dimensional layer of neuron activations which is also known as a feature map. As illustrated in Figure 4.2, these feature maps are stacked together in the feature dimension. For a two dimensional input layer, the output would thus have the shape width $\times$ height $\times$ features.

The stacked output can then be used as the input for the next convolutional layer, and so on.
As convolutional layers are applied consecutively, the detected features grow more abstract. Later layers no longer learn features in the input itself, but in the feature maps of earlier layers. However, if the kernel size remains small in relation to the spatial dimensions of the input, even later layers can only detect small-scale features. This would render the CNN unable to learn more global properties of the input, such as e.g. recognizing images whose upper half is generally brighter than their lower half. Simply increasing the kernel size would, however, increase parameter count and thereby training and computation time. To avoid this, CNNs often gradually downscale their image in the spatial dimension, causing the kernel to cover a proportionally larger area in the input.

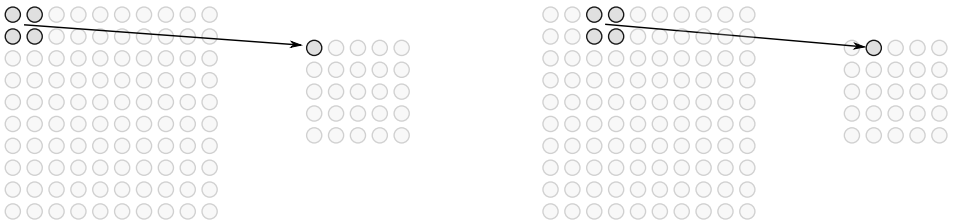


Figure 4.3: The max pooling operation in Convolutional Neural Networks.

The max pooling operation (shown in Figure 4.3) is commonly used for this purpose. It pools together the activations in a window (commonly 2×2) by taking their maximum. The information in the feature map is thereby simplified and the result contains more global information. In order to produce detailed outputs, the spatially reduced, abstracted feature maps must eventually be upsampled again. For this, the technique of transpose convolution or deconvolution is usually employed. This operation is essentially a convolution, but in reverse. Consider how in Figure 4.1, nine input neurons are connected to one output neuron. In deconvolution, this is done the other way around. The activation of every neuron in the input is distributed to a kernel-sized region in the output. As with convolution, this operation contains trainable parameters determining how exactly this distribution occurs.

In summary, convolution enables the CNN to process features that can occur at different spatial positions in the input data. Pooling then lets the network abstract the information and deconvolution allows it to produce detailed outputs from these abstractions. Because of this, CNNs are a good choice for our aim of predicting a pressure grid from an input divergence field. The trained models in this work therefore use a convolutional structure, albeit with some variations (see section 5.1).

5 Neural Network-Based Pressure Prediction

The goal of this work is to investigate how to train a Convolutional Neural Network (CNN) to predict the pressure $\hat{p}$ that can be used to make the input velocity field u divergence-free. Of particular interest is the quality of these predictions not just on their own, but with respect to being used as an initial guess for the CG solver. In the following, we refer to the trained CNN using the operator $\mathcal{C}$, so that $\mathcal{C}(u) = \hat{p}$. We focus on 2D cases, hence $u \in \mathbb{R}^{2 \times d_x \times d_y}$ and $\hat{p} \in \mathbb{R}^{d_x \times d_y}$, where d_x and d_y are the dimensions of the simulation domain.

In this chapter, we will present a solver-based approach to learning pressure fields and compare it with two others that represent more standard methods in this area. For better comparability, we only vary the loss function (section 5.2) between them, keeping the network architecture (section 5.1), training and validation dataset (section 5.3), as well as the training procedure (section 5.4) constant.

5.1 Network Architecture

We use the divergence ∇u as input for our neural networks. Thus, both their input and output are scalar fields, i.e. $\nabla u, \hat{p} \in \mathbb{R}^{d_x \times d_y}$. For CNNs whose spatial input and output dimensions are the same, U-Net structures are a natural fit [Ada20]. We therefore based our network architecture on that proposed in [RFB15]. The overall network layer structure, including kernel sizes and feature maps, is illustrated in figure 5.1.

The network consists of a contracting path (left) and an expanding path (right). They are sometimes also referred to as Encoder and Decoder respectively. In the contracting part, the spatial dimension of the data is successively reduced, while the feature dimension increases. This allows the CNN to learn more global features that span a larger part of the domain, compared to e.g. a ResNet-based architecture [He+16]. Our Encoder has three levels. In each level, two 5x5 convolutions are applied, each followed by an activation function. Then, the spatial dimension is reduced. In the original U-Net, this is achieved via a max pooling operation at the end of each level

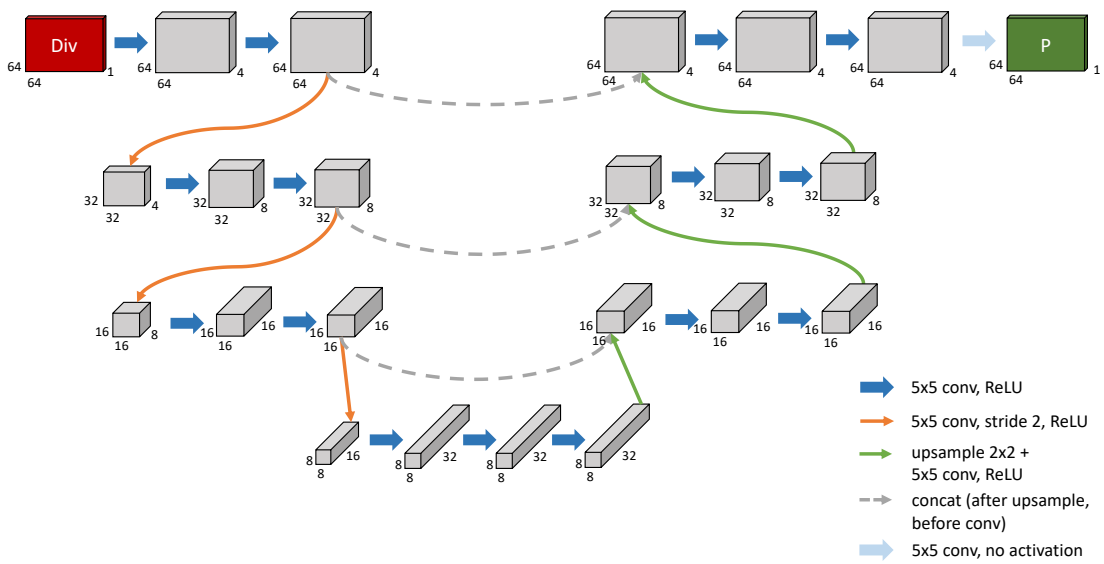


Figure 5.1: A visual summary of the U-Net-based architecture used in this work

[RFB15]. However, recent studies have shown that max pooling can and often should be replaced by a strided convolution layer [Spr+14]. This introduces some learnability to the down-sampling layer, making it less rigid. We therefore opted for a 5x5 convolution with stride 2 instead of max pooling. Each contractive level thus halves the spatial dimensions and simultaneously doubles the feature dimension.

At the lowest level of the network, the spatial dimension is reduced by a factor of 2^3 and three more convolutions with 32 features are performed. After this, the network begins to expand again to gradually return to its initial shape. In this expansive path, each level starts with an up-sampling layer: The input is linearly interpolated to double its spatial dimensions. Then, three consecutive, activated 5x5 convolutions are applied, using half the amount of features of the previous level. Though each level halves the feature dimension, this still leads to a large number of feature channels in the Decoder. This is an important characteristic of U-Nets and allows the network to propagate context information to higher resolution layers [see RFB15, page 3].
After three up-scaling levels, there is one final, activationless convolution to bring the output into the shape $d_x \times d_y \times 1$. Overall, our network structure has 22 convolutional layers and 127,265 trainable parameters, i.e. weights and biases.

Skip Connections Deep neural networks often suffer from the vanishing gradient problem. This occurs due to the use of the chain rule in back-propagation. As the gradient is propagated from the last layers of the network to earlier ones, the chain rule requires multiplication of partial derivatives at each layer [Ada20]. Since these are often values in $(0, 1)$, the gradient gets smaller and smaller for every layer it is back-propagated to.
To counter this, our architecture utilizes (long) skip connections. As visualized in figure 5.1, each Encoder level's last output before down-scaling is concatenated to each Decoder level's input after up-scaling. This ensures an uninterrupted gradient flow from later layers to earlier ones. It also helps to reintroduce detailed spatial information into the Decoder that would otherwise be lost due to down-scaling in the Encoder [Ada20].

Activation Function As the activation function for each convolution, we chose Rectified Linear Unit (ReLU), i.e. $f(x) = \max(0, x)$. Activation functions are necessary to introduce non-linearity into a network. Otherwise, the CNN would just be a chain of linear operations and could thus itself only output a linear transformation of its input. ReLU is computationally more efficient than classic non-linear functions like hyperbolic tangent or the sigmoid function. Additionally, its activation is not confined to $(0, 1)$, which helps to further alleviate the vanishing gradient problem.

5.2 Loss Functions

The choice of loss function is one of the most important factors in training a neural network, since its gradients with respect to the network weights determine how the weights change in each optimization step. We compare three variations: A basic supervised approach (SUP), an unsupervised loss informed by the physical meaning of the network prediction (PHY), as well as our proposed solver-based method (SOL_k). The latter uses a differentiable solver directly in the loss (see subsection 5.2.3), in line with recent research into differentiable physics in Deep Learning [HKT20; Um+20].

For all loss function variations, we apply the L2-Norm as a last step. This operation is defined as $\|v\|^2 = \frac{1}{2}\sum v^2$, i.e. the sum of the squared components of the loss vector. Since the loss vector is usually a difference, squaring the components before summing prevents terms with different signs from cancelling each other out. The factor $\frac{1}{2}$ is used instead of a square root to improve performance.

5.2.1 Supervised Loss

The first approach (SUP) employs a regular supervised loss. It minimizes the difference of the network's prediction from pre-computed ground truth pressures p. These ground truth samples are obtained by running the CG solver up to a fixed accuracy of 10^{-6} and are saved as part of the dataset.

$$\mathcal{L}_{\text{SUP}} = \|\mathcal{C}(u) - p\|^2 \tag{5.1}$$

Though $\mathcal{L}_{\text{SUP}}$ makes the CNN match the solver result as closely as possible for each training sample, it does not provide any information on the physical meaning of a prediction to the network. Pressure predictions that lead to velocity divergence are not penalized as long as their overall difference to the ground truth remains small.

5.2.2 Physical Loss

Our second approach (PHY) directly aims for the same physical goal as the CG solver. Similar physics-informed losses have been employed in previous works [RPK17; RYK18; SS18]. Specifically, our variant is based on the approach proposed by Tompson et. al. [see Tom+17, page 4], which has proven to yield good pressure predictions that can be used in a simulation as-is. The idea is to minimize the remaining divergence after the input velocity u has been corrected with the predicted pressure (see 3.5). This is equivalent to minimizing the CG-solver's residual $r = \nabla u - \Delta p$ (see equation 3.9).

$$\mathcal{L}_{\text{PHY}} = \|\nabla u - \Delta \mathcal{C}(u)\|^2 \tag{5.2}$$

$\mathcal{L}_{\text{PHY}}$ is an unsupervised loss, i.e. an error metric that is directly based on the network's prediction, without including any solver-generated terms. Therefore, the PHY-network learns to aim for the same goal as the solver, but independently of it.

