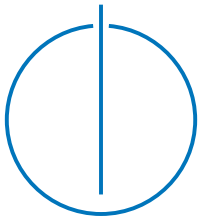
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Learning Spatio-Temporal Density Derivatives in Smoothed Particle Hydrodynamics Using Graph Neural Networks

Jakob Semmler



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Learning Spatio-Temporal Density Derivatives in Smoothed Particle Hydrodynamics Using Graph Neural Networks

Lernen von räumlich-zeitlichen Dichteableitungen in der geglätteten Teilchen-Hydrodynamik unter Verwendung von Graph neuronalen Netzen

Author:	Jakob Semmler
Supervisor:	Prof. Dr. Nils Thuerey
Advisor:	Dr. Rene Winchenbach
Submission Date:	15.10.2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.10.2023 Jakob Semmler

Acknowledgments

I would like to express my gratitude to my advisor, **Dr. Rene Winchenbach**, for his time, and passion. Without exception, he was always there for questions, giving guidance, and discussing ideas.

I thank **Prof. Dr. Nils Thuerey** for helping me connect with Rene to find this exciting topic at the intersection of physics and deep learning. Both of you gave me the opportunity to delve into interesting technologies that will undoubtedly shape the years to come.

A special thanks also goes to those who took the time out of their busy days to proofread the thesis and give me feedback. Thank you very much for this:
Gregor Semmler, **Lukas Semmler**, **Dr. Jürgen Herrler**, and **Dr. Ömer Uludağ**.

Finally, I would like to thank my parents, siblings, and loved ones for always giving me their unconditional support. I would like to dedicate this work to them.

Abstract

This bachelor's thesis improves the standard parameter choices of the Continuous Convolutional Neural Network (ConvNet) architecture applied to the spatial and temporal density derivative problem. We assume that this task represents a class of more complex problems inherent to particle-based fluid simulations, such as those with the Smoothed Particle Hydrodynamics (SPH) method. We additionally consider ConvNet representative of other Graph Neural Networks (GNN). They are a class of neural networks operating on graphs.

This work identifies ConvNet's hyperparameters. They define a network's concrete shape and properties of the learning process. If chosen correctly, they significantly enhance a network's performance. Using multiple random and adaptive hyperparameter searches, we show that an increasing number of parameters and kernel size are the strongest predictors of model capacity and potential performance. Other design choices have little impact, such as activation function, window function, and coordinate mapping. The baseline architecture performs surprisingly well for its relatively small number of parameters. To prove these findings, we find a simple architecture configuration that outperforms the original ConvNet design on the Dynamic Particle Interaction Network (DPI-Net) dataset. This configuration uses a simplified version of the continuous convolution with identity coordinate mapping and no normalization.

We perform a quantitative and qualitative analysis of the refined network design to understand the origins of improvement. We examine error distributions, the behavior of the learned convolution in the SPH context, the visual credibility of the predicted values, and the rollout error obtained by tracking the prediction error over multiple time frames from a contiguous simulation.

This work aims to broaden the understanding of the performance limiting or enabling factors of GNNs for particle-based fluid simulations.

Contents

Chapter 1

Introduction

Smoothed Particle Hydrodynamics (SPH) is a well-researched and flexible method for simulating continuous media, such as liquids or gases. It has increasing practical and industrial applications in various fields, such as the aerospace and automotive industries, cinematography, animation, video games, astrophysics, ballistics, and medicine. For example, SPH can be used for flow simulation, the creation of realistic water effects and simulations of forming galaxies, impacting meteoroids, shock waves, tsunamis, blood flow, and more [LGE08; LR15; Ihm+14; Bor+22; ZL21; Mau+20; Cab+17]. SPH distinguishes itself from other **Computational Fluid Dynamics** methods by, among other things, being purely **Lagrangian**, i.e., particle-based instead of grid-based. Using particles proves advantageous in scenarios that challenge traditional grid-based methods. Typical use cases include the simulation of complex geometrics and large deformations, inhomogeneities, deformable boundaries, and free surface simulations. SPH allows for a simple and robust implementation that can be parallelized with little additional effort. The core idea of the method is to divide a continuous medium into discrete particles and to let these particles interact. Field quantities such as density or pressure are calculated by weighting neighboring particles using a **smoothing** or **kernel function**. Changes in density are related to compression of the medium or fluid flow and are described by the **density derivative** [LL10].

A field that more recently gained enormous traction is the field of **Artificial Intelligence (AI)**. Within it, computers started to learn many skills previously limited to humans. They now understand and produce speech in text and sound, recognize objects and faces, make medical diagnoses, guide vehicles through traffic, create photorealistic images, or work as personal assistants, to name just a few practical applications [GBC16, p. 1; Cao+23]. In particular, the AI subfield of **deep learning** drives this development. It has its name from the **neural networks** it involves. These networks or models typically have many layers and, therefore, are called deep [GBC16, p. 1]. Neural networks are mathematical expressions that abstractly represent neural tissue loosely inspired by their biological counterpart [Kar23]. Deep learning models solve complex problems by learning patterns from data. This approach fundamentally differs from the programmatic approach, which directly describes the logical steps to solving a problem [GBC16, p. 1].

A relatively new and unexplored area is the use of neural networks for particle-based physical simulations. A **Graph Neural Network (GNN)** is one way to combine both of these technologies. This class of neural networks takes a graph and transforms it without changing its connecting edges. A GNN can learn the dynamics from a physical system encoded as a graph structure.

This thesis explores the application of Graph Neural Networks to the SPH method with the representative temporal and spatial density gradient problem. The focus lies on the ConvNet architecture as a representative GNN. The aim is to gain a more comprehensive view of GNNs and their applications to the SPH method. Performance-related design choices in the network setup are identified. Insights about their influence are used to improve the standard parameter choice of the ConvNet architecture presented in the original paper by Ummenhofer et al. [Umm+20]. An exemplary configuration of the architecture is found that outperforms the original architecture. This improved configuration is quantitatively and qualitatively evaluated against the original baseline and an analytical SPH approach. The work concludes with practical recommendations for using ConvNets and possible further research paths.

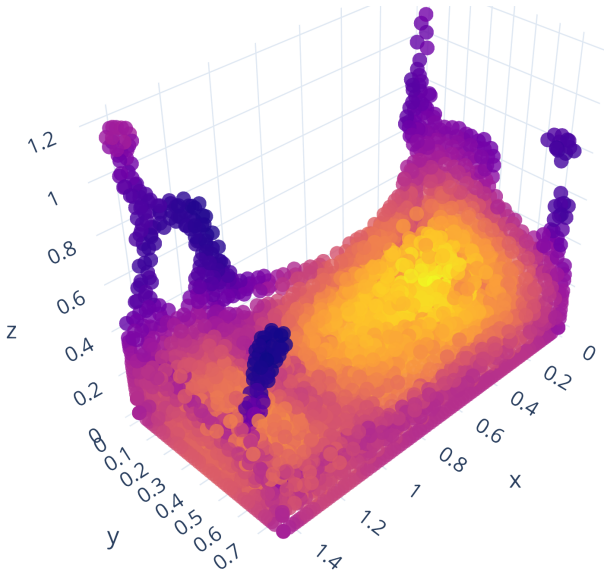


Figure 1.1.: SPH simulated fluid in cuboid.

Chapter 2

Foundations

Flows of gases, liquids, and other continuous media around the simplest objects exhibit a wealth of possible global flow patterns with amazingly complex local features [McL13, p. 6], such as those in Figure 2.1. Section 2.1 presents the Navier-Stokes equations that describe everything essential that can be observed in the viscous flow of continuum media. An exact solution of the Navier-Stokes equations is often not possible. With SPH, we introduce a numerical method in Section 2.2 that allows us to simulate complex patterns and fluid motions based on the physical model. Section 2.3 provides us with the concepts needed to introduce deep learning in the form of Graph Neural Networks into our particle-based simulation method. Section 2.4 focuses on finding suitable parameters for such a network. Section 2.5 introduces the mathematical concept of convolution. Intuitively, it is an operation that allows us to make particles learn to communicate spatially.

2.1 Navier-Stokes Equations

The **Navier-Stokes equations** are a set of partial differential equations that provide a highly accurate and comprehensive physical theory to describe the most relevant phenomena incumbent on the flow of liquids such as water. The theory ensures the conservation of mass, momentum, and energy and relates physical properties such as pressure, density, and viscosity. Fluids are treated as continuous materials with physical properties that continuous functions can represent [McL13, p. 13-15].

We choose a **Lagrangian** view of the partial differential equations as it provides a natural particle-based perspective. It describes fixed fluid parcels as they evolve in time and move along their trajectories. In contrast, the **Eulerian** perspective expresses what happens at a fixed point in space. The Lagrangian or material [Kos+19] derivative D/Dt denotes the rate of change of a quantity like velocity or density associated with a fluid parcel over time. In principle, both views are interchangeable, i.e., a field quantity A in Eulerian coordinates A^E can be expressed in Lagrangian coordinates A^L and vice versa via [Kos+19]

$$\frac{DA^E}{Dt} = \frac{\partial A^E}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} A^E \quad \text{and} \quad \frac{DA^L}{Dt} = \frac{\partial A^L}{\partial t}, \tag{2.1}$$

where $\nabla_{\mathbf{x}}$ is the spatial derivative in the direction of $\mathbf{x}$ and $\delta/\delta t$ the partial derivative with respect to time. In practice, however, this is rarely possible [McL13, p. 16-17].

Figure 2.1.: Complex flow patterns of dye around simple objects, by Firestone and Shane [FS07].

The **continuity equation** describes the conservation of mass in fluids [McL13, p. 34 ff.]. The change in density for each material particle is

$$\frac{D\rho}{Dt} = -\rho(\nabla \cdot \mathbf{v}), \tag{2.2}$$

where $\nabla \cdot \mathbf{v}$ is the divergence of the velocity vector, which describes the expansion rate at a given point in the fluid. A change corresponds to a change in density in the local flow. Incompressible fluids have the additional constraint that

$$\frac{D\rho}{Dt} = 0 \quad \Leftrightarrow \quad \nabla \cdot \mathbf{v} = 0. \tag{2.3}$$

The density remains constant at all material points at all times, and equivalently, no divergence in the velocity field can be observed. This is also only possible if there is no inflow or outflow [Kos+19].

The **momentum equation** ensures the conservation of linear momentum in a fluid system. The equation can be seen as a generalization of Newton's second law of motion to continua. It states that the momentum of a material particle changes at a rate equal to the sum of all internal and external volumetric forces acting on the particle,

$$\frac{D\mathbf{v}}{Dt} = \frac{1}{\rho}(\nabla \cdot \mathbf{T} + \mathbf{F}_{\text{ext}}), \tag{2.4}$$

where $\mathbf{T}$ is a stress tensor describing the forces inside the particle and $\mathbf{F}_{\text{ext}}$ denotes the external forces. For incompressible fluids, the equation is changed to

$$\rho\frac{D\mathbf{v}}{Dt} = -\nabla p + \mu\nabla^2\mathbf{v} + \mathbf{F}_{\text{ext}}, \tag{2.5}$$

where μ is the **dynamic viscosity**. It describe the internal resistance of the fluid to flow [Kos+19; Wei+18]. The expression can be rewritten as

$$\rho \frac{D\mathbf{v}}{Dt} = \mathbf{F}_{\text{Pressure}} + \mathbf{F}_{\text{Viscosity}} + \mathbf{F}_{\text{ext}}, \tag{2.6}$$

where $\mathbf{F}_{\text{Pressure}} = -\nabla p$ is the pressure force and $\mathbf{F}_{\text{Viscosity}} = \mu \nabla^2 \mathbf{v}$ is the viscosity force [Kos+19].

2.2 Smoothed Particle Hydrodynamics

2.2.1 Discretization

In an SPH simulation, we represent the initial state of our problem, that is, the solids and fluids to be simulated as a discrete set of particles. These particles discretize spatial field quantities and spatial differential operators such as gradients, divergence, and curl. The discretization scheme is based on the idea that a continuous compactly supported function can be represented exactly as integrals and that these integrals can be estimated. The exact representation is possible using the **Dirac delta distribution** [Kos+19].

The δ distribution is a general function or distribution, although it is often referred to in the literature as the Dirac delta function. It is only non-zero for an input value of zero and can be characterized as

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1, \tag{2.7}$$

$$\delta(x) = 0 \quad \text{for} \quad x \neq 0. \tag{2.8}$$

A function A with domain Ω can be expressed as an integral using the Dirac delta distribution, where each point of the function is weighted by itself,

$$A(\mathbf{x}) = \int_{\Omega} A(\boldsymbol{\tau})\, \delta(\mathbf{x} - \boldsymbol{\tau})\, d\boldsymbol{\tau}. \tag{2.9}$$

The number of particles needed for such a representation of a function would approach infinity, which is impossible to simulate given our finite computational capabilities. Instead, the resolution of the representation is reduced by using a smoothing or kernel function W as a substitute,

$$A(\mathbf{x}) \approx \int_{\Omega} A(\boldsymbol{\tau})\, W(\mathbf{x} - \boldsymbol{\tau}, h)\, d\boldsymbol{\tau}, \tag{2.10}$$

where h is the smoothing length [LL10].