5.2.3 Solver-based Loss

For the CNN to produce pressure guesses that are useful as initial states for the CG solver, the network should ideally receive feedback on the solver's behavior at training time. We therefore propose a method that incorporates the iterative CG-solver directly into the loss function.

$$\mathcal{L}_{\text{SOL}_k} = \left\| \mathcal{S}^k(\mathcal{C}(u)) - \mathcal{C}(u) \right\|^2 \tag{5.3}$$

This requires a differentiable version of the CG-solver, so that gradients can be back-propagated through it. In this work, we make use of the Φ_{Flow} fluid simulation toolkit [see HKT20, page 2], which was developed with such differentiable physics learning

tasks in mind. The linear operator $\mathcal{S}$ denotes the evaluation of one differentiable solver step, given an input initial guess. $\mathcal{S}^k = \mathcal{S}(\mathcal{S}(...))$ refers to k successive solver iterations. The network thus receives gradients through k solver evaluations based on its output. It can therefore see how the solver performs based on its prediction and learn accordingly. Additionally, since two successive pressure guesses $\hat{p}_j$, $\hat{p}_i$ (with $j > i$) have the property

$$|\hat{p}_j - \hat{p}_i| < |\hat{p}_j - p| \text{ [see H+52, page 416],} \tag{5.4}$$

it follows that minimizing $\left\|\mathcal{S}^k(\hat{p}_j) - \hat{p}_j\right\|^2$ makes the network converge towards the true solution p without the need to pre-compute ground truth samples.

As the difference $\hat{p} - p$ converges to zero, the residual r decreases. Thus, both $\mathcal{L}_{\text{PHY}}$ and $\mathcal{L}_{\text{SOL}_k}$ minimize the residual divergence. But while $\mathcal{L}_{\text{PHY}}$ minimizes it directly, $\mathcal{L}_{\text{SOL}_k}$ does so via the convergence of the solver instead.

5.3 Dataset

The dataset used to train, validate and test our CNN models was generated by running randomized numerical simulations using the Φ_{Flow}-framework [HKT20]. Each simulation starts out from a randomly generated velocity and density field, on a domain of size $d_x \times d_y = 64 \times 64$.

Random Generation of Density and Velocity Fields The initial density and velocity fields are generated by taking a normally distributed, 2D frequency spectrum, then performing Discrete Inverse Fourier Transformation (DIFT) to convert it to the corresponding 2D signal. Specifically, the following steps are taken:

1. Generate a normally distributed 2D scalar field $\tilde{\mathcal{N}} = \mathcal{N} + 1j \cdot \mathcal{N}$. Here, $\mathcal{N}$ is a random 64×64 field sampled from a standard normal distribution. Since $\tilde{\mathcal{N}}$ is the sum of a real normal distribution and an imaginary one, its values are complex numbers.
2. Generate a 64×64 field $\mathcal{F}$ of frequency bins.
3. Multiply $\tilde{\mathcal{N}}$ by $256 \cdot (\frac{1}{\mathcal{F}+1})^{32}$ to scale the probabilities of the frequencies in $\mathcal{F}$. $\mathcal{F}$ becomes a normally distributed 2D spectrum of frequencies.
4. Apply DIFT to the frequency spectrum. The result is a 2D signal, i.e. a combination of the *sin* and *cos* functions described by the spectrum.

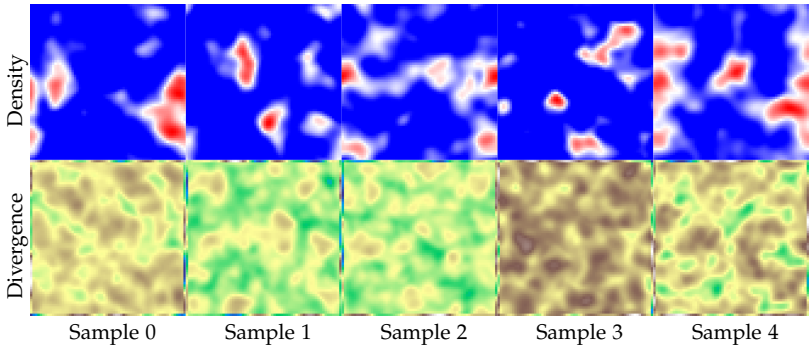


Figure 5.2: Density and velocity divergence samples from the training dataset.

Examples of the generated fields can be seen in Figure 5.2. From this random initial state, 16 frames are simulated with a CG pressure solver up to an accuracy of 10^{-6} maximum residual divergence per frame. We use a buoyancy factor of 0.1 and closed domain boundaries. Every time step, the current advected velocity, its divergence field, and the pressure field used to correct it are saved to disk. The dataset consists of 3,000 such simulations in total, resulting in $3,000 \cdot 16 = 48,000$ data samples. Of these, 2800 simulations (44,800 samples) were used as training data and the remaining 200 (3200 samples) as validation data. We additionally generated a separate test dataset comprised of another 3,000 simulations to evaluate the trained models on in chapter 6. This was created in the same way, but using a solver accuracy of 10^{-3} instead.

5.4 Training Procedure

To train our models, we iterate through the training dataset sequentially in batches of 32 samples. For each batch, a stochastic gradient descent optimization step is performed. We utilize the Adaptive Moment Estimation (ADAM) scheme for this [KB14]. This optimizer is widely used, as it is an efficient first-order method that builds momentum along those dimensions of the cost function whose gradients do not change directions much. It thereby mimics a ball rolling down a slope, more easily overcoming local minima in the loss landscape. All CNNs we compare in this work were trained for a total of 300,000 ADAM optimization steps, using a learning rate of $2 \cdot 10^{-4}$. Their parameters (weights and biases) were initialized using the method proposed by Glorot and Bengio in [GB10]. This initialization scheme draws the parameters from a Gaussian random distribution with mean zero and a variance based on the number of inputs and outputs of the corresponding neuron. Apart from the randomness resulting from this parameter initialization, the training procedure is identical across all trained models, making them easily comparable.

6 Results

In this chapter, we evaluate the pressure prediction capabilities of the three approaches (SUP, PHY and SOL_k) described in chapter 5. In particular, we focus on their performance in conjunction with the CG solver. For each approach, we trained a representative model (see section 6.1), choosing SOL_5 for the solver-based variant (see subsection 6.4.1 for an exploration of different look-ahead step sizes k). We compare how the models perform on test data (see section 6.2), as well as in-simulation (see section 6.3). In our comparisons, we additionally include an all-zero initial guess as a baseline.

Input Normalization When using the trained models to infer a pressure field, we first normalize the input. Specifically, we calculate the standard deviation σ of the divergence field. We then divide the divergence field element-wise by σ, before feeding it into the trained network. Lastly, we denormalize the network's output by multiplying it by σ again. As the divergence fields are already zero-centered, it is unnecessary to subtract their mean.
This normalization process is possible because the pressure equation the network is solving, $Ap = d$, is linear. It has the advantage of making pressure prediction independent of the scale of the input divergence. The trained models should thus be able to handle very strongly and weakly divergent inputs equally well.

6.1 Training Results

The models were trained on an Nvidia GeForce RTX 2080 Ti, using the training procedure detailed in section 5.4. They took 22 hours 4 minutes (SUP), 22 hours 21 minutes (PHY) and 23 hours 15 minutes (SOL_5) to train, respectively. Figures 6.1, 6.2 and 6.3 visualize their training processes. The left side shows how the loss decreased over the course of training. On the right, it is shown how many iterations the CG solver needs to reach an accuracy of 10^-3, given the network output as an initial guess. For both, the blue line represents batches from the training dataset and the orange line validation data batches. It is important to note here that we used a batch size of 32 for training and 16 for validation, which is why the validation loss is generally lower than the training loss.

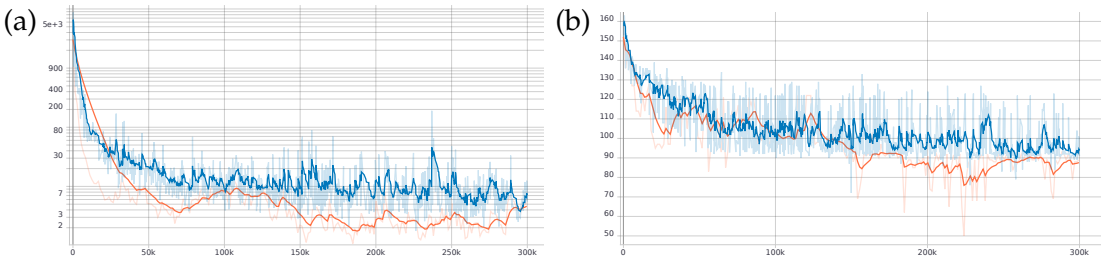


Figure 6.1: SUP, (a) Loss (b) CG iterations for accuracy 10^{-3} with network guess.

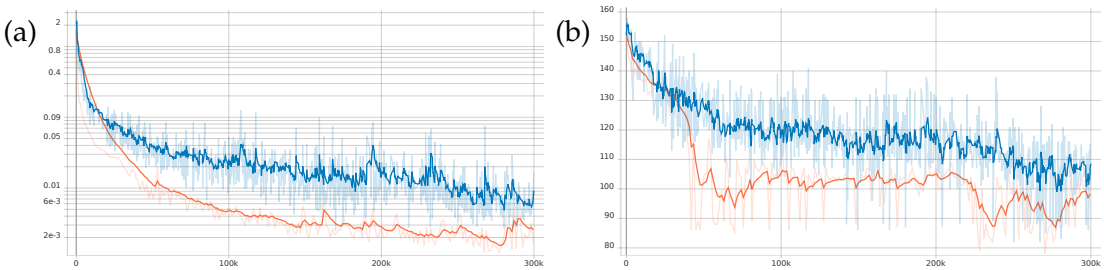


Figure 6.2: PHY, (a) Loss (b) CG iterations for accuracy 10^{-3} with network guess.

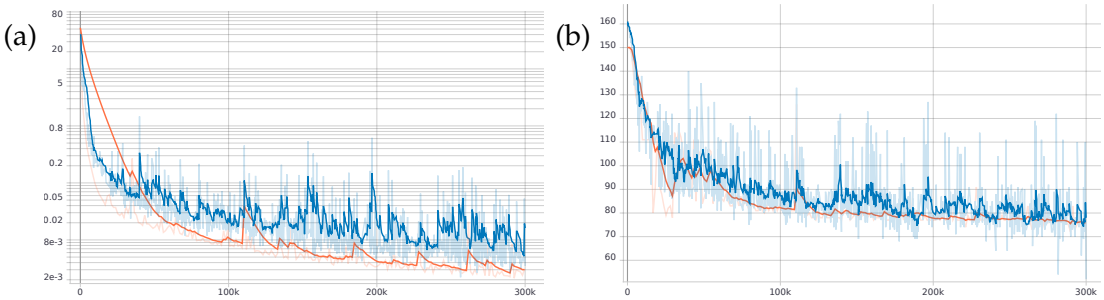


Figure 6.3: SOL_5, (a) Loss (b) CG iterations for accuracy 10^{-3} with network guess.

For all three models, the loss decreases steadily throughout training. Using a logarithmic scale and a smoothing factor of 0.8, we can observe that none of the models' training loss fully stagnates by the 300,000 step mark. PHY and SOL_5 in particular appear to still be improving at that point. As such, training these networks for longer might further improve their performance.
The CG iterations also decrease for all three models as their pressure predictions improve. However, there is a clear ranking to be observed. SOL_5 clearly does the best, reducing the maximum required iterations for a training batch from roughly 160 to below 80 by the end of training. SUP manages around 90, while PHY does the worst with more than 100. PHY however also shows a large gap between validation and training

data. This implies that the quality of its predictions varies more strongly depending on the divergence data, with the validation set evidently containing favorable data. To more accurately assess the generalization capabilities of the networks, it is therefore necessary to test them on a large set of varying divergence fields not seen at training time.

6.2 Test Dataset Performance

To gauge how well the trained models perform on unknown data, we first evaluate them on the test dataset (see section 5.3), i.e. on sample divergence field inputs the networks did not observe during training. For all results in this section, we computed the average of 100 such test cases. These 100 cases are randomly sampled from the test set once in the beginning. They are therefore random but consistent across the compared models.

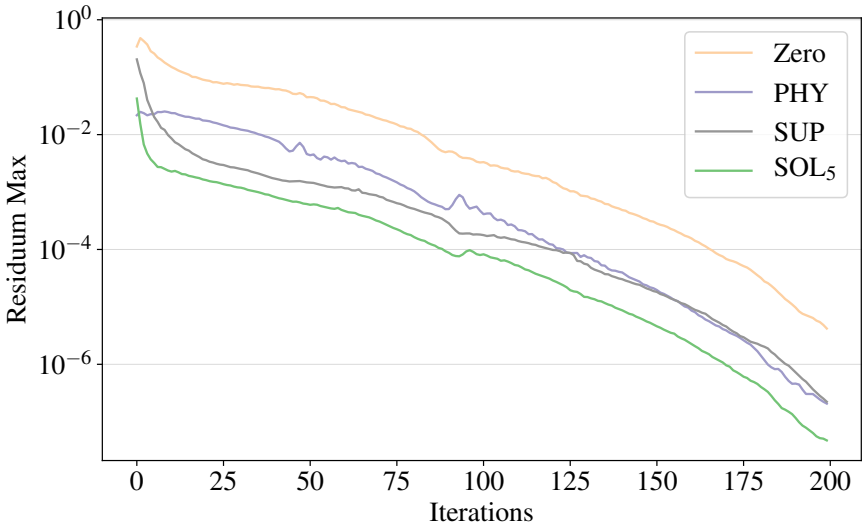


Figure 6.4: CG solver residual error, given initial guesses by different models.

To understand the quality of the models' outputs as initial pressure guesses, we observe the CG solver's behavior when starting from these outputs. Figure 6.4 depicts how the residual divergence decreases over the solver's iterations. The starting point of each curve represents the accuracy of the model's guess itself, without any further solver iterations. Here, the PHY model fares the best. It reaches an accuracy of nearly 10^{-2} out of the gate, closely followed by SOL_5. SUP's output is much less accurate and

only a slight improvement over the zero-guess baseline, even though its predictions seem visually close to the reference (see Figure 6.6). This is due to the residual error being measured locally per grid point. For a low residual, a low sum of the grid cells' errors is more important than matching the overall structure of the true solution. Correct large-scale structures in the output only become more relevant as the solver iterates.

Over the first few iterations, the accuracy of SOL_5 improves significantly, quickly overtaking PHY. SUP also shows rapid improvement, though it is less steep than that of SOL_5 and starts from a much worse accuracy. By contrast, the PHY approach shows barely any improvement in the initial iterations. When running the solver for more iterations, the increase in accuracy becomes more similar across the three models, with SOL_5 retaining the advantage obtained in the first few iterations.

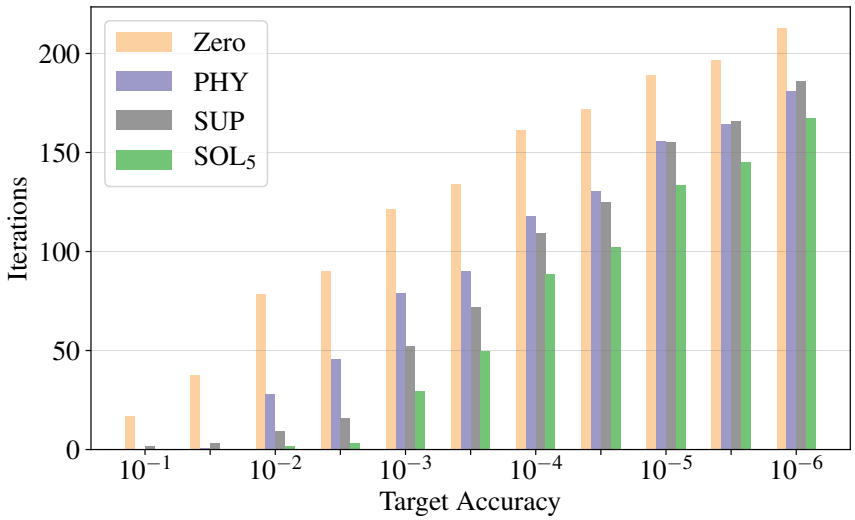


Figure 6.5: Visual comparison of required CG iterations, given initial guesses by different models.

The rapid improvement in accuracy facilitated by the SOL_5 guess means fewer iterations are needed to reach higher accuracies. This is shown in Figure 6.5, which visualizes how many CG iterations are required for a specified target accuracy given the pressure solutions inferred by each model. The numerical results are listed in Table 6.1. An accuracy of 10^{-2}, for example, requires an average of around two solver steps for SOL_5, nine steps for SUP, 28 steps for PHY and 78 steps when starting from a zero guess. SOL_5 consistently requires the least CG iterations to reach the target. The only

exception are the lowest accuracies, which only require very few CG iterations. The supervised model initially also shows a good reduction in needed iterations, but loses this advantage compared to PHY the closer the target accuracy gets to the accuracy of the ground truth it was trained with (see subsection 5.2.1). By contrast, SOL_5 retains its advantage even for higher accuracies.

Model	**Iterations for Target Accuracy**, Mean (Standard Deviation)					
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
Zero	16.96	78.48	121.61	161.57	189.03	212.86
	(10.512)	(9.138)	(13.443)	(10.294)	(6.172)	(5.365)
SUP	1.67	9.33	52.16	109.12	155.37	186.12
	(1.010)	(5.428)	(17.540)	(15.875)	(10.155)	(5.719)
PHY	0.0	27.79	79.06	117.97	155.76	181.07
	(0.0)	(15.255)	(10.042)	(13.234)	(9.403)	(6.052)
SOL_5	0.03	1.97	29.59	88.37	133.59	167.37
	(0.171)	(1.118)	(14.832)	(13.465)	(11.605)	(8.549)

Table 6.1: Average CG solver iterations required for target accuracies, given initial guesses by different models.

Overall, these results reveal an interesting dichotomy. On one hand, the PHY model produces pressure fields which lead to the least amount of residual divergence. They can therefore be considered the best pressure guesses if used as-is. On the other hand, when using the network outputs in conjunction with the CG solver, our solver-based approach outperforms PHY noticeably.

This shows the significance of training with the solver in the training loop. The PHY model only learns to directly minimize the residual, measured per grid point. It does not receive any feedback on the solver's behavior during training. Therefore, it does not learn to match the true solution on a global scale. The CG solver then has to retroactively work out the larger structures, leading to more iterations. By contrast, the SOL_5 model sees how the solver corrects its prediction at training time, enabling it to adjust its guess accordingly. The supervised approach learns the global structure of the true solution to some extent as well, due to being fed pre-computed solutions. Like PHY, however, it does not receive information on the solver's behavior during training and is consistently outperformed by SOL_5.

Investigating the inferred pressure fields themselves further corroborates these findings. As can be seen in Figure 6.6, the PHY model's outputs have significant large-scale differences from the reference solution. Both SUP and particularly SOL_5 come much closer to the reference. Looking at the residual divergence (see Figure 6.7), PHY also produces a noticeable error pattern near the domain border. These border-cell errors are small individually but significantly influence the solution as a whole. The physics-based loss consequently does not give a lot of weight to them, whereas SOL_5 learns about the solver's behavior at the borders directly. Again, the lack of solver feedback during training leads to the PHY model being unable to learn these more global influences.

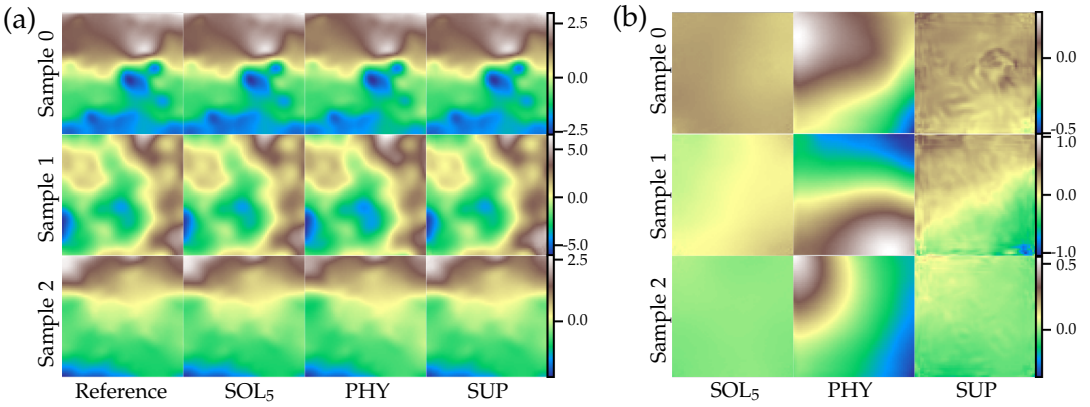


Figure 6.6: (a) Sample outputs of the models (b) Difference of output from reference.

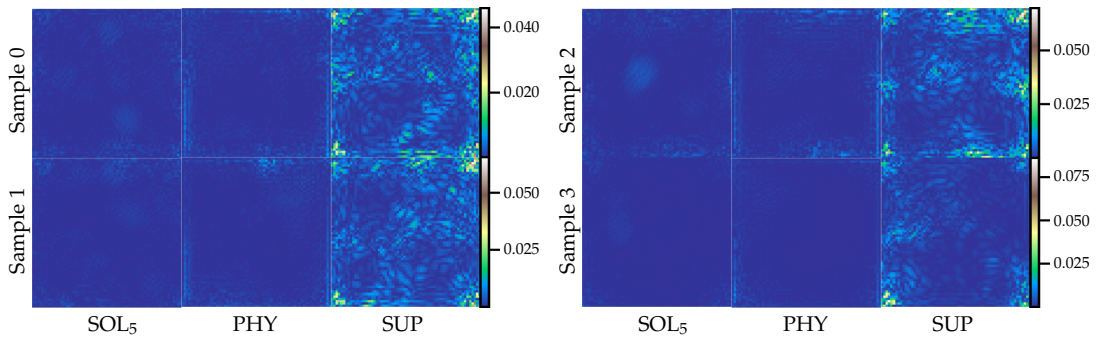


Figure 6.7: Absolute residual divergence after correcting the velocity with different models' outputs and one additional CG solver iteration.

6.3 Simulation Performance

In this section, we apply the trained models to actual fluid simulations. We take two different approaches to this. First, we use the trained CNNs directly as pressure predictors in a simulation, fully replacing the CG solver (subsection 6.3.1). In our second approach, we use the CNNs and CG solver in conjunction to compute the pressure for each time step (subsection 6.3.2).