In most cases, Ω is a volume that is discretized with particles. A particle i with position $\mathbf{x}_i$ is associated with a fixed lumped volume ΔV_i with no fixed shape. With the mass-volume-density relation, the volume can be replaced by m_i/ρ_i. A field variable f can then

be approximated by summing over the values of the N nearest neighboring particles and calculating a weighted average,

$$f(\mathbf{x}) \approx \sum_{j=1}^{N} \frac{m_j}{\rho_j} f\left(\boldsymbol{x}_j\right) W\left(\boldsymbol{x} - \boldsymbol{x}_j, h\right). \tag{2.11}$$

Estimating the derivative follows an analogous procedure,

$$\nabla \cdot f(\boldsymbol{x}) \approx -\sum_{j=1}^{N} \frac{m_j}{\rho_j} f\left(\boldsymbol{x}_j\right) \cdot \nabla W\left(\boldsymbol{x} - \boldsymbol{x}_j, h\right), \tag{2.12}$$

where ∇W is the gradient of the kernel [LL10]. Figure 2.2 shows a schematic representation of the SPH particle summation.

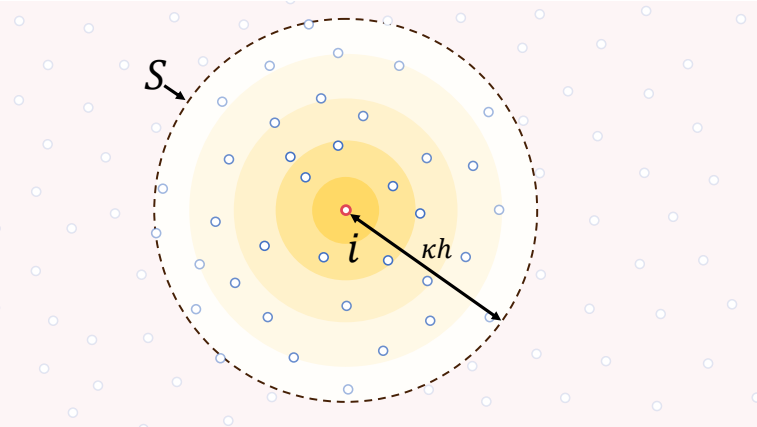


Figure 2.2.: SPH particle summation for a two-dimensional problem domain with surface S. Below the particle i the smoothing function W is shown in yellow. There is a cutoff distance κh beyond which W evaluates to zero. Particles outside the cutoff do not affect particle i.

Using these formulas the density and its temporal and spatial gradient for a particle i can be calculated with [BK15; Ihm+14]

$$\rho_i = \sum_j m_j W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{D1}$$

$$\frac{D\rho_i}{Dt} = \sum_j m_j (\mathbf{v}_i - \mathbf{v}_j) \nabla W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{D2}$$

and

$$\nabla_i \rho_i = \frac{1}{\rho_i} \sum_j m_j (\rho_i - \rho_j) \nabla_i W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{D3}$$

with ∇_i denoting the spatial, Lagrangian derivative with respect to the particle i.

2.2.2 Kernel Properties

The smoothing kernel must satisfy specific properties for optimal performance and stability of the SPH method. Understanding the essential criteria may allow us to evaluate the varying performance of models. M. B. Liu and G. R. Liu [LL10] lists the following properties.

Dirac delta property. For a smoothing length h that approaches 0, the function should behave like the Dirac delta distribution, i.e.,

$$\lim_{h\to 0} W\left(\mathbf{x}_i - \mathbf{x}_j, h\right) = \delta\left(\mathbf{x}_i - \mathbf{x}_j\right). \tag{2.13}$$

If the number of particles approximating the field approaches infinity, the approximation quality should approach the exact value [GM77].

Unit property. The kernel function must be normalized,

$$\int_\Omega W\left(\mathbf{x} - \boldsymbol{\tau}, h\right) d\boldsymbol{\tau} = 1. \tag{2.14}$$

This property ensures, for example, that varying the number of particles with the same values does not change the value of the approximation.

Compact Support Property. The support of W must be bounded,

$$W\left(\mathbf{x}_i - \mathbf{x}_j, h\right) = 0, \quad \text{for } \left|\mathbf{x}_i - \mathbf{x}_j\right| > \kappa h, \quad \kappa \in \mathbb{R}^+. \tag{2.15}$$

There is a distance after which the kernel returns zero. This way, the summation of neighbors can be performed as a local operation instead of a global operation.

Positivity property. Physically, negative weighting of particles and their influence, e.g., negative density or energy, is not possible,

$$W(\boldsymbol{x}, h) \geq 0. \tag{2.16}$$

Decay property. The further apart two particles are, the less effect one particle has on the other, i.e.,

$$W(\boldsymbol{x}, h) \leq W(\boldsymbol{y}, h), \quad \text{for} \quad |\boldsymbol{x}| \geq |\boldsymbol{y}|. \tag{2.17}$$

Smoothing property. A function is smooth if it has no abrupt jumps, corners, or discontinuities. Alternatively, the function W is differentiable, and its derivative ∇W is continuous.

Symmetry property. Any two particles at the same distance will exert the same effect on each other, i.e.,

$$W(\boldsymbol{x}, h) = W(\boldsymbol{y}, h), \quad \text{for} \quad |\boldsymbol{x}| = |\boldsymbol{y}|. \tag{2.18}$$

This property causes visualizations of the kernel to be circularly symmetric.

2.2.3 Kernel

Gaussian kernel. The original paper of Gingold and Monaghan [GM77] introduces the normal distribution as a natural, physically-aligned choice for a kernel to simulate non-spherical stars. The kernel is radially symmetric, smooth, monotonic, positive, approaches the δ distribution for low h, and has well-defined, closed-form expressable derivatives. It often is regarded as the best choice for a kernel in terms of accuracy. In theory, the kernel does not fulfill the compact support property. In practice, it approaches zero numerically for large values and can often be truncated. It is still computationally expensive, as we may need a large support domain with more particles for the approximations [LLL03; LL10].

Cubic B-spline kernel. The most commonly used smoothing function is the cubic B-spline kernel.

$$W(\mathbf{x}, h) = \sigma_d \times \begin{cases} \frac{2}{3} - |\frac{\mathbf{x}}{h}|^2 + \frac{1}{2}|\frac{\mathbf{x}}{h}|^3, & 0 \leq |\frac{\mathbf{x}}{h}| < 1, \\ \frac{1}{6}(2 - |\frac{\mathbf{x}}{h}|)^3, & 1 \leq |\frac{\mathbf{x}}{h}| < 2, \\ 0, & |\frac{\mathbf{x}}{h}| \geq 2. \end{cases} \tag{K1}$$

with the dimensional factors

$$\sigma_1 = \frac{1}{h}, \quad \sigma_2 = \frac{15}{7\pi h^2} \quad \text{and} \quad \sigma_3 = \frac{3}{2\pi h^3}. \tag{2.19}$$

This kernel is a polynomial approximation of the Gaussian kernel with compact support. The kernel is well-established for its good qualities, but is not perfect. Its second derivative is a piecewise linear function. Therefore, its stability properties are less pleasant compared to other smoother kernels [LL10]. The function also could be made even more accurate with minor changes [LLL03]. However, this inaccuracy is not a problem for us, as we are primarily interested in constructing models to replicate existing physical dynamics, whether they behave naturally or not.

2.2.4 Divergence-Free SPH

Divergence-Free SPH (DFSPH) is a state-of-the-art variant of the classical method for realistic, physically based simulation of incompressible or nearly incompressible fluids. The method adds additional computational steps to correct for density errors present in the standard method. This method includes the calculation of the temporal density gradient from Equation D2. The correction guarantees a nearly divergence-free velocity field to enforce volume compression below 0.01 %. The approach provides highly accurate

Listing 2.1: Pseudocode of the DFSPH algorithm, based on Bender et al. [BK15].

```
function PerformSimulation:
  Initialize particles with positions, velocities, etc.
  for each particle: Compute density using smoothing kernel

  while simulation_time < max_simulation_time:
    for each particle:
        Compute non_pressure_forces such as
            gravity, surface tension and viscosity

    Compute time_step such that simulation is stable

    for each particle:
        Compute predicted_velocity using
            current_velocity and non_pressure_forces

    Correct density error and compute predicted_velocity

    for each particle: Compute position using predicted_velocity
    for each particle: Compute density using smoothing kernel

    Correct divergence error and compute predicted_velocity

    for each particle: compute current_velocity

    simulation_time = simulation_time + time_step
```

results at the cost of computational complexity while being comparatively faster than other SPH methods, even for simulations with millions of particles. DFSPH, in its original form, uses the cubic spline kernel from Equation K1. It forms the basis of the data used to train the networks. A simplified version of the algorithm is shown in Listing 2.1 [BK15].

Calculating densities, pressures and forces to update locations and velocities of particles are essential steps for SPH based simulations. Additional calculations improve stability and quality of the simulation. Often, even small errors such as initial states with unreasonable particle distances or a mix-up of calculation steps will cause a feedback loop where forces increase explosively and the simulation *blows up*.

The pressure p_i of a fluid particle i is calculated using the **Tait Equation** [DM88]. It applies pressure in the direction of the **rest density** ρ^0, [BK15]

$$p_i = \frac{\kappa \rho_0}{\gamma} \left(\left(\frac{\rho_i}{\rho_0} \right)^{\gamma} - 1 \right), \tag{2.20}$$

where κ and γ are stiffness constants. Bender and Koschier [BK15] sets γ to 1, justifying it as a standard choice in computer graphics, and arrives at the **Ideal Gas Equation**, [BK15]

$$p_i = \kappa \left(\rho_i - \rho_0\right). \tag{2.21}$$

Other precise calculation steps are beyond our scope.

2.3 Incorporating Deep Learning

This section presents the notion of a learnable simulator, as defined by Sanchez-Gonzalez et al. [San+20]. This concept merges the traditional idea of simulators based on SPH with function approximators like GNNs, allowing us to employ deep learning methods on the SPH theory. Basic deep learning concepts, such as backpropagation and mini-batch gradient descent, are assumed to be familiar to the reader. Various introductions to these topics are available, such as Goodfellow et al. [GBC16].

Graphs are defined. They provide a powerful and versatile framework for representing physical and other complex systems [San+21]. GNNs that operate on these graphs are also introduced.

2.3.1 Learnable Simulator

A **simulator** can be represented as a function $s : \mathcal{X} \to \mathcal{X}$ that applies physical dynamics over a single time step. The simulator takes a simulation state $X^t \in \mathcal{X}$ as input and uses it to predict the next state, $s(X^{t_k}) = X^{t_{k+1}}$. This prediction ideally closely matches the actual state obtained in the real world by applying the modeled physical laws. We define a **rollout** over κ steps as the iterative computation of the states $\mathbf{X}^{t_{0:\kappa}} = (X^{t_0}, \ldots, X^{t_\kappa})$ with $X^{t_{k+1}} = s\left(X^{t_k}\right)$ for each time step [San+20].

This concept is extended to a **learnable simulator** s_θ by adding a parametrized function approximator $d_\theta : \mathcal{X} \to \mathcal{Y}$. The output of this approximator can be optimized for some training objective by adjusting θ. The $Y \in \mathcal{Y}$ is additional, not explicitly specified, information about the physical dynamics. The update mechanism uses this information in its update procedure to inform its next prediction, $X^{t_{k+1}} = \text{Update}(X^t, d_\theta)$ [San+20].

For us, the function approximator is a GNN outputting density gradient information. As we have seen with DFSPH, this could be used in the update procedure to correct the density error and avoid unintended volume compression. The SPH simulator operates based on the physical dynamics provided by the Navier-Stokes equations. Our implementation will cover a partial learnable simulator as we only implement the initial part of the update procedure. The focus lies on how well the network replicates the density derivative data.

2.3.2 Graphs

A **graph** $G = (\mathbf{V}, \mathbf{E})$ is a mathematical object consisting out of a set of **vertices** or **nodes** $\mathbf{V} = \{v_1, \ldots, v_n\}$ and **edges** $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$. We use the notation $e_{ij} = (v_i, v_j) \in \mathbf{E}$, which represents an edge pointing from v_i to v_j. We call a graph **undirected** if for all $e_{ij} \in \mathbf{E}$ there is $e_{ji} \in \mathbf{E}$. The **neighborhood** of a node v is denoted by $\mathcal{N}(v) = \{u \in V \mid (v, u) \in E\}$. In scenarios where nodes represent particles with positions, $\mathcal{N}(\boldsymbol{x}, R)$ is the set of all particles within a radius of R around $\boldsymbol{x}$ [Wu+21].

We create a graph to represent the particles in our SPH simulation. In this graph, each particle represents a fluid parcel and is a node. We connect each particle to its neighboring particles within a radius of r by an edge that describes the Euclidean distance between them.