6.3.1 Neural Network Pressure Solve

The most straightforward way of using a trained pressure predictor in a fluid simulation is to directly use it instead of the numerical pressure solving step. This allows a qualitative visual analysis of the prediction quality of each trained network. Our setup for these neural network simulations is as follows:

1. Load the trained model $\mathcal{C}$
2. Generate a random velocity field u and density field ρ (see section 5.3)
3. Simulate for 150 time steps:
   a) Render the current density field ρ as an image
   b) Advect u and ρ using a Semi-Lagrangian scheme (see subsection 3.2.3)
   c) Apply buoyancy by subtracting $0.1 \cdot \rho \cdot g$ from u, where $g = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix}$
   d) Apply boundary conditions to u
   e) Calculate the divergence ∇u
   f) Obtain the pressure by feeding the divergence into the trained CNN, i.e. $\hat{p} = \mathcal{C}(\nabla u)$
   g) Correct the velocity by subtracting $\nabla \hat{p}$

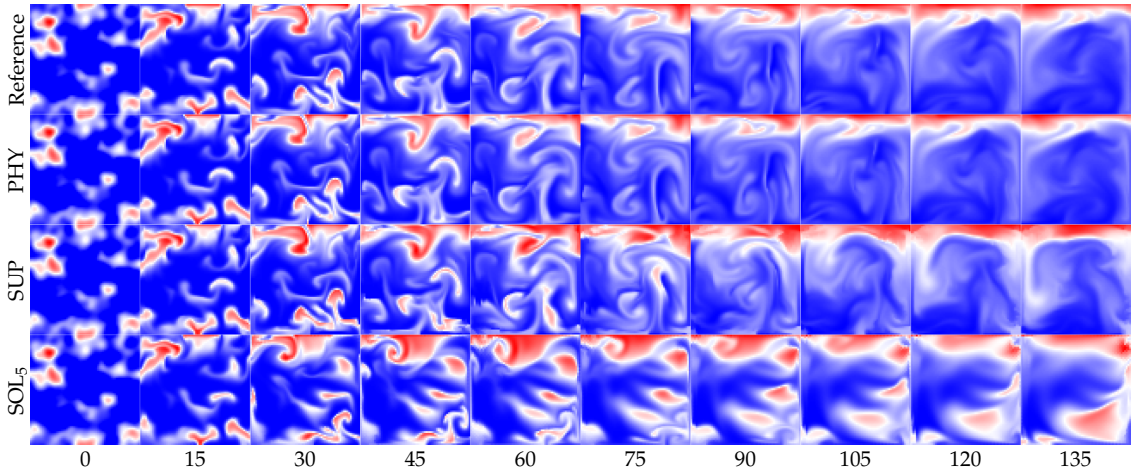


Figure 6.8: Example simulation using different trained models as pressure solver.

An example of such a neural network simulation can be seen in Figure 6.8. It shows how the density field evolves over time with different CNNs as pressure predictors, as well as with a standard CG pressure solve (target accuracy 10^{-3}) as reference. We only show every fifteenth simulation frame here, but the full simulation may be viewed in video form in our supplemental material.

All simulations shown in Figure 6.8 start out from the same density and velocity fields. Initially, they are thus identical. The further the simulations progress, however, the more strongly they deviate from the reference. This can be seen as early as frame 15 for SOL_5. Throughout the SOL_5 simulation, this error accumulates, making the fluid's behavior quite implausible from frame 45 onward. SUP retains its similarity to the reference a little longer, noticeably starting to deviate around frame 45. PHY does the best, staying very close to the reference simulation visually, with only minor artifacts visible in e.g. frames 90 and 105. Its simulation is plausible enough that it presents a viable alternative to the CG solver if high accuracy is not required. PHY's in-simulation performance is not unexpected, as its pressure predictions also showed high initial accuracy (nearly 10^{-2}) on the test dataset (see Figure 6.4). It therefore makes sense for it to perform similarly to the 10^{-3} accuracy reference. What is more surprising is the significantly worse performance of SOL_5, given that its initial accuracy on the test dataset (without additional solver iterations) is only slightly lower than PHY's.

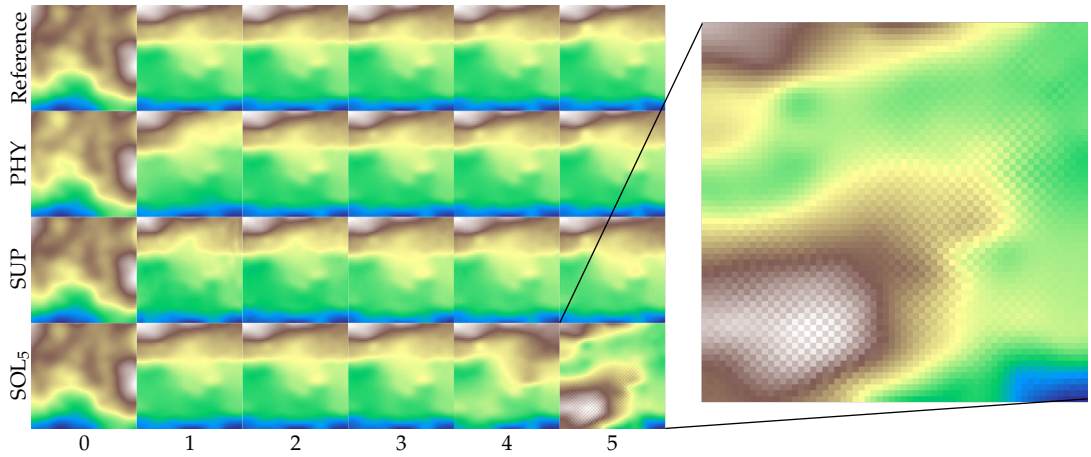


Figure 6.9: Pressure outputs of the trained models throughout the example simulation.

Looking at the pressure outputs for this example simulation (see Figure 6.9), we see that SOL_5's outputs match the reference pressure very well initially, but start to deviate strongly around time step 4 to 5. From that point on, the SOL_5 pressures also show a notable checkerboard pattern. This kind of pattern is usually associated with a mismatch of kernel size and stride when performing deconvolutions in a CNN (see [ODO16]). However, instead of deconvolutions, our architecture uses linear upsampling, followed by normal convolution and should thus not suffer from this issue. Furthermore, all models use the same architecture but only SOL_5 exhibits these artifacts. It is therefore unlikely to be an architectural issue.

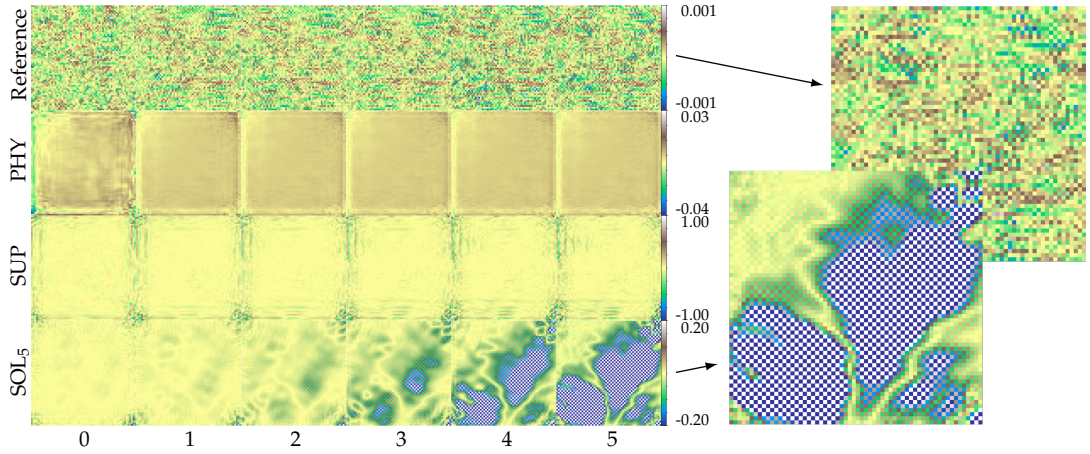


Figure 6.10: Residual divergence throughout the example simulation after successively correcting the velocity with the pressure output of different trained models.

Instead, the residual divergence (shown in Figure 6.10) suggests a different reason for the pattern. Evidently, it is the result of correcting the input divergence with SOL_5's pressure guess, as it appears immediately after the first correction, i.e. in frame 0. At this point, SOL_5's pressure is still very close to the reference with respect to the overall structure of the pressure solution (cf. Figure 6.9) - more so than PHY, for example. Correcting with it successfully reduces the overall divergence, but introduces locally alternating positive and negative grid cells. These cells' residual values are of similar magnitude. Their main difference is their sign.

This explains why SOL_5's accuracy is high on the test set, yet its in-simulation performance is lacking. Accuracy is measured as the maximum absolute value of the residual, since that is the convergence criterion of the CG algorithm (see Algorithm 2). Because the absolute value is used, the alternating positive and negative cells do not affect the accuracy. When used in-simulation, however, they cause the residual error to accumulate much more quickly. This is likely because the training dataset does not contain any divergence fields that feature more erratic patterns such as the checkerboard artifacts, as we used a CG solver with very high accuracy to generate them (see section 5.3). The SOL_5 network consequently cannot deal well with receiving such inputs, making its next pressure prediction introduce even stronger artifacts. Continually feeding SOL_5's residual divergence back to it as input, as is done when it is used as a standalone pressure predictor in a simulation, thus exacerbates this error.

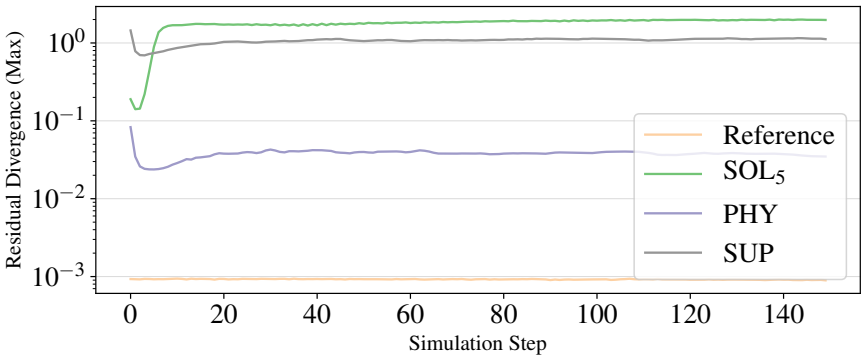


Figure 6.11: Average residual divergence over the course of a neural network simulation.

To quantify these findings and show that they are not just the case for this example simulation, we evaluated the maximum absolute residual over 100 randomized simulations. The average is shown in Figure 6.11. The residuals start out much as

on the test dataset (see Figure 6.4), with PHY achieving the best accuracy, followed by SOL_5 and then SUP. All three curves decrease initially. This is because the generated starting velocity field is not divergence-free. The initial divergence is much larger than that introduced by advection and buoyancy in subsequent time steps. All models successfully reduce this starting divergence over the first few steps. After this, however, the residuals begin to increase again as the networks repeatedly receive input divergences based on their own pressure corrections. Here, we can observe the effect of SOL_5's checkerboard pattern. SOL_5's pressure corrections lead to input divergences that are strongly dissimilar to the divergence data it was trained on. Consequently, SOL_5's residual accumulates much more quickly and drastically compared to the other models'. By contrast, both SUP'S and PHY's residual increases more gradually and stabilizes as the fluid becomes less turbulent. They balance out at an average of 1.070 and 0.038 respectively (see Table 6.2).

	CG (10^{-3})	**CG** $(4 \cdot 10^{-2})$	**SOL_5**	**PHY**	**SUP**
Time (avg.)	0.208 ms	0.105 ms	0.054 ms	0.056 ms	0.054 ms
Residual Error (avg.)	0.001	0.039	1.796	0.038	1.070

Table 6.2: Average residual and per time step computation time over 100 simulations.