2.3.3 Graph Neural Networks

Graph Neural Networks are a promising new class of neural networks with many unanswered research questions. Physics simulations, antibacterial discovery, recommender systems, and medicine are some areas where they have achieved remarkable success [Tha+22; Sto+20; Gao+23; Zha+21]. Following a graph-in-graph-out architecture, GNNs transform graphs with their rich information while preserving the connecting edges. This class of networks learns to capture the hidden patterns present in graphs. This approach differs from typical deep learning, which does the same for Euclidean data. The GNN maps its input graph into latent space and allows individual nodes and/or edges to communicate with each other over multiple steps. Latent space is an abstract and multidimensional space in which features with similar values are grouped together to encode meaningful relationships within the data [Ash19]. GNNs can predict individual nodes, edges, or the entire graph [San+21].

In the case of particle-based simulations, this might involve predicting particle properties such as pressure, the relationship between particles such as pairwise distance, or something more general such as the temperature of the simulation based on the kinetic energy of individual elements.

Message passing is a central concept for GNNs. It describes how information is passed between nodes and edges. First, nodes and edges are mapped to feature vectors describing their information. A **message** is constructed for each node v by aggregating the information of its neighborhood $\mathcal{N}(v)$ and connecting edges. The surrounding embedding vectors are typically concatenated into a matrix. This matrix is aggregated into a single vector, the message, by a **pooling** operation β. It could be a summation, weighted average, or other operation over the individual embeddings. Figure 2.3 illustrates the process. A message passing step is completed when the collected information is used to predict new feature vectors for each node, edge, and/or the graph itself. In a simple GNN, this could be done by transforming each message using a single parametrizable function approximator. A block performing the above steps defines a single layer of a GNN. The number of message passing steps gives the depth of the network. For example, a straightforward GNN architecture with message passing would be one where each node, edge, and general graph feature U is mapped by a parametrizable function approximator as shown in Figure 2.4. Figure 2.5 displays the entire graph transformation [San+21].

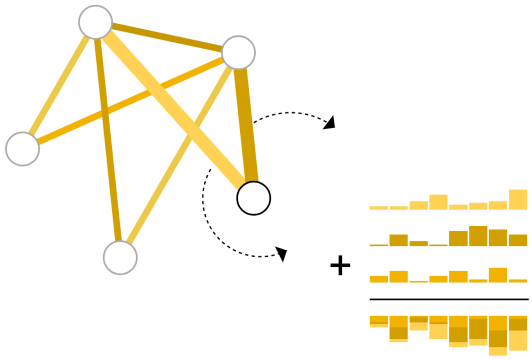


Figure 2.3.: A symbolic representation of the message formation, from Sanchez-Lengeling et al. [San+21].

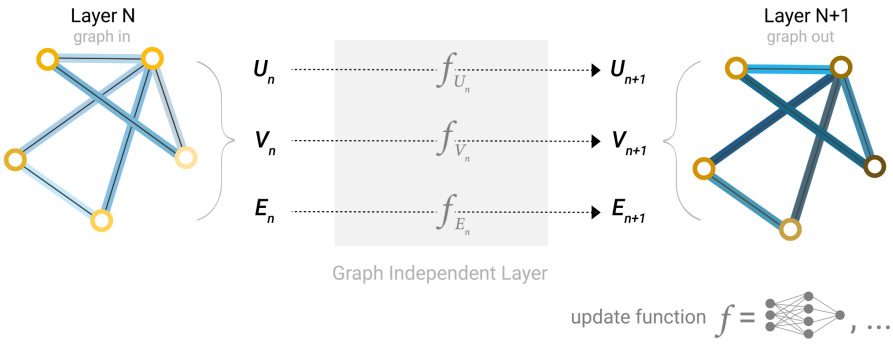


Figure 2.4.: A prediction step, from Sanchez-Lengeling et al. [San+21]. The step completes when nodes **V**, edge values **E**, and the global context **U** of the graph are updated. For ease of explanation, one neural network transforms the three parts separately. In practice, more sophisticated mechanisms are used [San+21].

Interestingly, Sanchez-Gonzalez et al. [San+20] finds that this concept can be directly applied to particle-based simulations and the SPH method. Interactions between particles

(nodes) are computed using their pairwise relations (edges). A method like SPH can be understood as a framework that passes messages between nodes in each simulation step, e.g., by evaluating pressure using the density kernel [San+20].

Whether one or more message passing steps are ideal is still an open research question. After observing the results of Q. Li et al. [LHW18], Wu et al. [Wu+21] notes that increasing the depth of message passing steps can significantly decrease performance. They point out that message passing pushes the representation of adjacent nodes closer together, converging to a single point with an infinite number of network layers. Ummenhofer et al. [Umm+20] uses four message passing steps and achieves good results, while Sanchez-Gonzalez et al. [San+20] observed even better performance for ten or more steps.

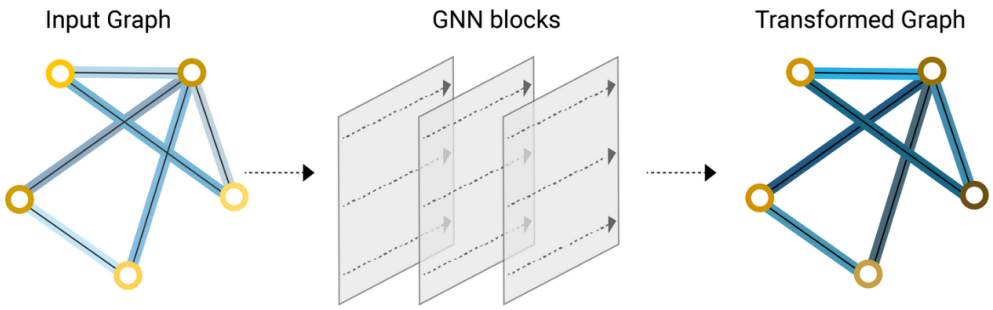


Figure 2.5.: The complete graph transformation, from Sanchez-Lengeling et al. [San+21]. If the network is used for classification, a linear layer can be added with the graph features as input.

2.3.4 Architectures for Particle-Based Simulators

Graph Network (GN) and Graph Network-based Simulator (GNS). The GN architecture is utilized by Sanchez-Gonzalez et al. [San+20] to build a simulator that delivers convincing, state-of-the-art results even for rollouts over thousands of timesteps. GNs and their variants are GNNs with a strong relational inductive bias, sometimes called interaction networks. They can learn to simulate rigid-body, mass-spring, and other systems [Bat+18]. The GNS learns the entire SPH update mechanism and predicts each particle's next position and velocity. Particles track their last $C = 5$ velocities, position, and material features. Velocities are corrupted by adding cumulative random walk noise during training to simulate the buildup of errors. This likely allows the network to learn a built-in error correction mechanism that allows for these long and stable rollouts [San+20].

Their end-to-end learning approach follows an encode-process-decode scheme. A graph with connectivity radius R is generated such that for each particle i, the number of

neighbors $|\mathcal{N}(\boldsymbol{x}_i, R)|$ is about 10 to 20. An encoder maps particles onto nodes and edges using learned features. The GNS performs, for example, $M = 10$ message passing steps and processes the graph in latent space. Using an Euler integrator, a learnable decoder extracts physical dynamics information to update positions $\mathbf{x}$ and velocities $\mathbf{v}$ [San+20].

Continuous Convolutional Neural Network (ConvNet). Ummenhofer et al. [Umm+20] presents a GNN architecture for the simulation of Lagrangian fluids, which gives good results, especially for the simulation of water and similar fluids. This convolutional network architecture uses continuous convolutional layers introduced in Section 2.5. ConvNet accumulates more errors over extended rollouts on the same dataset than GNS but is substantially more lightweight [San+20] and, therefore, faster to train. Models instantiated using this architecture default to four message passing steps. In their implementation, the ConvNet is used to correct the position of individual particles after performing an SPH update routine [Umm+20]. It is possible to simulate ConvNets using GNs [San+20].

We focus on this architecture. It gives great results and is fast and easy to train. This factor is vital given the computational and time constraints of this work.

2.4 Understanding Hyperparameter Optimization

Hyperparameters often have a crucial impact on the performance of the network, defining the speed, behavior, complexity, and other aspects of the learning process and the final model. They are configuration settings set prior to training and used to configure machine learning algorithms and the structure and design of the neural network itself. Unfortunately, selecting a good set of hyperparameters is a challenging problem. Popular optimization methods are usually inapplicable because they often require gradients or many evaluations to converge to a well-performing solution. Optimizing a network by tuning hyperparameters individualy is generally impossible, as their relationship to model performance tends to be non-linear and non-convex [YZ20]. The manual selection of well-performing hyperparameter configurations is, therefore, often an irreproducible, time-consuming, and error-prone process of trial and error. A different and more practical approach to the problem is the use of automatic **Hyperparameter Optimization (HPO)** methods [Bis+21].

We use HPO algorithms to refine the ConvNet architecture with its hyperparameter configuration given by Ummenhofer et al. [Umm+20]. We can identify and gain insight into which hyperparameters relate to network performance by evaluating many design decisions.

2.4.1 Formalization

The performance of a predictive model is best evaluated on unseen test data due to overfitting. Performance can be measured by the **generalization error** GE, which describes how close the model's predictions are to their targets on unseen data. While a precise definition of the generalization error, including concepts such as data splitting and random resampling, is possible, it is also quite tedious. An estimate is sufficient for our purposes. We define the **estimated generalization error** $\widehat{\mathrm{GE}}$ for a given model as the pointwise loss value on the test dataset, where we pretend the values are constant by averaging out the randomness over multiple resamples. Let $\boldsymbol{\lambda} \in \tilde{\Lambda}$ denote a hyperparameter configuration. $\tilde{\Lambda} \subset \Lambda$ is called the search space and contains all possible configurations of hyperparameters and their respective ranges,

$$\tilde{\Lambda} = \tilde{\Lambda}_1 \times \tilde{\Lambda}_2 \times \cdots \times \tilde{\Lambda}_l, \tag{2.22}$$

where $\tilde{\Lambda}_i$ is a bounded subset of the domain of the i-th hyperparameter $\boldsymbol{\lambda}_i$. Domains can be either continuous, discrete, or categorical. It is also possible for hyperparameters and their domains to be interdependent. The general HPO problem can be defined as

$$\boldsymbol{\lambda}^* \in \underset{\boldsymbol{\lambda} \in \tilde{\Lambda}}{\arg\min}\, \widehat{\mathrm{GE}}(\boldsymbol{\lambda}), \tag{2.23}$$

with $\boldsymbol{\lambda}^*$ being a theoretical optimum. The HPO problem corresponds to an expensive black-box optimization problem, where each function evaluation can take a significant amount of time. The black box refers to a scenario where a function takes an input to produce an output without revealing any of its inner workings. For $c(\boldsymbol{\lambda})$, there is, in most cases, no closed-form mathematical representation and consequently no analytic gradient information [Bis+21].

2.4.2 Search Algorithms

Grid Search is a simple search algorithm that can find the optimal hyperparameter configuration, given sufficient computational budget [Ndi+19]. This optimization procedure discretizes the search space for each numerical or integer hyperparameter into equally spaced values. For categorical parameters, it tests all parameters or subsets. The number of values per hyperparameter defines the **resolution** of the grid. The implementation of Grid Search is straightforward, and its parallelization is trivial compared to other methods [BB12]. However, it has a significant disadvantage in that it suffers from the curse of dimensionality. The consumption of computational resources increases exponentially with the number of tuned hyperparameters. Therefore, Grid Search can often only be used when training times are short, only a few hyperparameters or small subsets of the domains

of each hyperparameter are explored. Improving the quality of a found hyperparameter configuration by increasing the grid resolution arbitrarily without recreating the entire grid is generally impossible [YZ20; Bis+21].

Random Search is similar to Grid Search in its simplicity, but in practice, it finds models that are as good or better in a fraction of the time. In this algorithm, the hyperparameters are selected independently from either uniformly or logarithmically uniform distributions for numeric, integer, and, in the former case, categorical parameters [Bis+21]. Bergstra and Bengio [BB12] discovered that for most HPO problems, only a narrow subset of hyperparameters determines the performance of a model on the dataset. As shown in Figure 2.6, this allows Random Search to exploit the relevant subset more efficiently than Grid Search for the same computational budget. Random Search serves as a baseline for more sophisticated HPO optimization algorithms due to its superior performance despite its simplicity compared to the grid-based alternative. Furthermore, the quality of an optimum found using this approach can also be improved by simply evaluating more random hyperparameter configurations, i.e., by exploring more random points in the configuration space [YZ20; Bis+21].

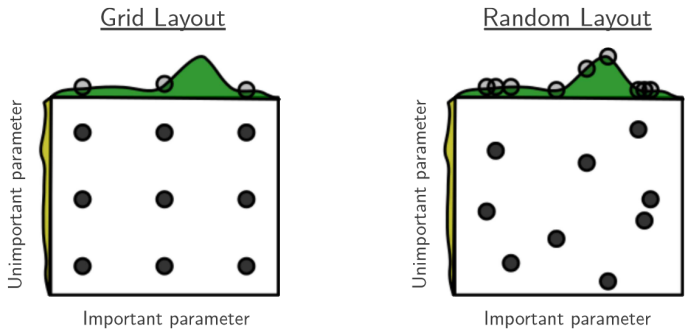


Figure 2.6.: A comparison of Grid and Random Search, from Bergstra and Bengio [BB12]. In the illustrated example, a function $f(x, y) = g(x) + h(y)$ is evaluated, where only $g(x)$ (above) strongly affects the performance of the model. Even though both use the same number of evaluations, Random Search effectively gains more information about g [BB12].