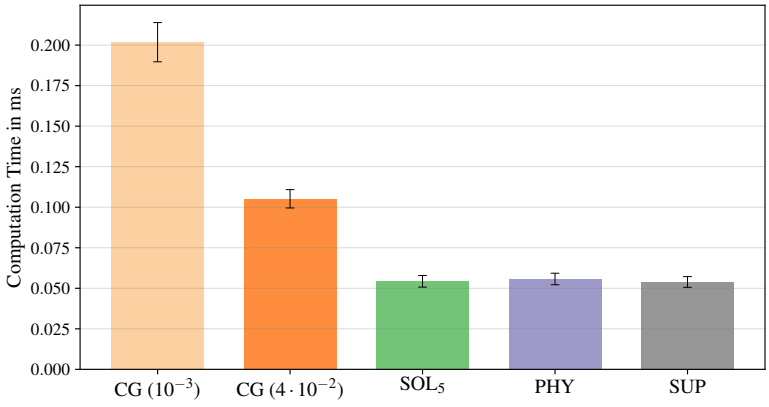


Figure 6.12: Comparison of average computation time per simulation step.

Of course, achieving a low average residual error using a CNN instead of the CG solver is only interesting if the network requires less computation time. We therefore also measured the average calculation time needed for each simulation step. The system we measured on uses an Intel i7-3770 CPU, Nvidia GTX 1070 GPU and 24 GB

of memory. The results are listed in Table 6.2 and visualized in Figure 6.12. Since PHY achieves an average accuracy of roughly $4 \cdot 10^{-2}$, we included a CG solver with that target accuracy for a more direct comparison. The time it takes to evaluate the trained models is practically identical as they use the same architecture. Compared to the CG solver, though, PHY only requires approximately half the computation time to obtain a similar accuracy. For a target accuracy of $4 \cdot 10^{-2}$, PHY is therefore a viable alternative to the CG solver, providing a speed-up.

In summary, the neural network simulations show that PHY and, to a lesser extent, SUP can be used as a replacement for the CG solver, if no very high accuracy must be achieved. Our solver-based approach, however, cannot, even though the initial accuracy of its pressure output is similar to PHY's. The reason for this are small scale, regular artifacts introduced through correction with its pressure guess. The network cannot properly deal with inputs that contain these patterns and thus fails when predicting pressures successively. These artifacts are only produced by SOL_5, not the other models. Since the models only differ in the loss function, it stands to reason that SOL_5 learns to introduce them because of training with the solver. This is corroborated by the fact that the residuals of the CG solver itself also contain similar checkerboard structures (cf. Figure 6.10). Though comparatively small for an accuracy of 10^{-3}, intermediate solutions of the solver, e.g. after 5 iterations, exhibit strong checkerboard artifacts. As SOL_5's loss formulation includes a direct difference to the solver's output after a limited amount of iterations (see Equation 5.3), it learns to imitate these patterns. Along with the observation that SOL_5's outputs present the best initial state for the CG solver (cf. section 6.2), this implies that SOL_5 learns to solve the pressure equation in a way that is more similar to the CG solver than the other approaches.

6.3.2 Hybrid Pressure Solve

Completely replacing the CG solver with a trained model can lead to a speed-up of the simulation. However, this ties the simulation accuracy to the fixed accuracy of the network. It becomes impossible to set an arbitrarily high target accuracy, as is possible when using the CG solver. This is fine for simulations where visual plausibility and computation time are the most important. However, there are several areas of application where high accuracy is essential.

We therefore propose a hybrid approach that does not fully replace the CG solver by a trained pressure predictor but instead uses them in conjunction. The setup for these hybrid simulations is similar to that outlined in subsection 6.3.1. However, instead of directly using the network's pressure guess $\mathcal{C}(\nabla u) = \hat{p}$ to correct the divergent

velocity, we use $\hat{p}$ as an initial guess for the CG solver. As we showed in section 6.2, our solver-based model SOL_5 in particular significantly reduces the iterations the CG solver needs to reach higher accuracies. The hybrid simulation can thus benefit from the neural network's speed-up and still provide the same accuracy guarantee as the CG solver normally does.

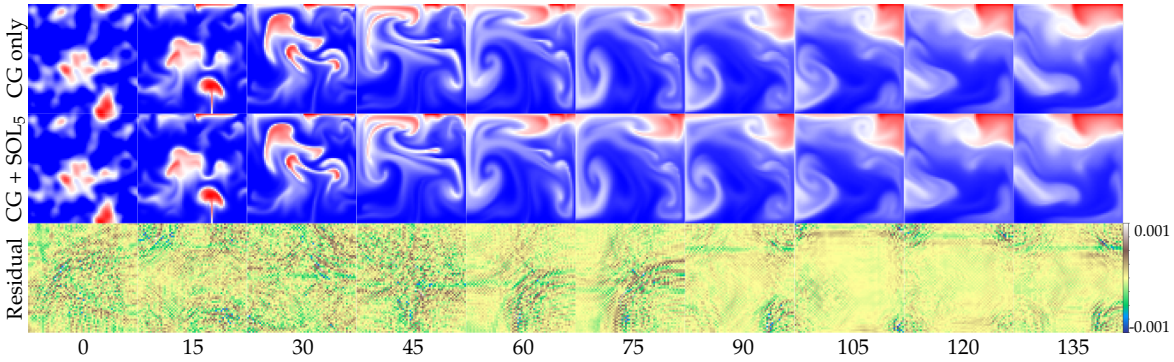


Figure 6.13: Example 10^{-3} accuracy SOL_5 hybrid simulation and its residual divergence.

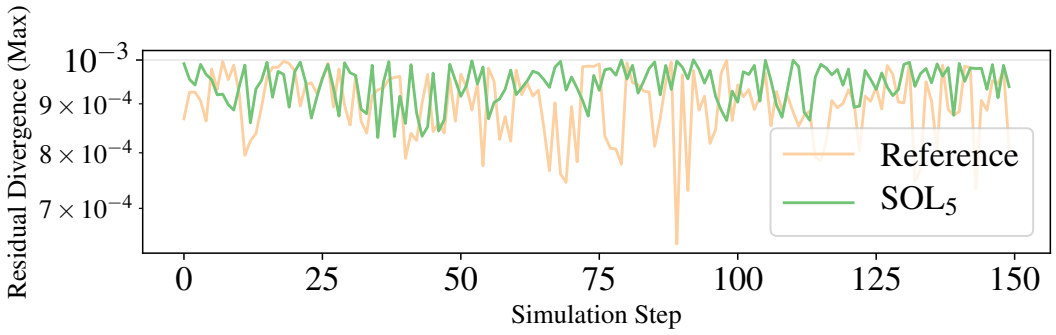


Figure 6.14: Residual divergence over example 10^{-3} accuracy SOL_5 hybrid simulation.

Figure 6.13 shows an example hybrid simulation, wherein the SOL_5 model produces an initial pressure guess for a CG solver with target accuracy 10^{-3}. The maximum absolute residual is additionally shown in Figure 6.14. Indeed, the hybrid approach produces a simulation result that is visually identical and provides the same accuracy guarantee as the solver-only variant.
More interesting is how the hybrid approach compares regarding computational efficiency. We tested this comprehensively for all trained models and target accuracies from 10^{-1} to 10^{-6}, running 100 randomly initialized hybrid simulations for each combination. We used the same machine to measure performance as in section 6.3.

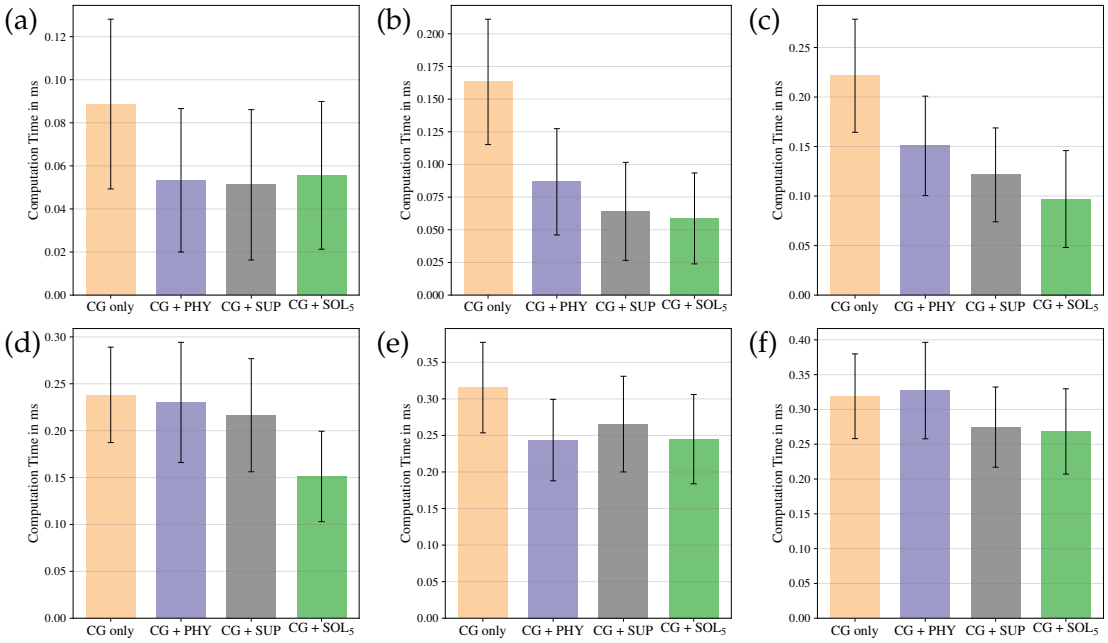


Figure 6.15: Computation time per hybrid simulation step for different models with accuracy (a) 10^{-1} (b) 10^{-2} (c) 10^{-3} (d) 10^{-4} (e) 10^{-5} (f) 10^{-6}.

Pressure Solver	**Computation Time for Target Accuracy**, Mean (Standard Deviation)					
	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
CG only	0.0887 ms (0.0394)	0.1632 ms (0.0480)	0.2215 ms (0.0571)	0.2382 ms (0.0509)	0.3154 ms (0.0619)	0.3189 ms (0.0609)
CG + SUP	0.0512 ms (0.0349)	0.0641 ms (0.0375)	0.1214 ms (0.0474)	0.2165 ms (0.0603)	0.2655ms (0.0654)	0.2746ms (0.0576)
CG + PHY	0.0533 ms (0.0333)	0.0867 ms (0.0407)	0.1507 ms (0.0502)	0.2301 ms (0.0641)	0.2436 ms (0.0557)	0.3271 ms (0.0693)
CG + SOL_5	0.0556 ms (0.0343)	0.0587 ms (0.0348)	0.0970 ms (0.0489)	0.1512 ms (0.0482)	0.2449 ms (0.0611)	0.2684 ms (0.0613)

Table 6.3: Computation time per simulation step using different hybrid pressure solvers (averaged over 100 randomized simulations).