Bayesian Optimization is a more sophisticated algorithm best used for optimizing expensive objective functions, in the case of HPO, expensive models to train. This method promises to find a favorable hyperparameter configuration with fewer trials. The algorithm constructs a probability surrogate model for which it makes guesses and refines it to match the actual problem function more closely. The process consists of the following steps. First, build a prior distribution for the surrogate model. Then, acquire a set of hyperparameters most likely to perform well on that model. Compute the acquisition function with the

current model and apply the hyperparameter configuration to the objective function. Finally, update the surrogate model with the data from the new evaluation. The algorithm balances step by step between exploration and exploitation of its target function. Each time, it updates its knowledge to inform its next step [YZ20; Bis+21].

2.4.3 Scheduler

The **Successive Halving Algorithm (SHA)** optimizes resource usage during a hyperparameter search by iteratively reducing the number of trained models by a specific factor, often half. It can be used as strategy for the scheduler. First, divide all candidate configurations randomly into n groups, where $n = 1$ is recommended. Evaluate each candidate in each group for a short duration, such as one or two training epochs. Stop all but the $1/\eta$ runs with the highest performance, where η is the reduction factor. Repeat the above steps with the remaining $1/\eta$ configurations and η times the training time for each candidate. The **Asynchronous Successive Halving Algorithm (ASHA)** is a more sophisticated variant of the method that allows for parallelization. In practice, ASHA is used to scale training up to hundreds of thousands of nodes, which, unfortunately, is beyond the computational resources available for this thesis. It is also possible to train models in parallel on a single machine using an ASHA scheduler [Li18].

SHA and ASHA multiply the number of configurations that can be evaluated within a given time and computational budget. A drawback of the algorithm is that promising configurations may be eliminated prematurely, especially in the beginning. This unintended elimination happens when the reduction factor is too aggressive, and the performance metrics show too much variance. To counteract this, a grace period for which each model configuration is minimally trained can be configured [Li18].

2.5 Convolution Approaches

Convolution is a mathematical concept that, intuitively, can be used to propagate information spatially [San22]. We explore how it has been adapted for traditional deep learning and for Graph Neural Networks.

2.5.1 Convolution

The **convolution** is best explained intuitively as the blending of a function f and a function g into a new function. This blending is achieved by first mirroring g on the y-axis. Then, the value for any point t of the new function is calculated by shifting the mirrored version of g to t and then using it as a weight for f. Alternatively, it can be understood as an integral expressing the amount of overlap between f and g. Figure 2.7 illustrates this. In its mathematical formulation, this is expressed as an integral. Let $(f * g)$ be the convolution of f and g, then we define

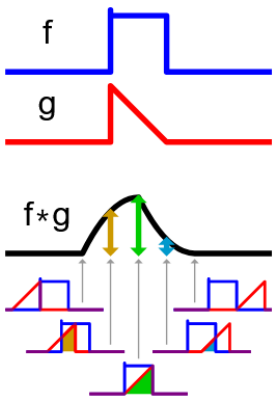


Figure 2.7.: An exemplary convolution of f and g, by Cmglee [Cmg16].

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.24}$$

to be the convolution of both with argument t [Wei].

It becomes apparent that the concept of blending serves as the basis for the previously introduced SPH discretization and smoothing method. In the discretization scheme, field quantities A are convolved with the Dirac delta distribution δ or with the smoothing kernel W in the Equations 2.9, 2.10. We can write it as

$$A(\mathbf{x}) = (A * \delta)(\mathbf{x}) \approx (A * W)(\mathbf{x}). \tag{2.25}$$

2.5.2 Discrete Convolution

One way to translate this concept to functions defined on discrete domains is to replace the integral with a summation operation. This sum is called the **discrete convolution**. It is an operation implemented by the common convolutional layer used for mostly two or three-dimensional data. The layer is a fundamental building block of the **Convolutional Neural Network (CNN)** architecture. CNNs are one of the standard types of networks in deep learning and have demonstrated tremendous success in computer vision and other areas [Li+22]. Let $[f * g]$ denote the discrete convolutions of f and g, then

$$[f * g](\boldsymbol{t}) := \sum_{\boldsymbol{\tau} \in Dom(g)} f(\boldsymbol{\tau})g(\boldsymbol{t}-\boldsymbol{\tau})d\boldsymbol{\tau}, \tag{2.26}$$

The domain of the functions is the set of integer vectors $\mathbb{Z}^D$ of length D with the restriction that the support of g is finite, i.e. $\mathrm{Dom}(g) = \{-M, -M+1, \ldots, M-1, M\}^D$ [Wan+18; San22; GBC16].

The function f is called **feature**, and g is called **kernel**, though g should not be confused with the SPH smoothing kernel. Similar to the continuous case, the kernel's learned or hand-crafted values are shifted over the input data. The kernel can extract certain features or patterns depending on its values. The operation itself is **translation invariant**, i.e., patterns are extracted regardless of their position in the data as the kernel shifts over the entire input. The maximum distance over which information is aggregated between neighbors is limited by the finite size of the kernel. A prototypical usecase for the discrete convolution would be finding edges or corners in an image or video [Wan+18; San22; GBC16].

2.5.3 Parametric Continuous Convolution

A kernel with discrete and finite support is not optimal for various use cases. Many real-world concepts, particularly those captured in images or governed by physical laws, exhibit continuity. Valuable information is lost when these concepts are reduced to a discrete representation, such as an image of pixels. The discrete convolution can only use this lossy form and is therefore limited in its potential. The kernel function g should be defined on continuous values to learn these concepts. Figure 2.8 shows that the resulting operation generalizes the discrete convolution from above [Wan+18].

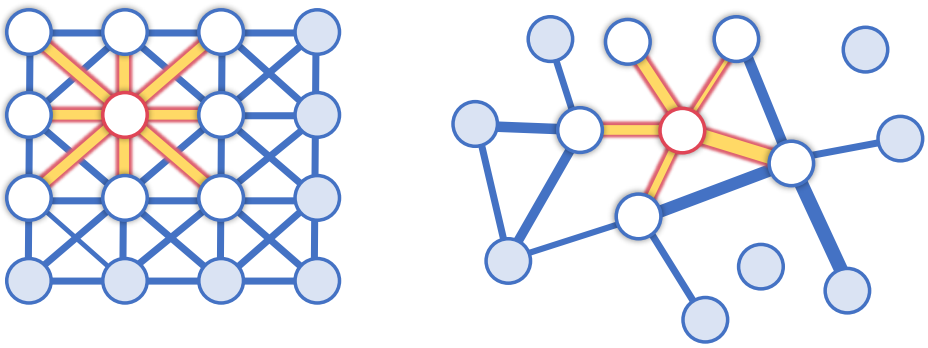


Figure 2.8.: The continuous convolution generalizes the two-dimensional convolution. The value of the red node is calculated by aggregating information from its connected partners, e.g., via weighted summation. (Left) An image is transformed into a graph. Pixels correspond to nodes, and edges mark neighbors. Each edge has the same weight. (Right) Nodes are unordered and vary in number of neighbors. Edges and their weights are arbitrary.

Wang et al. [Wan+18] presents one way to enforce continuity in the kernel with their successful **Deep Parametric Continuous Convolutional Neural Networks**. The convolu-

tion integral of Equation 2.24 is approximated by a sum normalized by the number of evaluated points N, inspired by Monte Carlo integration,

$$\begin{aligned}\{f * g\}(\boldsymbol{x}) &= \int_{\infty}^{\infty} f(\boldsymbol{y}) g(\boldsymbol{x}-\boldsymbol{y}) d\boldsymbol{y} \\ &\approx \sum_{i}^{N} \frac{1}{N} f\left(\boldsymbol{y}_i\right) g\left(\boldsymbol{x}-\boldsymbol{y}_i\right)\end{aligned} \tag{2.27}$$

with $\mathrm{Dom}(g) = \mathbb{R}^D$. For g, a **Multi-Layer Perceptron (MLP)** is used as an attempt to store the infinite number of function values. According to the **Universal Approximation Theorem** of Hornik et al. [HSW89], estimating any continuous function using an MLP with at least one hidden layer and a sufficiently large number of hidden neurons is possible. A drawback of using MLPs is their high parameter count [Wan+18].

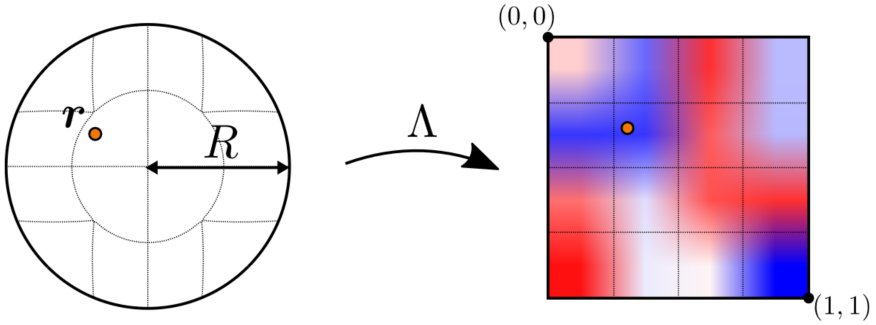


Figure 2.9.: Three-dimensional sphere with radius R mapped onto a normalized cube with the values of its points defined by the grid of g, from Ummenhofer et al. [Umm+20]. Colors illustrate the trilinear interpolation of grid values.

2.5.4 Continuous Convolution

A primary research object of this thesis is the **continuous convolution** of Ummenhofer et al. [Umm+20]. This operation translates the convolution to Lagrangian particles and combines ideas from the above variants. The function f is sampled with a finite number of points, not on a grid. Similar to the parametric convolution, g is continuous but does not use an MLP. Instead, g returns values from a learnable grid where the points between are interpolated, as shown in Figure 2.9. For a set of particles $\{1, \ldots, N\}$ with feature vectors $\{\mathbf{f}_1, \ldots, \mathbf{f}_N\}$ and positions $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ the value of the continuous convolution $(f * g)$ at point $\mathbf{x}$ is defined as

$$(f * g)(\boldsymbol{x}) := \frac{1}{\psi(\boldsymbol{x})} \sum_{i \in \mathcal{N}(\boldsymbol{x}, R)} a_R\left(\boldsymbol{x}_i, \boldsymbol{x}\right) f_i g\Big(\Lambda\left(\boldsymbol{x}_i - \boldsymbol{x}\right)\Big). \tag{CC1}$$

The features f_i of each particle i within a radius of R around $\mathbf{x}$, denoted by $\mathcal{N}(\boldsymbol{x}, R)$, are summed and weighted by g. The window function $a_R(x_i, x)$ can force a smooth response

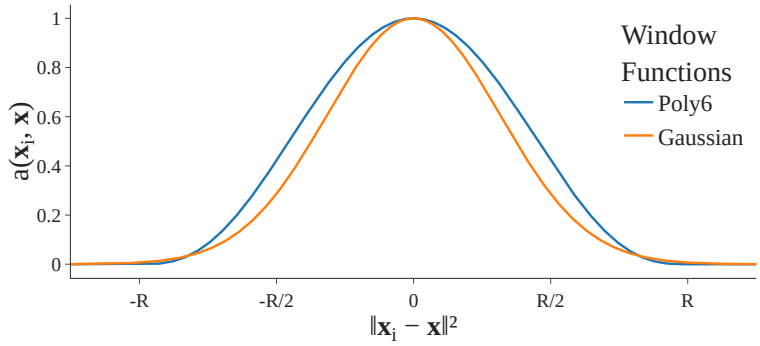


Figure 2.10.: The `Poly6` and `Gaussian` window function scale down the influence of particles further away.

of the convolution by returning values closer to 0 the closer the distance between its input positions is to R. Ummenhofer suggests the constant,

$$a_R(x_i, x) = 1 \tag{2.28}$$

or a variant of the popular `Poly6` SPH kernel, [MCG03]

$$a_R(\mathbf{x}_i, \mathbf{x}) = \begin{cases} \left(1 - \frac{\|\mathbf{x}_i - \mathbf{x}\|_2^2}{R^2}\right)^3 & \text{for } \|\mathbf{x}_i - \mathbf{x}\|_2 < R, \\ 0 & \text{else.} \end{cases} \tag{2.29}$$

During this work, we also will use a `Gaussian` window,

$$a_R(\mathbf{x}_i, \mathbf{x}) = \begin{cases} \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}\|_2^2}{R^2}\right) & \text{for } \|\mathbf{x}_i - \mathbf{x}\|_2 < R, \\ 0 & \text{else.} \end{cases} \tag{2.30}$$

Figure 2.10 illustrates the window functions a_R. The term $1/\psi(\boldsymbol{x})$ is an additional normalization factor that can be set to

$$\psi(\mathbf{x}) = 1 \quad \text{or} \quad \psi(\mathbf{x}) = \sum_{i \in \mathcal{N}(\mathbf{x}, R)} a_R(\mathbf{x}_i, \mathbf{x}). \tag{2.31}$$

The mapping function Λ translates relative coordinate values, e.g., from a unit sphere to a unit cube. Otherwise, possibly, corner values of the grid of g are not well used. This can be easily seen by drawing a circle on a checkered sheet of paper. Some cells of the grid are never accessed because the circle defines the radius in which particles are found. Figure 2.11 shows this. The problem gets worse in three dimensions [Umm+20].

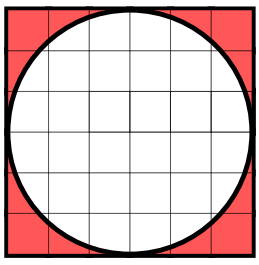


Figure 2.11.: Cells in corners are not well utilized.

The continuous convolution layer is more lightweight than the one implementing the deep parametric convolution [Umm+20]. It

does not use a multi-layer perceptron for g but a grid of values. It, has fewer free axes and fewer learnable parameters, resulting in less complexity. Nevertheless, we know from Ummenhofer et al. [Umm+20] that it is sufficient to achieve good simulation results. We investigate whether adequate results can be obtained by an even simpler form of the continuous convolution. We consider the case where

$$a_R(\mathbf{x}_i, \mathbf{x}) = 1, \quad \psi(\boldsymbol{x}) = 1 \quad \text{and} \quad \Lambda = \mathbb{1}, \tag{2.32}$$

with $\mathbb{1}$ being the identity function. Equation CC1 then simplifies to

$$(f * g)(\boldsymbol{x}) = \sum_{i \in \mathcal{N}(x,R)} f_i g\left(\boldsymbol{x}_i - \boldsymbol{x}\right). \tag{CC2}$$

Let $\{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$ be a set of particle features representing particles, $\{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$ be a different set of positions where the convolutional is evaluated, G be the earlier described filters or coordinate mappings and D be the diameter of the convolution, then this evaluation is denoted by

$$[\mathbf{f}_1, \ldots, \mathbf{f}_M] = \text{CConv}\left(\{\mathbf{p}_1, \ldots, \mathbf{p}_N\}, [\mathrm{x}_1, \ldots, \mathbf{x}_M], G, D\right). \tag{2.33}$$

Note that the evaluated positions can, but do not have to be, the particle positions and N can differ from M. Output are the convoluted features for each of the evaluated positions [Umm+20].

Chapter 3

Network Design

We familiarize ourselves with the ConvNet architecture. From a software development perspective, the architecture gives a network its structure, akin to a code skeleton [Kar19]. The dataset defines the behavior of the network [Kar19]. The training process compiles and compresses the dataset, encoding the behavior into the network by filling its weights and biases [Kar19]. We examine these elements individually in the Sections 3.1, 3.2, and 3.3. Using the knowledge gained, we identify modifiable components in the architecture and training process that can help us improve performance in Section 3.4.

3.1 Architecture

The convolutional architecture provided by Ummenhofer et al. [Umm+20] can be viewed as a Graph Neural Network with four message passing steps and an equal effective depth. The network can be decomposed into nine units consisting of one Continuous Convolution (CConv) for boundary handling, four CConvs for fluid particles, and four **Fully Connected (FC) layers** that learn particle features distributed over four layers as shown in Figure 3.1. For both fluid and boundary particle CConvs, the output points are the positions of the fluid particles $\mathbf{x}_i$. Filters and filter extents are the same for all CConvs [Umm+20].

We distinguish between fluid particles and boundary box particles. A fluid particle i is assigned the feature vector $\mathbf{f}_i = [1, v_{i1}, v_{i2}, v_{i3}, \nu_i]$, where the first component is the constant scalar 1, followed by the three components of the velocity $\mathbf{v}_i$ and the viscosity ν_i of the particle. The feature vector of a boundary box particle i is set to its box normal vector $\mathbf{N}_i$, i.e. $\mathbf{N}_i \in \{[1,0,0], [-1,0,0], [0,1,0], \ldots\}$. These vectors and the particle positions $\mathbf{x}_i$ serve as input to the neural network. In the original paper, the resulting output is a correction to the position Δx of particle i that takes into account both other particle interactions and collision handling with the scene [Umm+20].

The neural network is adapted to the density gradient problem by replacing ν_i with the calculated particle density ρ_i. Instead of Δx, the ConvNet learns the density gradient $\frac{D}{Dt}\rho_i$ or $\nabla_i \rho_i$ for particle i, where the former describes the temporal and the latter the spatial density gradient. We examine the layers of the network as illustrated in Figure 3.1.

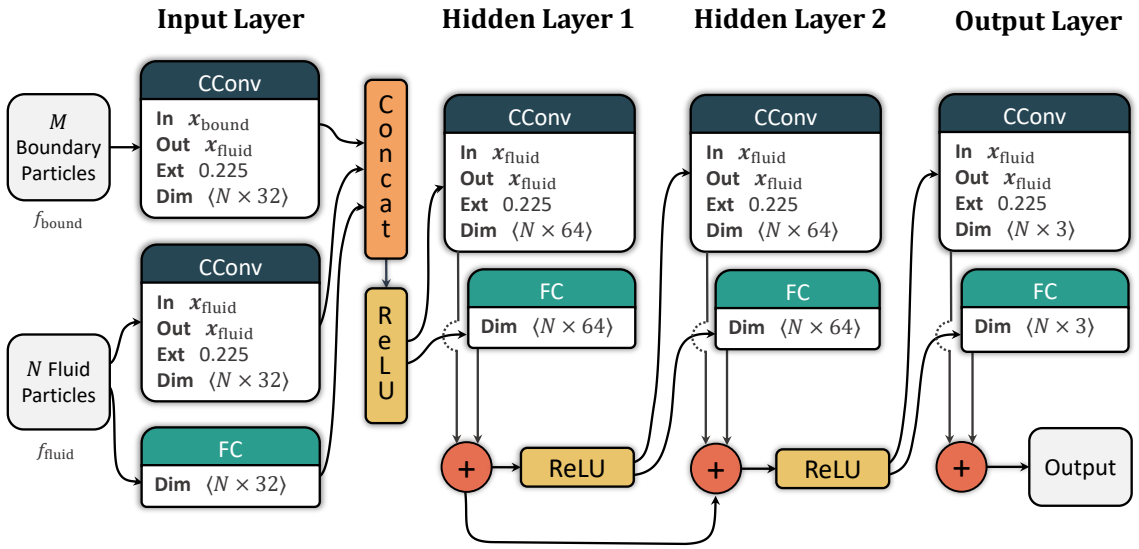


Figure 3.1.: The ConvNet architecture.

Input Layer. The input layer encodes the particle graph into latent space while already doing a round of message passing. This layer contains the single CConv for boundary handling, a CConv for fluid particles, and an FC layer for encoding particle features into latent space. The input features for each layer are the aforementioned $\mathbf{N}_i$, $\mathbf{f}_i$, and $\mathbf{f}_i$. The particle handling CConv takes as input the positions and features of the boundary particles but is evaluated on the positions of the fluid particles. This difference makes it an exception to all other CConvs in the network, which use the same input and output positions of the fluid. All three units output 32-component feature vectors. They are transformed by a **Rectified Linear Unit (ReLU)** and concatenated into feature vectors of length 96 [Umm+20].

Hidden Layer 1. This layer and subsequent layers are parallel pairs of a continuous convolution and a fully connected layer. They perform a message passing step. The output of each element is of length 64. These vectors are summed and transformed via ReLU [Umm+20].

Hidden Layer 2. This layer has the same elements with an additional skip connection. The skip connection in the paper transports the gradient of the previous output into the ReLU, which differs from the implementation where the skip connection is placed after the activation unit [Umm+20]. Our implementation follows the paper.

Output Layer. The output layer does the final round of message passing, translating the latent space graph back into the final predicted values for each particle. It again uses a CConv and an FC layer. Their outputs are summed but not transformed by an activation function [Umm+20].

3.2 Dataset

> *"The first step to training a neural net is to not touch any neural net code at all and instead begin by thoroughly inspecting your data. This step is critical. I like to spend copious amount of time [...] scanning through thousands of examples, understanding their distribution and looking for patterns."*
>
> — Andrej Karpathy [Kar19]

The dataset defines the behavior of the network [Kar19]. Ideally, it is large and has a diverse and balanced representation of all relevant scenarios that the final model should learn [Kar19]. Imbalances or underrepresentation can skew the network's output. Missing scenarios may lead to significant prediction errors. Some parts of the model may remain untrained or have no encoding of the desired behavior. In the context of a GNN-based simulator, this leads to unnatural behavior or instability. For example, a network trained on water will struggle to simulate fluids with different viscosities and densities, such as honey or sand. Unless otherwise enforced, it will also fail to correctly understand the concept of rigid bodies or impassable obstacles.

Our experiments use the pre-established and quality-controlled default and Dynamic Particle Interaction Network (DPI-Net) dam break datasets from Ummenhofer et al. [Umm+20] and Y. Li et al. [Li+19]. The default dataset consists of several three-dimensional SPlisHSPlasH [Ben+17] simulations generated via DFSPH [BK15]. Objects are spawned in random rotations and dropped into a box. A high-fidelity solver was run with time steps down to 0.001 seconds and up to 0.02 seconds. The DPI-Net dataset has similar scenarios where fluid boxes are placed inside static boxes. It was generated using Nvidia Flex [Mac+14], a real-time, position-based physics simulator.

We shuffle data at training but not at validation or test time, allowing for comparison of different training runs.

3.2.1 Ground Truth

In machine learning, the ground truth, or target, refers to the exact outcome associated with a data point that our model tries to predict [Zho18]. We construct the ground truth analytically by recomputing the density information and then computing the density gradient values using the same smoothing length $h = 0.05$ and cubic spline kernel. These calculations use the equations D1, D2, D3 and K1. Is is also possible to obtain gradient data via a finite difference approximation scheme, by tracking the particles' density changes over multiple time steps and approximating $\frac{D}{Dt}\rho_i \approx \frac{\Delta\rho_i}{\Delta t}$. This method is less accurate and introduces the need to control the size of the time step. It has no relevant advantages,

making the exact, analytical approach the more favorable choice. Figure 3.2 shows a sample frame of the dataset used, with the density and gradient data color-coded.

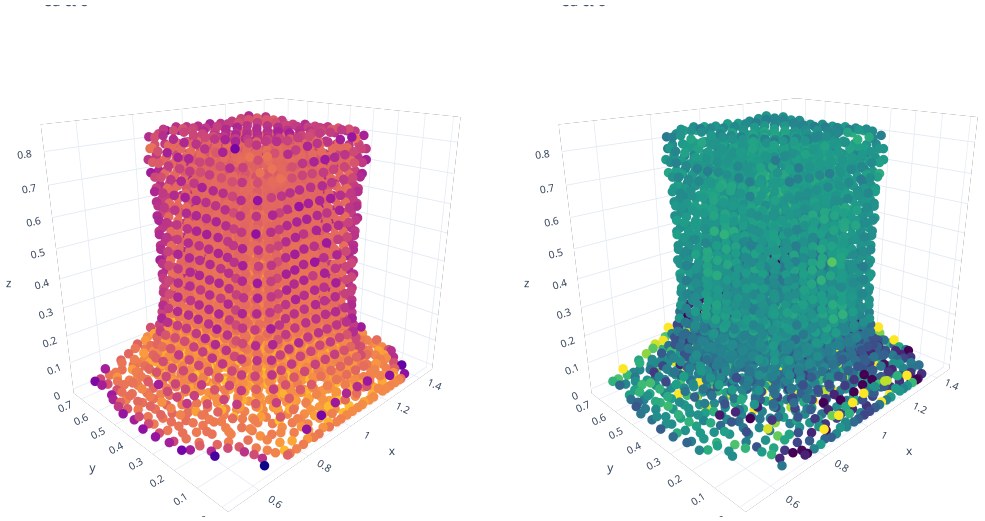


Figure 3.2.: A single simulation frame of the DPI dataset. The color encodes the density or change of density of the particles. (Left) The density is calculated using the same smoothing length and cubic spline kernel as the simulation. (Right) The density gradient is calculated using the analytical formula.

3.3 Training

3.3.1 Loss Function

The **Mean Squared Error (MSE)** based on the L^2 norm is calculated between the density gradient and the predicted values, i.e.,

$$L_2\left(\nabla\rho_t^{\text{pred}}, \nabla\rho_t^{\text{target}}; \theta\right) = \frac{\left\|\nabla\rho_t^{\text{pred},\theta} - \nabla\rho_t^{\text{target}}\right\|^2}{N} \tag{3.1}$$

where N is the number of samples in a batch. The model parameters θ were optimized with the Adam optimizer [KB17], using a mini-batch size of 2. The MSE penalizes larger errors more severely due to its squaring operation. This behavior is useful for fluid simulations, where outliers in density for a single timestep can lead to explosive forces and challenge the stability of the entire simulation, as discussed before [LL10].

3.3.2 Noise

Introducing noise into the data can improve stability [San+20]. Every model prediction contains a small error because it does not exactly reproduce the target data. In a typical rollout, these errors accumulate as they feed back into the simulator's input. The model

may begin to encounter states during rollout that were not present during training. At this point, the behavior of the GNN becomes practically undefined, which usually leads to significant errors.