The average results are listed in Table 6.3 and shown in Figure 6.15. Right away it can be seen that in every setup, the hybrid solvers generally perform better than, or in a few cases at least similar to, the solver-only baseline. Particularly for accuracies 10^{-2}

and 10^{-3} there is a significant improvement. For example, SOL_5 provides a speed-up of roughly 64% and 56% respectively here. For higher accuracies, this performance improvement diminishes somewhat. This is due to the trained models' outputs having limited accuracy. As was seen in Figure 6.5, their pressure guess can substitute the solver's iterations up to a certain point, beyond which normal CG iterations have to be performed. Therefore, the speed-up is tied to how many iterations the model can replace compared to how long it takes to evaluate the model itself. On our setup, the models take between 0.054 and 0.056 milliseconds to evaluate (see Table 6.2). There is naturally potential for improvement here. Firstly, it may be possible to achieve similar reductions in CG iterations using a leaner neural network architecture than ours (see section 5.1). Secondly, recent advances in specialized neural network hardware promise to cut down network evaluation time further in the future.

For a low target accuracy such as 10^{-1}, network evaluation time is the only contributing factor, as no additional CG iterations are required (cf. Figure 6.15 (a)). In these cases, the hybrid simulation can generally simply use the network's guess directly, as was done in subsection 6.3.1. However, there is a significant advantage of the hybrid approach compared to a straightforward neural network simulation. For SOL_5 in particular, the non-hybrid simulations suffer from quickly accumulating residuals, due to the network being incapable of dealing with inputs based on its own previous pressure corrections.

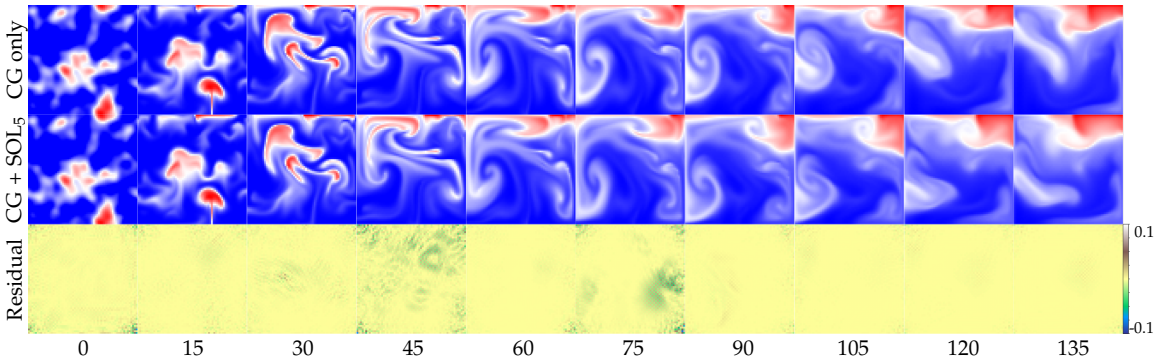


Figure 6.16: Example 10^{-1} accuracy SOL_5 hybrid simulation and its residual divergence.

Figure 6.16 shows an example hybrid simulation using SOL_5 and a target accuracy of 10^{-1}. Unlike Figure 6.13, this hybrid simulation is not visually identical to the CG-only simulation, due to mainly using SOL_5's pressure directly. The same checkerboard pattern as in the neural network simulation can be observed here, too. However, in the hybrid simulation, it does not grow more and more intense as the simulation goes on.

The reason for this can be seen in Figure 6.17. As in the neural network simulation, the residual quickly increases. Due to being a hybrid simulation, however, once the error would increase beyond the guaranteed accuracy of 10^{-1}, the solver performs a single additional iteration, bringing the residual back down below the threshold.

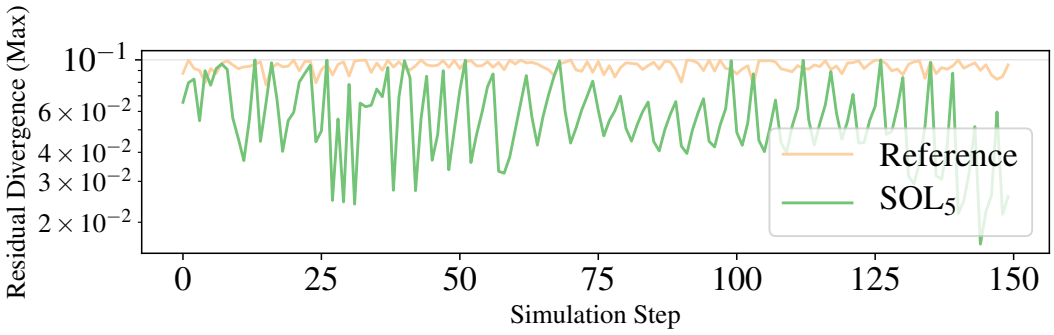


Figure 6.17: Residual divergence over example 10^{-1} accuracy SOL_5 hybrid simulation.

Even in this scenario, the hybrid simulations thus guarantee the chosen target accuracy to be met while providing a non-negligible speed-up. Generally, the hybrid approach therefore shows much more promise than simply using the trained models to replace the solver. And, while PHY proved to be the most suitable model for stand-alone pressure prediction, the results in this section suggest that our solver-based approach is the better choice for hybrid simulations.

6.4 Further Experiments

This section contains additional experiments wherein we further investigate the solver-based approach under different training and simulation conditions than were employed so far. Specifically, we look at training solver-based models with different look-ahead step sizes k (see Equation 5.3) and applying trained models to different simulation domain sizes than they were trained on. Lastly, we also investigate combining the physics-based and solver-based training approaches to obtain the advantages from both.

6.4.1 Look-Ahead Step Size

The solver-based loss function we proposed in subsection 5.2.3 defines the parameter k. It describes the number of CG solver iterations that are performed on top of the net-

work's output, before the two are compared. We therefore refer to k as the look-ahead step size, since a larger k allows the network to observe how its guess affects the solver's output further down the road. Though we focused on $\text{SOL}_{k=5}$ in our comparisons to the other approaches, we also experimented with varying this look-ahead. Figures 6.18 and 6.19 show how solver-based models with different k perform on the test dataset.

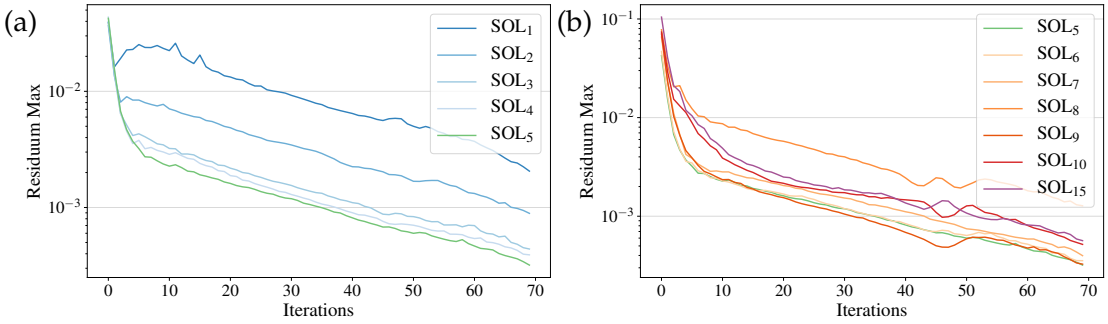


Figure 6.18: CG solver residual, given initial pressure guesses predicted by different SOL_k variants. (a) $k \leq 5$ (b) $k \geq 5$.

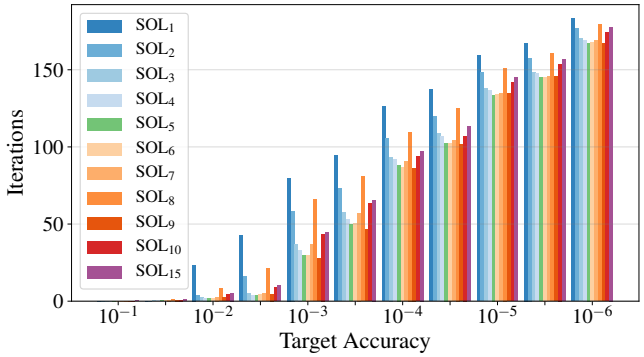


Figure 6.19: CG iterations for target accuracies, given different SOL_k pressures.

For $k \leq 5$, there is a clear tendency to be observed (cf. Figure 6.18 (a)). If k is too small, too few solver iterations are performed and the model's performance deteriorates noticeably. The residual does not see the reduction previously associated with the solver-based variant and consequently, a lot more CG iterations are needed to reach the target accuracies (see Figure 6.19). SOL_k variants with $k \geq 5$ do not show such an obvious trend. It seems that increasing k beyond this point does not further improve the usefulness of the network's output for the CG solver.

Another interesting observation can be made when looking at the development of the residual when these SOL_k variants are used as standalone pressure predictors in a simulation (as in subsection 6.3.1). The average results of this over ten example simulations are depicted in Figure 6.20.

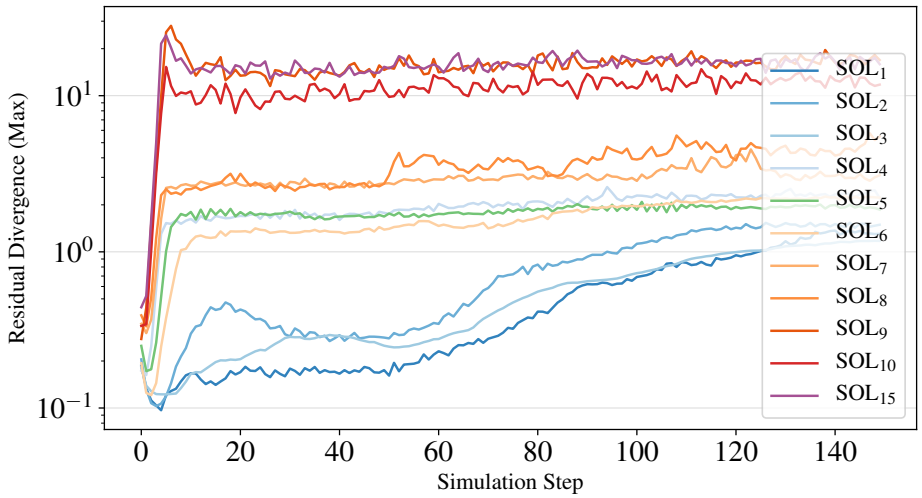


Figure 6.20: Residual throughout randomized simulations using various SOL_k as standalone pressure predictors (averaged over 10 example simulations).

Though there is of course some variance due to the randomness of both the model parameters' initialization and the example simulations, we can see notably different behavior between SOL_k variations with low and high k. Models with a higher look-ahead step size evidently accumulate the residual error from their own pressure corrections much more quickly. This matches our findings in subsection 6.3.1 concerning the checkerboard artifacts in SOL_5's residual and gives further credence to our speculation that training with a direct comparison to the solver's intermediate solution is the cause. For a small look-ahead step size, the solver cannot introduce strong checkerboard patterns on top of the network's output. With higher look-ahead step size, however, the solver's alterations gain more weight and the network learns to imitate them.

6.4.2 Domain Size

To keep data generation, training and simulation times reasonable, we mainly evaluate our approaches on domains of size 64×64. Obviously, though, larger resolutions are desirable and often required. It is therefore interesting to see how our trained models

can be applied to bigger domains than they were originally trained on. As an example, we will focus on applying models trained on 64×64 to a domain of size 256×256. We compare two methods to do so. First, since our network architecture is fully convolutional, it is possible to simply apply it to inputs of any size. This method is straightforward but does not consider that the networks may perform worse on domain sizes they have not seen at training time.
Our second method therefore first downscales the divergence field to 64×64, feeds it into the network and then upsamples the pressure output to 256×256 again. In this way, the network need only infer pressure based on a 64×64 input, as it was originally trained to do.