Noise in deep learning is often used to reduce overfitting [Bis95, p. 347]. The CConv can overfit like any other layer, as the appendix Figure A.1 shows. This overfitting, however, never happened during our training runs with the DPI and default dataset. Sanchez-Gonzalez et al. [San+20] uses over multiple time steps accumulating random walk noise. This input corruption could stabilize longer rollouts. It rewards the correct and accurate network prediction even when the input contains errors. We only consider single time steps. Our implementation allows uniform noise such as $U(0, \sigma = 0.0005)$. However, we keep $\sigma = 0$ for the following chapters, as it complicates the hyperparameter search and consider it, while interesting, beyond the scope of this work.

3.3.3 Normalization

The precomputed density and density gradients are normalized with a mean of 0 and a standard deviation of 1. This type of normalization is standard to ensure stability and convergence [LeC+98]. It guarantees that processed feature values of different ranges are all shifted in the same ranges. For GNNs applied to SPH problems, this normalization appears to offer only minimal improvements. Sanchez-Gonzalez et al. [San+20] reports faster training times but no enhancement in network convergence.

3.3.4 Software and Hardware

The neural network is implemented using the GPU accelerated tensor and neural networks library PyTorch [Pas+19] version 1.13.1 with CUDA 11.6 support and PyTorch Lightning [Fal19] version 2.0.3, a lightweight PyTorch wrapper. Ummenhofers continuous convolutional layer is part of the Open3D-ML [ZPK18] library version 0.17.0. It describes itself as an extension of Open3D for three-dimensional machine learning tasks. Both scripts and the distributed hyperparameter tuning library Ray Tune [Lia+18] version 2.6.1 were used to orchestrate the training runs.

The experiments were performed on an Intel Core i7-3770K processor and an Nvidia GeForce GTX 1080 Ti graphics processing unit.

3.4 Search Space

The search space is the set of all possible hyperparameter configurations. It grows exponentially with the number of tuned parameters. An exhaustive computational exploration of the space is often practically impossible [Bis+21]. For example, a non-

parallelized Grid Search over 5 parameters with on average 4 possible options and 90 minutes of training time requires $4^5 \cdot 90 = 92,160$ minutes or 25 hours. An analogous search with 20 parameters requires more than 3.1 million years to complete.

It is also often necessary to fix bugs or change parts of the network during development. Each of these changes potentially renders the current set of hyperparameters ineffective. Consequently, the search space should be kept small and training times short [Bis+21].

For the ConvNet architecture, we identify the following parameters worth exploring in our initial search space.

Kernel size. The size of the kernel sets the spatial resolution of the learned filters within the continuous convolution. For instance, a $[3, 3, 3]$ kernel corresponds to a $3 \times 3 \times 3$ grid of values. When calculating the value for a given point, the continuous convolution maps three-dimensional space onto the filter using a coordinate mapping function Λ. Then, it uses the interpolated value of the requested cell for its computation. We use a scalar kernel size $s \in \mathbb{N}$ as a shorthand notation for a grid $[s, s, s]$.

Default value:
4

Search space:
$\{1, 2, 3, 4, 8, 16\}$

Dimensionality. This parameter defines the capacity and complexity of the network. We differentiate between the number of output channels of the three elements composing the input layer, the number of output channels per hidden layer, and the number of hidden layers. The default dimensions are $[96, 64, 64, 3]$, corresponding to $32 \cdot 3 = 96$ output channels of the input layer, 64 output channels for each hidden layer, and an output with 3 dimensions. Any additional hidden layer added to the second one will be identical, e.g., for $[96, 64, 64, 64, 3]$, the second hidden layer will be duplicated. See Section 3.1 for details. Let $N \in \{0, 1, 2, 3, 4\}$ be the number of hidden layers, $C \in \{16, 32, 48, 64, 128\}$ be the number of output channels per hidden layer, and $I \in \{4, 8, 16, 32, 64\}$ be the output channels per unit of the input layer.

Default value:
$[32 \cdot 3, 64, 64, 3]$

Search space:
$[I \cdot 3] \times [C]^N \times [3]$

Coordinate mapping. For the coordinate mapping Λ, the available choices are `identity`, `ball_to_cube_radial`, and `ball_to_cube`. The mapping defines how the search sphere defined by the filter extents is mapped to the values of the kernel. `ball_to_cube` transforms the relative coordinates of the search ball into a unit cube. It guarantees that each cell of the kernel is connected one-to-one to a piece of volume in the search ball. The `ball_to_cube_radial` function does the same but uses radial coordinates. The `identity` mapping leaves relative coordinates unchanged.

Default value:
`ball_to_cube`

Search space:
`ball_to_cube`, `ball_to_cube_radial`, `identity`

Filter Extents. The filter extents give the diameter d of the search for which particles will be found. They define the maximum distances at which a single particle can affect the next particle. Increasing the filter extents dramatically increases the training time and memory requirements of the training. The volume of the search ball increases with $V = \frac{4}{3}\pi(\frac{d}{2})^3$ cubically, and thus the size of the neighborhood lists also increases in $O(d^3)$. Larger extents enable the network to learn more general trends from the data. The size of a particle is 0.025 in our datasets. Initial trials with extents over 0.3 showed run times of over 24 hours for 15 epochs.

Default value:
0.225

Search space:
$(0.075, 0.25)$
linear

Activation functions. The activation function is a functional transformation when passing the output between layers. It introduces the necessary nonlinearity into the network. We compare the performance of `ReLU`, `Tanh`, `Leaky ReLU`, and `GELU` (Gaussian Error Linear Unit). The latter two are variations of ReLU with non-zero gradient values. Their modifications fix weaknesses by having non-zero gradient values, such as the dying neuron problem or vanishing gradients [Kar23]. The expected effect on total loss and training time is small. The `GELU` smoothes its output around 0 and is based on an exponential function, which may be advantageous since many physical relationships are of this nature. The `Tanh` (hyperbolic tangent) is a standard and smooth activation function that often learns slower. It is included for comparison.

Default value:
`ReLU`

Search space:
`ReLU`, `Leaky ReLU`, `GELU`, `Tanh`

Window function. The radial window function $a_R(x_i, x)$ applies an additional weighting to the output of the continuous convolution, as explained in subsection 2.5.4. Its use can be justified in that it approximates the kernel function used to generate the density data, potentially speeding up the training time and guiding the process to more favorable optimums. The window functions examined are `Poly6` from Equation 2.29, the `Gaussian` function from Equation 2.30, or `None`. The first two guarantees a smooth response of the continuous convolution, while using no window function forces the network to learn scaled values. Whether the window function can be made obsolete is examined later.

Default value:
`Poly6`

Search space:
`Poly6`, `Gaussian`, `None`

Ignore query points. This option determines whether the continuous convolution weights the points around which the search ball is spanned with 0, effectively canceling their effect. In other words, this setting ensures that particles cannot find themselves as neighbors. The rate of change of a particle's density depends only on its neighbors. Therefore, enabling this option could help approximate the density gradient. Alternatively, if query points are not ignored, the network has more opportunities for information about a point to affect itself.

Default value:
`True`

Search space:
`True`, `False`

Learning rate. The learning rate determines how large the mini-batch gradient updates are. The original paper uses the Adam optimizer [KB17] with the standard learning rate of 0.001. They also implement a scheduler to gradually reduce the learning rate over multiple steps and optimize their loss function throughout 50,000 iterations. In our experiments, a slightly higher learning rate of 0.002 with no decay produced the best results. Furthermore, we capped the training at $32,000$ gradient updates to reduce the training and search time.

Default value:
0.001

Search space:
$(0.0001, 0.01)$
loguniform

The **batch size** gives the number of samples the network processes for a single learned parameter update. In a sequential setting, smaller batch sizes allow for more frequent parameter updates. However, in a modern deep learning parallel processing scenario, the performance bottleneck usually lies in memory loading. The batch size should be as big as possible for maximal GPU utilization. Bigger batch sizes result in higher-resolution gradient estimations of the MSE. This improvement in precision allows for more aggressive learning rates, thereby speeding up training [Smi18].

We set the batch size to two. Our samples consist of thousands of particles with tens to hundreds of thousands of interactions. The size gives a good compromise for GPU utilization and loss landscape approximation. We keep it constant for all trials to allow for a fairer comparison of models.

Chapter 4

Exploring Hyperparameters

The correct choice of hyperparameters is crucial for the performance of the network. This chapter deals with the search for optimal parameters for the architecture, the so-called **Hyperparameter Optimization (HPO)**. We are particularly interested in the influence of the previously identified parameters of the ConvNet architecture. Section 4.1 determines whether the initialization of the network parameters significantly impacts the performance of the GNN, which would complicate the HPO problem. Section 4.2 then evaluates how a model with the original hyperparameter configuration performs on the density gradient prediction problem. This evaluation provides a baseline against which other models can be compared. Finally, Section 4.3 performs random and adaptive searches over hundreds of configurations to find candidate models that outperform the baseline. After each search, we examine specific hyperparameters and their utility for performance.

4.1 Importance of Initialization

We show that the impact of the randomness in the initialization of the weights and biases on the performance of the trained network is negligible for our cases. This finding justifies the results of the subsequent hyperparameter exploration. It shows that a single evaluation of a hyperparameter configuration through training is sufficient to measure the quality of that configuration. Otherwise, each configuration would require multiple re-evaluations or a step where initial weights and biases are pre-optimized.

The weights and biases of the ConvNet are set with a uniform **Xavier initialization** [GB10], and the weights of the linear layers are set to zero. Xavier initialization is a method designed to ensure that activations and loss gradients remain constant during forward and backward passes of the network. As a result, the learning process gets balanced. Typical problems like vanishing gradients dramatically disrupting training are mostly avoided [Kar23].

These initial parameters define the model's starting point on the multidimensional loss landscape. A danger with gradient descent and its various enhancements, like Adam, is that bad initial starting points may lead the training process on trajectories to inescapable, suboptimal local minima [Smi18]. In other words, poor initial performance could result in a suboptimally configured network with worse-than-expected capabilities. Consequently,

architectural design decisions in the form of hyperparameter choices may have less impact than randomness in the initialization phase. We show that this is not the case.

4.1.1 Test Setup

We use the parameter selection shown in Table 4.1 and assume that it is representative of other hyperparameter configurations. The weights and biases of the GNN are set deterministically using seeds. The subsequent training and its processes, such as the mini-batch selection, are kept independent and random. A failure of this step would result in every run under a given seed collapsing to the same deterministic result.

	Parameter	**Value**
1	Learning rate	0.004354
4	Hidden layers	1
5	Dimensions	$[32 \cdot 3, 64, 1]$
6	Activation function	`ReLU`
7	Kernel size	$[4, 4, 4]$
8	Filter Extent	0.225
9	Interpolation	`trilinear`
10	Window function	`Poly6`
11	Coordinate mapping	`ball_to_cube_radial`
12	Ignore query points	`True`
13	Dense Layer	`False`
14	Align corners	`True`
15	Normalize	`False`
	Trainable parameters	**836,290**

Table 4.1.: The hyperparameter configuration for the ANOVA test.

We evaluate the initial performance of neural networks initialized with 1000 randomly selected seeds on 140 random samples of the DPI dataset. Figure 4.1 shows the distribution of initial losses resulting from 140,000 forward passes. The distribution and boxplot of the initial model performance of 1000 seeds has a mean of 0.970 and a standard deviation of 0.00708. On average, the performance over the 140 samples is better than the expected average loss of 1.00. This deviance indicates that the selected samples slightly favored the network initialization process. The worst, median, and best-performing seeds on the validation sample yielded losses of 0.957, 0.968, and 1.03. The worst seed is an extreme

outlier with 8.4 sample standard deviations from the mean. For further comparison, we select the three seeds with the initial best, worst, and median performance. A network is trained ten times for each chosen seed using 9,900 mini-batches. Figure 4.2 displays the resulting loss values.

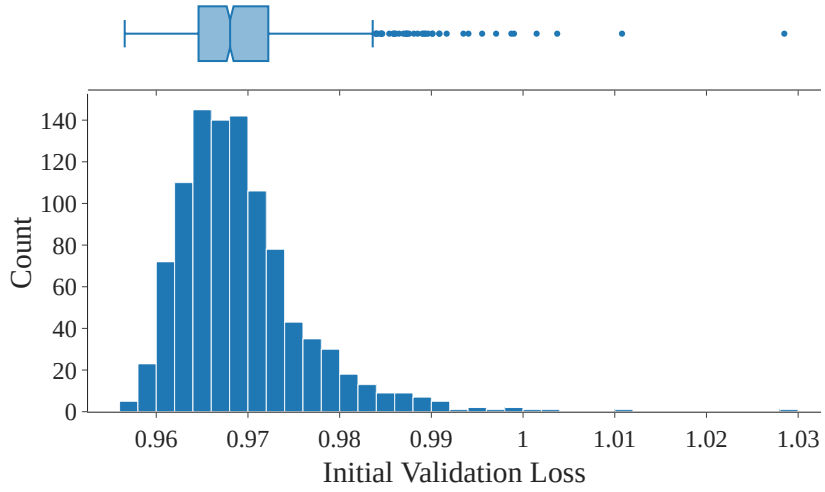
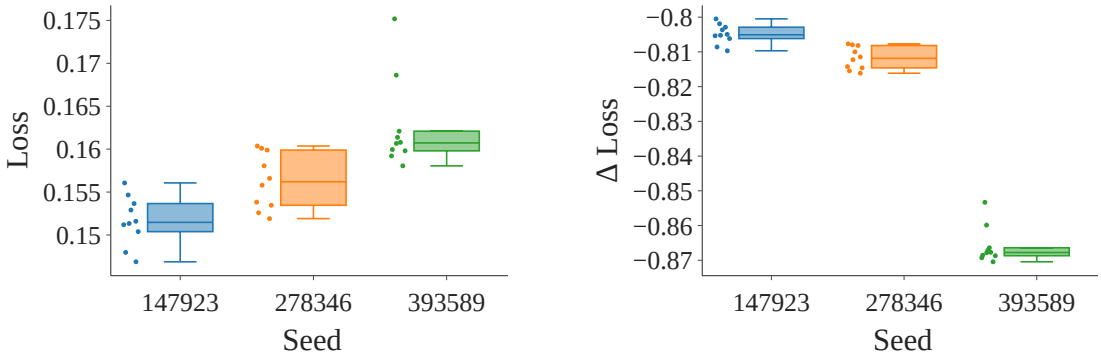


Figure 4.1.: Initial performance of 1000 seed-initialized models.



(a) Final loss values

(b) Change in loss

Figure 4.2.: The losses of the 30 models trained for 9,900 iterations. The initially best performing models tended to maintain their performance advantage but improve the least, and vice versa.

4.1.2 Analysis of Variance

A one-way **Analysis of Variance (ANOVA)** is used to determine whether there are any statistically significant differences between the means of the three groups of loss values. The problem statement leads to the following hypotheses:

Null Hypothesis (H_0). There is no significant difference in the mean loss values for each seed.

Alternative Hypothesis (H_A). There is a significant difference in each seed's mean loss values.

The test requires some basic conditions to be fulfilled. Each observation must be independent. Independence is satisfied because the data selection in each run's training process is independently randomized. Homoscedasticity is expected, i.e., the variances for each group must be approximately equal. Constant variance appears to be given except for the third seed and its two outliers, as can be seen in Table 4.2. The loss values for each group should ideally be normally distributed. This assumption may or may not be met, particularly for the group with seed 278346. It appears to deviate from normality. We assume that each group is normally distributed.

	Best	**Median**	**Worst**	**Total**
Seed	147923	278346	393589	-
Mean	0.151666	0.156258	0.162574	0.156833
Variance	0.000008	0.000011	0.000028	0.000035

Table 4.2.: Training loss values after 9,900 gradient updates per seed.

With $N = 30$ observations, there are $N - 1 = 29$ total degrees of freedom (DFT), $k - 1 = 2$ between-group degrees of freedom (DFB), where k is the number of groups and $N - k = 27$ within-group degrees of freedom (DFE). The total sum of squares (SST) is 0.00102, the between-group sum of squares (SSB) is 0.000600, and the within-group sum of squares (SSE) is 0.000423. We calculate

$$F_{2,7} = \frac{SSB/DFB}{SSE/DFE} = 19.14498, \tag{4.1}$$

which has a p-value of 0.00000665. Under the assumption that there is no significant difference in the mean loss values per group, the probability of having loss values deviating this much or more is less than 1 %. Hence, we have significant evidence to reject H_0 at the $\alpha = 0.01$ significance level. Under the same fixed hyperparameters, the randomness in the initialization of the weights and biases can significantly affect the network's performance.

4.1.3 Interpretation

Although there is a significant difference in performance for each seed group after training, context is critical in this case. The seeds were selected as the most extreme cases out of 1,000. Their initial evaluation necessitated 140,000 forward passes or 70,000 mini-batch gradient updates, ignoring backpropagation. This computational effort far exceeds the 9,900 updates each model was trained in this experiment. If even a fraction of the computational time had been spent on improving the results of the worst run, it likely outperformed the best group. This finding is supported by Figure 4.2b. It shows that the runs in the worst-performing group improved the most in the same time duration. Relative performance differences shrink throughout training.

Moreover, the deviations from the median were minimal. All but two observations fell within 10 % of the performance in the median group. The variability in loss values after training is minimal compared to the differences observed between architecture variations. The average loss value for the baseline parameters in Table 4.3, calculated over ten training sessions with 9,900 gradient updates each, is 0.106 with a variance of 0.000014, compared to 0.157 with this configuration. It is unnecessary to pre-optimize the initial seeds for the best ones. The approach is ineffective compared to regular backpropagation-based parameter updating. One possible explanation for why the initial seed may not be that relevant is that our mini-batch size of two generates imprecise gradient approximations. It causes the training process to rapidly jump over the loss landscape and thus skip any truly unfavorable training states.

As a counterargument, it could be argued that the seed's initial performance does not necessarily determine the model's final performance. The quality of the learning trajectory may start well but deteriorate later. Seeds, therefore, should be evaluated over multiple iterations. However, an additional evaluation of 10 different random seeds also showed losses very similar to the median group, where none of the observed values were as extreme as what can be seen in Figure 4.2. Interestingly, for all 30 runs, the general group order was generally stable. A run from the best group tended to outperform runs from other groups. As we see in the next section, this behavior persists when comparing different hyperparameter sets. The order of performance between models tends to remain consistent during training.

4.2 Baseline Performance

In machine learning, establishing the **baseline** provides a reference point against which other models can be compared [Ame18]. The baseline architecture is trained on a significant portion of the DPI dataset using the Adam optimizer with the slightly increased initial learning rate of 0.002 mentioned previously. The selected subset of the training dataset contains $n_{\text{train}} = 36{,}000$ samples compared to 7,710 in the validation and test datasets, corresponding to a $(0.7, 0.15, 0.15)$ split. We define an **epoch** as processing 1,000 samples from the dataset instead of the traditional deep learning definition of the entire dataset. As mentioned before, for datasets based on SPH simulations, the total size of the set is arbitrary because it can be easily expanded.

Parameter	Value
Learning rate	0.002
Batch size	2
Optimizer	Adam
Dimensions	$[32 \cdot 3, 64, 64, 1]$
Activation function	`ReLU`
Kernel size	$[4, 4, 4]$
Filter Extent	0.225
Interpolation	`trilinear`
Window function	`Poly6`
Coordinate mapping	`ball_to_cube`
Ignore query points	`True`
Dense Layer	`False`
Align corners	`True`
Normalize	`False`
Trainable parameters	**686,658**
Validation loss 32k	**0.0758**

Table 4.3.: The baseline

The network using the baseline architecture from the previous chapter is trained ten times over 32 epochs. The results are averaged to get a balanced view of the network's performance. The network registers 686,658 trainable parameters. It ends with a loss on the validation dataset of 0.0758. This loss value is a good result for the number of parameters, as we will show later. An overview of the hyperparameters used and their values is given in Table 4.3.

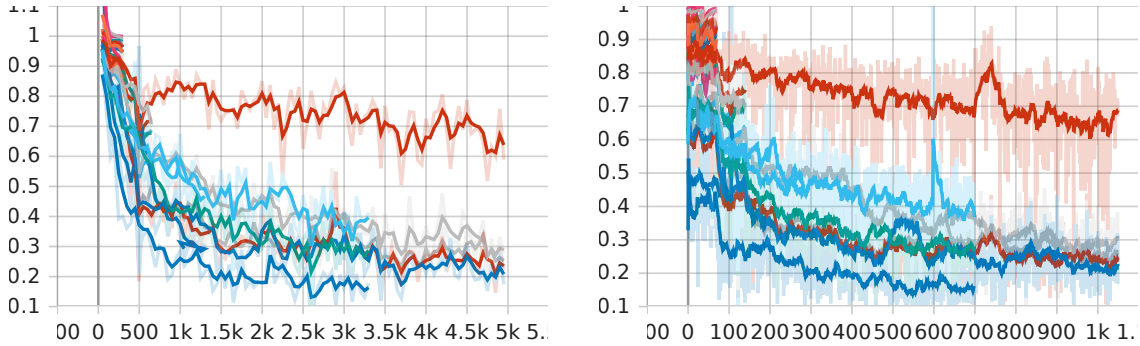
4.3 Search

We conduct individual hyperparameter tuning experiments and perform two random searches to determine the hyperparameters significantly impacting network performance and their relevant domains. The first hyperparameter search aims to identify performance-relevant hyperparameters. The second search assesses 100 configurations of a reduced set of parameters. We employ Bayesian Optimization to improve the results. We check an additional 100 hyperparameter sets. Due to technical limitations, we fix categorical hyperparameters to maximize performance with the remaining degrees of freedom. There are noticeable patterns in the loss plots. These patterns appear because, while training

itself is randomized, the trainloader object is serialized and copied for each run by Ray Tune. Thus, each model within a search uses the same randomized order of samples, allowing for better comparison.

4.3.1 Refinement of the Initial Search Space

Our initial search space from Section 3.4 is based on educated guesses and individual experiments. Table 4.4 summarizes it. The space is potentially too vast, which can lead to several problems. Promising hyperparameter configurations are typically hidden in minimal subsets of the search space. The larger the search space, the less likely it is to find a promising set of parameters. Additionally, there are regions in space that can be enormously expensive to explore. The complexity of the network and the training time grows at least quadratically or cubically with the number of channels, the kernel size, and the length of the filter extents. A poor parameter choice can lead to a gigantic function approximator that is difficult to evaluate with our limited computational resources. It is also not suitable for our purpose of learning the density gradient. Typically, in other deep learning scenarios, these networks also start to overfit, which risks losing their generalization ability as they start to memorize samples [Smi18]. The domains of each hyperparameter can also be misaligned For example, upper and lower bounds can exclude reasonable solutions. Numerical parameters can be suboptimally discretized. We are interested in lightweight configurations with good generalization capabilities for the network architecture without wasting too many computational resources. These constraints make it necessary to test and adjust the initial search space.



(a) Training loss over time.

(b) Validation loss over time.

Figure 4.3.: The training process for the adjustment of the initial search space.

A 30-epoch training run of the baseline network takes about 45 minutes. Initial experiments have shown that performance differences between architectures quickly become apparent after about 1,500 gradient updates. We thus limit the number of epochs to a low 3.3 or later 5, where the validation dataset is only checked in about 20 %. ASHA, the method

to prematurely abort suboptimal runs, is used with a conservative reduction factor of $\eta = 2$ and a single group $n = 1$ to avoid wasting time on unproductive runs while not eliminating promising ones. We use random search, a typical choice to identify potential problems of the initial search space [Kar23]. The total number of tested configurations is limited to 40, which gives us a rough estimate of the hyperparameter space. Ray Tune orchestrates the HPO process.

	Parameter	**Search Space**
1	Learning rate	$[0.0001, 0.01]$
2	Hidden layers	$[0, 1, \ldots, 4]$
3	Input layer channels	$[8, 16, 32, 64]$
4	Hidden layer channels	$[32, 48, 64, 128]$
5	Activation function	`ReLU`, `Leaky ReLU`, `GELU`, `Tanh`
6	Kernel size	$[3, 4, 8, 16]$
7	Filter Extent	$[0.075, 0.25]$
8	Interpolation	`trilinear`, `linear border`, `nearest neighbour`
9	Window function	`None`, `Poly6`, `Gaussian`
10	Coordinate mapping	`identity`, `ball_to_cube`, `ball_to_cube_radial`
11	Ignore query points	`True`, `False`
12	Dense Layer	`True`, `False`
13	Align corners	`True`, `False`
14	Normalize	`True`, `False`

Table 4.4.: The search space used for the initial 40 configuration search with ASHA. A maximum of 3,300 gradient updates were performed per run.

After 40 hours, 39 of 40 trials finished. Figure 4.3 shows the train and validation loss diagrams. Only trial 11 did not complete. It ran for 14 hours before being aborted. It was only 26 % complete. Its architecture used a hidden layer with a kernel size of 16 and employed filter extents of 0.245 for its continuous convolutions. Most other runs took as little as 10 minutes if canceled by the ASHA scheduler or about 1.5 hours if not canceled. Except for the red run, the scheduler quickly aborted less promising runs. The best model achieved a loss of 0.152 with only 3,300 gradient updates. It used three hidden layers corresponding to five message-passing steps, identity coordinate mapping, no window function, and `GELU` as its activation function.

The educated guesses for the initial search space were better than expected and require only minor adjustments. The maximum kernel size is reduced from 16 to 8 to reduce training times, limiting the number of parameters. In addition, the filter extents are limited to less than the baseline of 0.19. The assumption was that using large filter extents could cause neighbor lists to consume excessive memory due to the increased number of neighbors per particle, leading to thrashing [Den68]. The discretization of the number of channels for the input and hidden layers was made finer by adding more options. The hypothesis was that this might reveal patterns in the loss values that would otherwise not be apparent. The later third Bayesian search shows that this conjecture may be wrong. Filter extents aren't necessarily critical for time consumption. To counteract the increase in search space, the normalize and align corners options are set to `True` and `False`, respectively, analogous to the baseline. They showed little to no impact in the search. Table 4.5 shows the modified search space.