Neural Network Simulation We previously established PHY to be the most suitable model to run a network-only fluid simulation with (cf. subsection 6.3.1). Therefore, we used it to investigate higher resolution simulations. An example simulation with domain size 256×256 is shown in Figure 6.21.

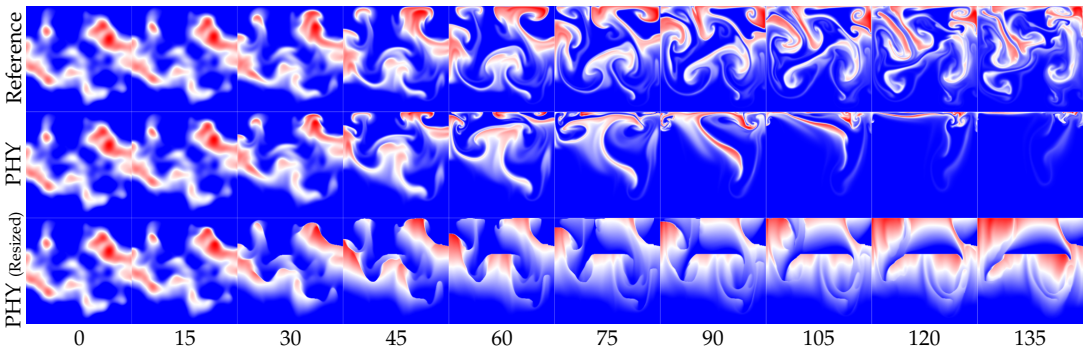


Figure 6.21: Example 10^{-3} accuracy simulation on a domain of size 256×256.

The results are noticeably worse than when the PHY network is used on a 64×64 domain (cf. subsection 6.3.1). Naively applying PHY to the larger domain makes the simulation drastically deviate from the reference. Interestingly, resizing makes the network perform even worse. The pressure (see Figure 6.22) shows that the resizing approach does help the network to initially predict outputs closer to the reference. However, as the fluid becomes more turbulent (and with it the input divergence and reference pressure), the resized approach fails to capture any of the details. This is very likely due to the input divergence's details getting lost in the downscaling process. The naively applied PHY does not have this issue and is visibly better at dealing with more turbulent flow, yet still proves unable to make useful pressure predictions.

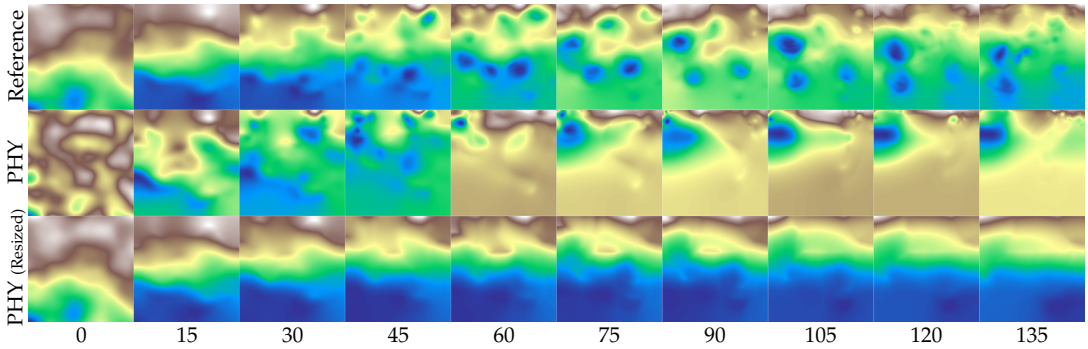


Figure 6.22: Pressure outputs for an example simulation on a domain of size 256×256.

Hybrid Simulation Regarding performance in conjunction with the solver, SOL_5 showed the best results (see subsection 6.3.2). We thus use it to investigate how our models fare on higher dimensional domains in terms of reducing the solver's iterations. As Figure 6.23 shows, the SOL_5 model reduces the iterations the solver needs to reach an accuracy of 10^{-3} here too. Though the reduction seems small, the SOL_5 hybrid approach requires 80 iterations less than the solver would on its own. Absolutely speaking, this is actually a similar reduction in iterations as was seen on the test dataset for 64×64 samples (see Figure 6.5). Of course, it is probable that the reduction would be bigger relatively if the model had been trained on 256×256 data.
The resizing approach again shows worse results than simply applying the fully convolutional model naively. Likely here, too, the downscaling and upsampling lead to a loss of detail in the predicted guess, making it less useful to the solver.

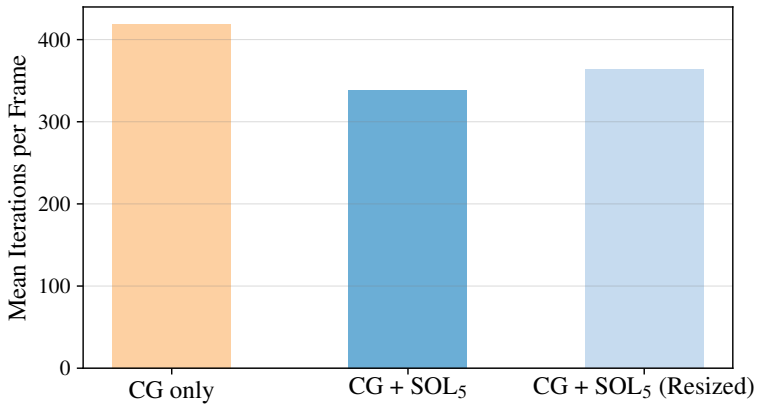


Figure 6.23: Iterations per time step of a 256×256 simulation with target accuracy 10^{-3} (averaged over 10 example simulations).

6.4.3 Combining Solver-Based and Physics-Based Learning

Our experiments in section 6.2 and subsection 6.3.2 suggest that solver-based learning greatly benefits the usefulness of a network's output in conjunction with the solver. On the other hand, we found that our solver-based models are not suitable for standalone use in simulations (see subsection 6.3.1 and subsection 6.4.1). They accumulate the residual from their own pressure corrections much more quickly than e.g. physics-based methods. This appears to result from direct comparison of the network's output to the CG solver's output several iterations further, with more iterations exacerbating the issue.

Therefore, we formulate another loss which forgoes this direct comparison and instead includes the physics-informed error from Equation 5.2 on top of the differentiable solver's output:

$$\mathcal{L}_{\text{SOL}_{\text{PHY}}} = \left\| \nabla u - \Delta \mathcal{S}^5(\mathcal{C}(u)) \right\|^2 \tag{6.1}$$

This combined loss allows the network to observe the solver's behavior given its prediction, but also ultimately aims to minimize the residual divergence instead of the distance to the solver's output. We intentionally fix the look-ahead step size to 5 here, to simplify notation.

We trained an additional model, SOL_{PHY}, using this loss in the same way as our previous models. Evaluating it on the test dataset (see Figure 6.24), we see that SOL_{PHY} generally retains the reduction in solver-iterations and residual that the previous solver-based approach also showed. The slightly worse performance compared to SOL_5 can likely be attributed to the randomized initialization of the network's parameters. Due to reducing the iterations in a similar manner, SOL_{PHY} also provides a speed-up

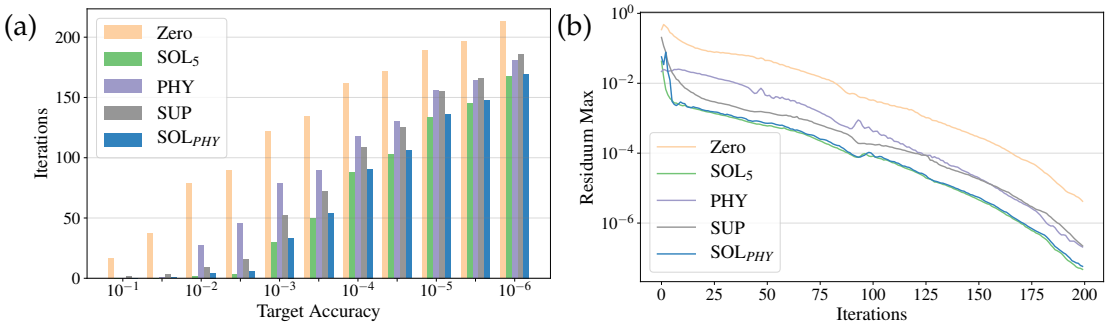


Figure 6.24: SOL_{PHY}, (a) Iterations needed to reach target accuracy (b) Comparison of maximum residual error over iterations.

comparable to SOL_5 when used in a hybrid simulation. For example, it requires an average of 0.0933 ms per time step for a target accuracy of 10^{-3}, whereas the CG solver would require 0.2215 ms by itself (cf. Table 6.3).
These results are very similar to SOL_5, so far. When used to replace the CG solver in a non-hybrid simulation, though, SOL_{PHY} shows a clear improvement over SOL_5. SOL_{PHY} stays visually much closer to the 10^{-3} accuracy reference, only showing small deviations e.g. in frames 60, 75 and 90 (see Figure 6.25). Its pressure (see Figure 6.26) also does not exhibit the same checkerboard pattern as SOL_5 and generally stays much more stable throughout the simulation.

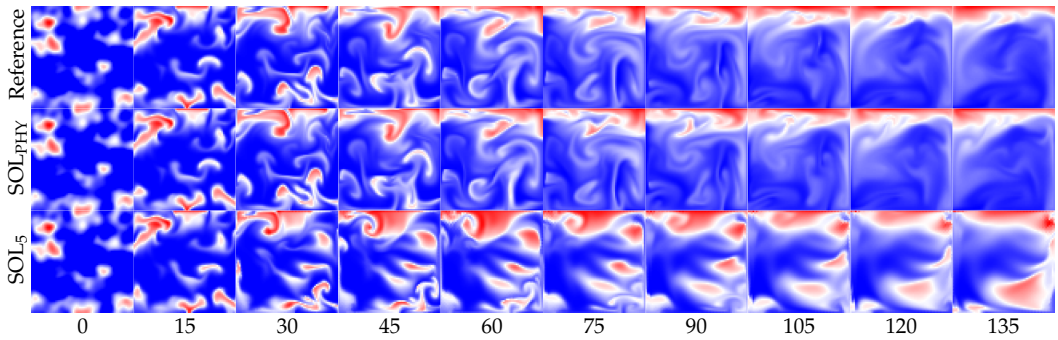


Figure 6.25: Example simulation using SOL_{PHY} as a standalone pressure solver.

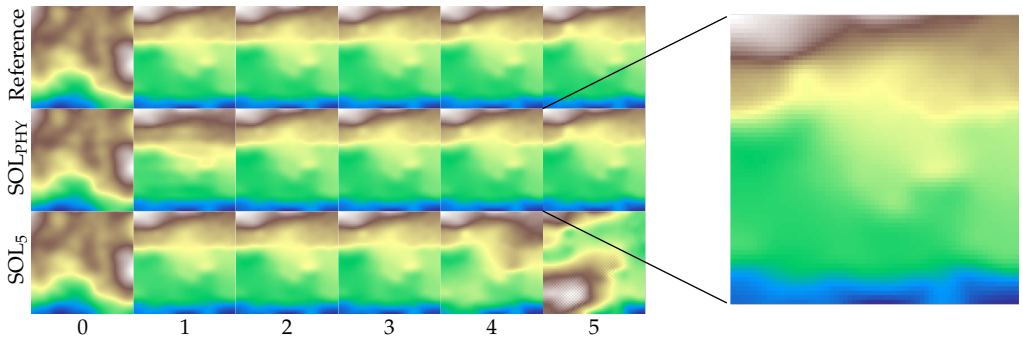


Figure 6.26: Pressure outputs of SOL_{PHY} in the beginning of the simulation.

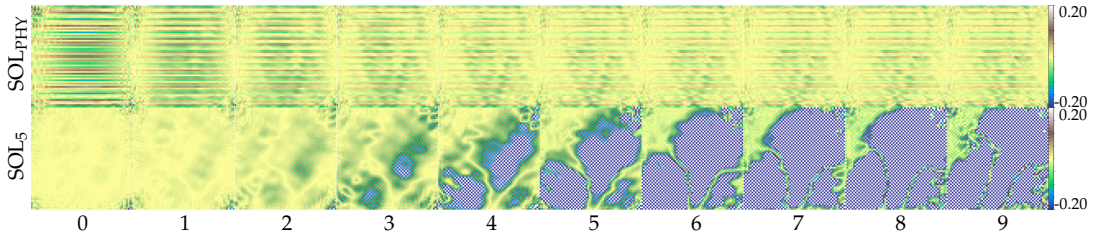


Figure 6.27: Residual divergence in the initial frames of the SOL_{PHY} simulation.

Accordingly, the residual divergence (see Figure 6.27) also does not accumulate as it does for SOL_5. This becomes even more clear when looking at the average residual throughout 100 example simulations, as shown in Figure 6.28.

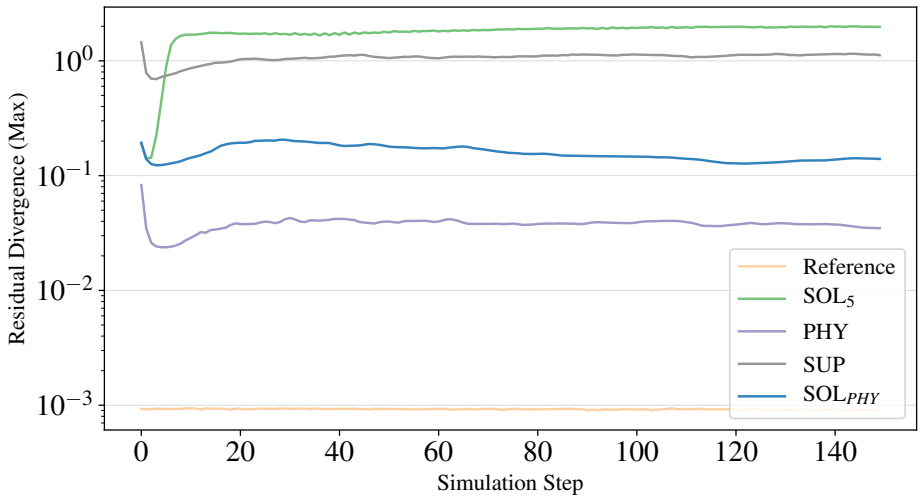


Figure 6.28: Comparison of SOL_{PHY}'s average residual error throughout a simulation to that of the other models.

The SOL_{PHY} model is able to cope with the divergence resulting from its own pressure corrections much better, leading to a residual divergence that only slowly increases and even stagnates, much like the PHY model's. This is even more interesting considering that, although it does not have the same checkerboard artifacts as SOL_5, there is a noticeable striped pattern in SOL_{PHY}'s residual divergence. It appears that using the physics-informed error metric for the final comparison (see Equation 6.1) instead of a direct difference between network output and solver output (see Equation 5.3) makes the trained network more tolerant to erratic patterns in its input.

In summary, SOL_{PHY} combines the advantages of PHY and SOL_5 effectively. Due to observing the solver at training time, it retains the usefulness as an initial guess for the CG solver and thus achieves an almost identical speed-up as SOL_5 in a hybrid simulation. It also gains PHY's in-simulation stability, as its error is ultimately determined by the actual physical residual.

7 Conclusions

In this thesis, we explored different approaches to training a Convolutional Neural Network to accelerate the pressure solving step of Eulerian fluid simulations. We focused on varying the loss formulation of these networks, comparing supervised, physics-based and solver-based methods. Our results show that the physics-based and solver-based variants excel in different areas, while the supervised variant is generally outperformed. The physics-based variant achieves the highest accuracy when used as a standalone pressure predictor. In a simulation, it can effectively be used to substitute a numerical solver, trading some accuracy for computational performance while still producing visually convincing results. However, its output is ill suited for use as an initial state that the solver can further iterate upon. Here, our solver-based approach performs much better. Its pressure predictions significantly reduce the amount of iterations the solver needs to perform to reach its target accuracy. If used in conjunction with the solver in a hybrid simulation, the solver-trained model leads to noticeable real-world performance benefits. For a target simulation accuracy of 10^{-3}, our solver-based hybrid approach more than doubles computation speed. Even so, we also found that our solver-based loss formulation is not optimal. Training to directly minimize the difference to an intermediate output of the solver causes the network to adopt the same checkerboard patterns found in these limited-iteration solutions. If not included in the training dataset, the trained model then becomes unable to handle input divergences containing these artifacts. When used as a standalone pressure predictor, our solver-based model therefore accumulates its residual error much faster, leading to an unstable simulation. We showed that this issue can be alleviated by combining the physics-based and solver-based loss functions, so that the result no longer contains a direct comparison of the network's output to an intermediate solver solution. This variant retains the iteration performance benefits of solver-based training and can also be used as a standalone pressure predictor.

Though these findings were made in the context of fluid simulation, they can be extrapolated to solving partial differential equations (PDEs) in general. Solver-based learning can be used to train specific neural networks for different classes of PDEs. Their output can then be used as an initial guess for a numerical solver to further improve upon. In the same vein as our hybrid simulations, this promises to provide notable performance benefits to solving PDEs in different areas.

8 Future Work

The solver-based approach discussed in this thesis showed promising results for accelerating grid-based fluid simulations and the numerical solution of partial differential equations in general. Nevertheless, there is, of course, ample room for improvement and follow-up research. As we mainly compared the effects of different approaches to the loss function, an obvious next step would be to further optimize the network structure. Reducing the number of trainable parameters while maintaining comparable iteration reduction would cut down on the cost of evaluating the network itself, thereby accelerating hybrid simulations further. For this, an ablation study could be performed on the network architecture we proposed in this work. If less manual trial-and-error is desired, structured network pruning techniques [see Bla+20] could also be employed. In doing so, weights, biases and even entire filters and layers could be identified that do not meaningfully contribute to the trained model's prediction. Removing them would lead to a leaner network that could be evaluated more quickly. A different approach to further speeding up hybrid simulations would be to combine our method with correction-based schemes as discussed in [Um+19; Hsi+19]. A trained model could first infer a guess for the numerical solver to use as a starting point. A second, corrective network could then improve the solver's updates, reducing the overall required iterations further.

Besides improving the computational speed of our hybrid approach to simulations, it would be worth investigating the method in different and more complicated settings. For simplicity, we largely omitted obstacles in the simulations in this work. In- and outflows were likewise left out in our investigations. As these features would greatly increase the amount of interesting phenomena that our hybrid simulations can capture, exploring how they could be integrated into our approach would definitely benefit its practical usefulness. Similarly, applying our method to three dimensional simulations and evaluating its performance could be an interesting follow-up investigation. Lastly, as we focused on a standard Conjugate Gradient solver, the approach should also be evaluated with respect to different numerical solution techniques. Other algorithms such as Jacobi and Gauss-Seidel could be experimented with, as could Preconditioning. Our findings thus provide several avenues for further research, which we feel the tangible performance improvements demonstrated in this work warrant.

Bibliography

[Ada20] N. Adaloglou. "Intuitive Explanation of Skip Connections in Deep Learning." In: *https://theaisummer.com/* (2020).

[Bla+20] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. "What is the state of neural network pruning?" In: *arXiv preprint arXiv:2003.03033* (2020).

[BM07] R. Bridson and M. Müller-Fischer. "Fluid simulation: SIGGRAPH 2007 course notes." In: *ACM SIGGRAPH 2007 courses*. 2007, pp. 1–81.

[GB10] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[Goo16] I. Goodfellow. "NIPS 2016 tutorial: Generative adversarial networks." In: *arXiv preprint arXiv:1701.00160* (2016).

[H+52] M. R. Hestenes, E. Stiefel, et al. "Methods of conjugate gradients for solving linear systems." In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[HKT20] P. Holl, V. Koltun, and N. Thuerey. "Learning to control pdes with differentiable physics." In: *arXiv preprint arXiv:2001.07457* (2020).

[Hsi+19] J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon. "Learning neural PDE solvers with convergence guarantees." In: *arXiv preprint arXiv:1906.01200* (2019).

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[Lad+15] L. Ladick, S. Jeong, B. Solenthaler, M. Pollefeys, and M. Gross. "Data-driven fluid simulations using regression forests." In: *ACM Transactions on Graphics (TOG)* 34.6 (2015), pp. 1–9.

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[Lon+18] Z. Long, Y. Lu, X. Ma, and B. Dong. "Pde-net: Learning pdes from data." In: *International Conference on Machine Learning*. 2018, pp. 3208–3216.

[Nie15] M. A. Nielsen. *Neural networks and deep learning*. Vol. 2018. Determination press San Francisco, CA, 2015.

[ODO16] A. Odena, V. Dumoulin, and C. Olah. "Deconvolution and Checkerboard Artifacts." In: *Distill* (2016). DOI: 10.23915/distill.00003.

[Özb+19] A. G. Özbay, S. Laizet, P. Tzirakis, G. Rizos, and B. Schuller. "Poisson cnn: Convolutional neural networks for the solution of the poisson equation with varying meshes and dirichlet boundary conditions." In: *arXiv preprint arXiv:1910.08613* (2019).

[RFB15] O. Ronneberger, P. Fischer, and T. Brox. "U-net: Convolutional networks for biomedical image segmentation." In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." In: *nature* 323.6088 (1986), pp. 533–536.

[RPK17] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations." In: *arXiv preprint arXiv:1711.10561* (2017).

[RYK18] M. Raissi, A. Yazdani, and G. E. Karniadakis. "Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data." In: *arXiv:1808.04327* (2018).

[Saa03] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[San+20] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia. "Learning to simulate complex physics with graph networks." In: *arXiv preprint arXiv:2002.09405* (2020).

[She+94] J. R. Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

[Spr+14] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. "Striving for simplicity: The all convolutional net." In: *arXiv preprint arXiv:1412.6806* (2014).

[SS18] J. Sirignano and K. Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations." In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364.

[Sta99] J. Stam. "Stable fluids." In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, pp. 121–128.

[Tom+17] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin. "Accelerating Eulerian Fluid Simulation With Convolutional Networks." In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, June 2017, pp. 3424–3433.

[Um+19] K. Um, Y. R. Fei, P. Holl, and N. Thuerey. "Learning Time-Aware Assistance Functions for Numerical Fluid Solvers." In: (2019).

[Um+20] K. Um, P. Holl, R. Brand, N. Thuerey, et al. "Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers." In: *arXiv preprint arXiv:2007.00016* (2020).

[WBT19] S. Wiewel, M. Becher, and N. Thuerey. "Latent space physics: Towards learning the temporal evolution of fluid flow." In: *Computer Graphics Forum*. Vol. 38. 2. Wiley Online Library. 2019, pp. 71–82.

[Xie+18] Y. Xie, E. Franz, M. Chu, and N. Thuerey. "tempogan: A temporally coherent, volumetric gan for super-resolution fluid flow." In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–15.