	Parameter	**Search Space**
1	Learning rate	$[0.0001, 0.01]$
2	Hidden layers	$[0, 1, \ldots, 4]$
3	Input layer channels	$[4, 8, 16, 32, 64]$
4	Hidden layer channels	$[16, 32, 48, 64, 128]$
5	Activation function	`ReLU`, `Leaky ReLU`, `GELU`, `Tanh`
6	Kernel size	$[1, 2, 3, 4, 8]$
7	Filter Extent	$[0.075, 0.19]$
8	Interpolation	`trilinear`, `nearest neighbour`
9	Window function	`None`, `Poly6`, `Gaussian`
10	Coordinate mapping	`ball_to_cube`, `ball_to_cube_radial`, `identity`
11	Ignore query points	`True`, `False`
12	Dense Layer	`True`, `False`
13	Align corners	`True`
14	Normalize	`False`

Table 4.5.: The modified search space was used for the second 100-configuration random search with ASHA.

4.3.2 Identifying Performance Relevant Factors

The effect and importance of each hyperparameter on model performance is evaluated. The CConv layer supports many options, as outlined in Section 3.4. Based on the results of Bergstra and Bengio [BB12] in Subsection 2.4.2, we suspect that only a tiny fraction of these options are relevant to make meaningful improvements to the baseline architecture. We perform a second, finer-grained random search over 100 hyperparameter configurations for confirmation. The most noteworthy change was the maximum training time for each run. For this search, up to 25 training epochs were performed. It increases the resolution of the first search as more points in space are evaluated. The data quality of each point also improves by investing more training time.

The time required on the same hardware using the restricted parameter space with seven epochs per run was 24 hours. Good trials consistently outperformed bad trials during training after 1,500 gradients updates, analogous to the first search and previous seed evaluation, justifying the use of the ASHA scheduler. The use of ASHA also improved the readability of the later plots by introducing more vertical space.

The relationships between parameters and model performance dynamics appear noisy. Differences per hyperparameter options are minor. The statistical variation observed in the Section 4.1 as seeds were evaluated suggests these patterns could be random. Otherwise, trends suspected from the first search are generally confirmed. Figure 4.4 displays scatter plots for each hyperparameter, with the option on the x-axis and the post-training validation loss of a model on the y-axis. The plots are color-coded based on the epoch in which the scheduler stopped training.

Coordinate mapping. The `identity` mapping had the best-performing trials compared to `ball_to_cube` and `ball_to_cube_radial`. Fewer or inaccessible corner cells were not a critical problem with the grid sizes used.

Window function. The performance from best to worst was `Gaussian`, `Poly6`, and no window. Likely, some degree of forced smoothing of the continuous convolution output helps the model to pick up the patterns in the data. This observation makes sense as it even puts the output of an untrained continuous convolution closer to the output of the cubic spline kernel used by the SPH method. However, some networks performed well even without a window. These models suggest that the networks with enough complexity can learn to smooth their response.

Activation function. All `ReLU` variants performed reasonably well. The default `ReLU` delivered the most successful configurations. `GELU` and `Leaky ReLU` did not provide the expected improvement. The smoothness they introduce may not significantly influence

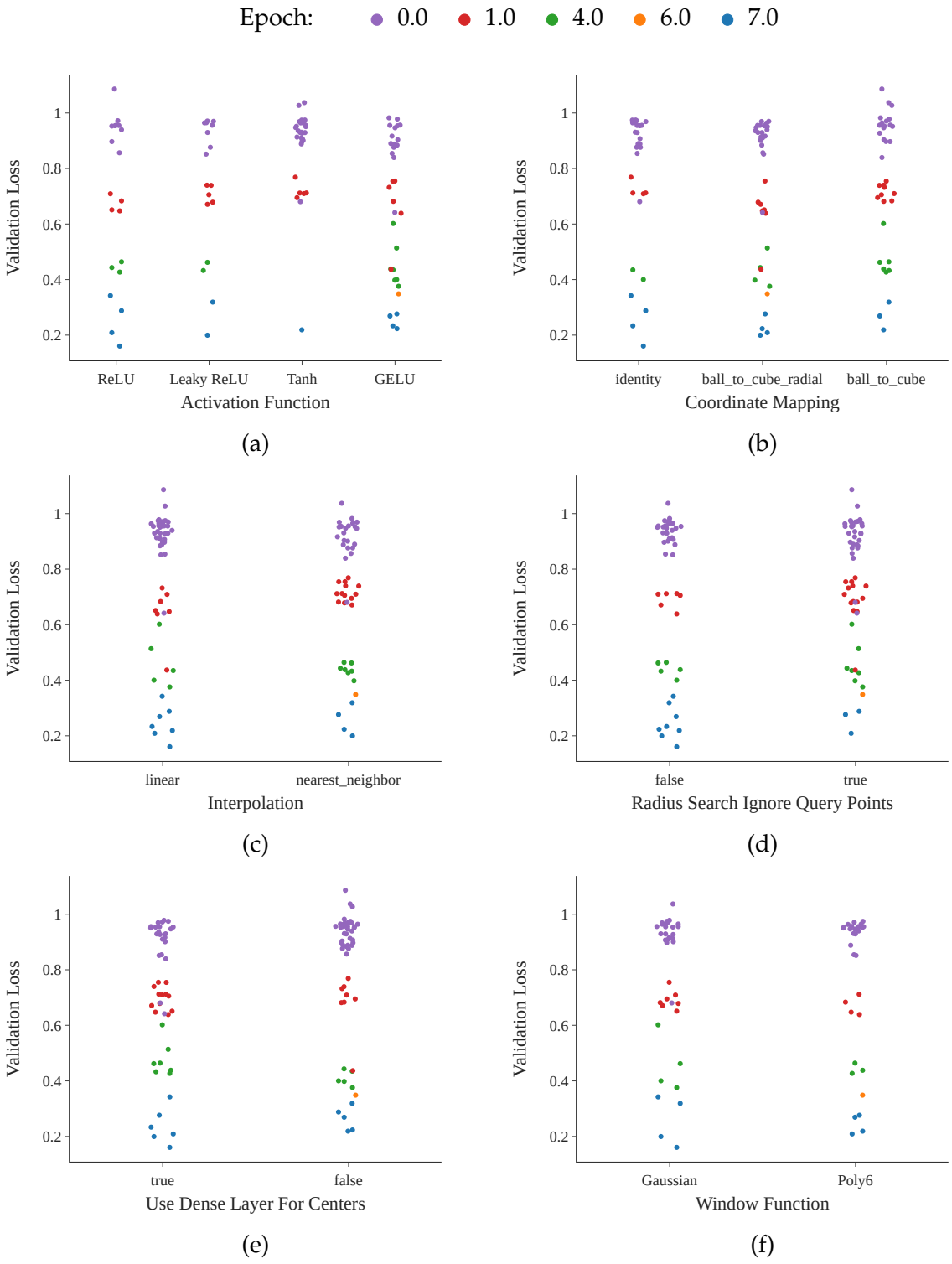


Figure 4.4.: The results of the second random search. The dots are colored by the epoch, after which the ASHA scheduler aborted the run.

the output. The `Tanh` unit underperformed the most. Further separate experiments with this activation function show that the learning process dramatically slows down when `Tanh` is used. All but one of the runs using it were aborted prematurely by the ASHA scheduler.

Interpolation. Trilinear interpolation performed slightly better than the less sophisticated nearest neighbor algorithm. Improving the smoothness of the convolution response is likely beneficial.

Ignore Query Points. Including the query points in the search results of the convolution provided better hyperparameter configurations. In other words, particles finding themselves during neighbor search and processing their own features helped the network produce slightly better predictions. Although the density gradient theory does not incorporate this data, the neural network may use the additional particle information, such as particle velocity, to enhance its understanding of the changes in density.

Center Dense Layer. This option adds a linear layer that adds the output of this layer as a bias for grid setups without a center element, i.e., setups with even kernel sizes. Of the 100 configurations, only attempts with kernel sizes of 4 and 8 were successful, though the sizes 5, 6, and 7 were not explored. Adding the dense layer also marginally improved the validation loss in the top-performing trials. This improvement is potentially due to the lack of a central element of sizes 4 and 8.

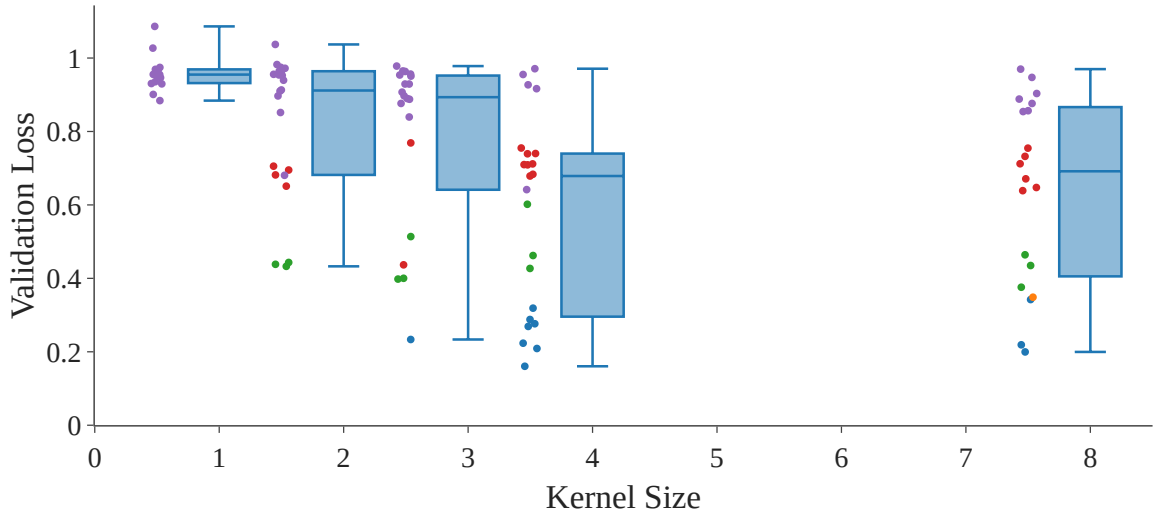


Figure 4.5.: Outcome in validation loss with respect to kernel size. The coloring is analogous to the previous figure.

Kernel Size. The grid size was one of the most significant predictors of model capacity, as seen in Figure 4.5. Networks with kernels of size one, meaning kernels that only store a

single value, showed next to no learning ability. They had all their trials canceled in the first epoch. An increase in kernel size up to 4 is associated with improvement. The size of the kernel establishes the minimum level of optimal performance attainable by the GNN. However, there is no observable distinction between size 4 and 8 kernels. We hypothesize that optimal performance occurs between sizes 4 and 8. The default baseline uses size 4. A kernel size of 8 is associated with significantly larger networks.

An increase in the trainable parameters of the instantiated model displayed in Figure 4.6 shows a clear association with higher maximum possible performance. The trend is even more pronounced than for kernel size. The kernel size may serve as a proxy for the total number of parameters. Given the general noise and other factors, the parameters are associated with a high correlation coefficient of 0.55. All networks with a validation loss of less than 0.3 used between 200,000 and 5,000,000 parameters. This range represents a zone where optimal model performance can be found. Performance declines for 10,000,000 or more parameters. Sufficiently large models may be able to learn the necessary adjustments for optimal performance. The following section tries to prove this hypothesis.

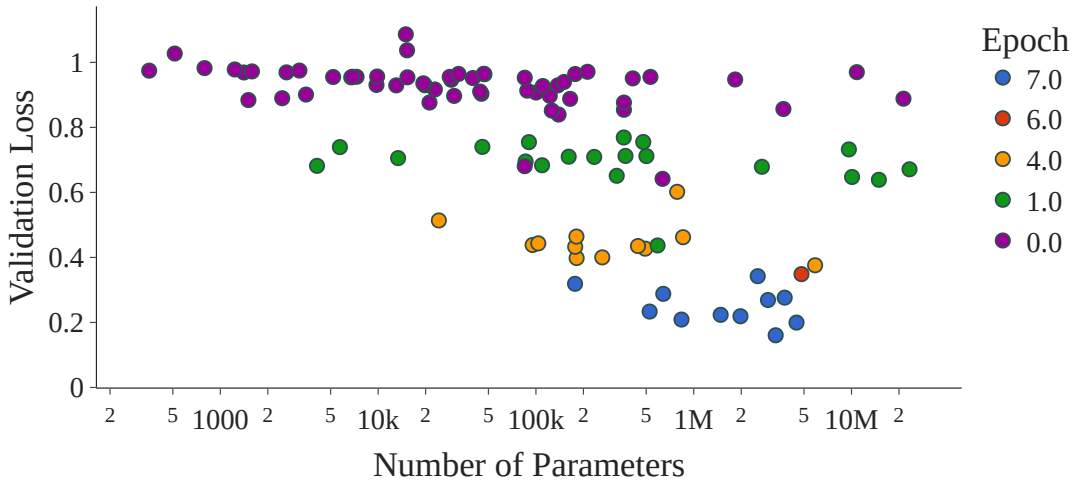


Figure 4.6.: The best performing models all had between around 200,000 and 5,000,000 parameters. The number of parameters defines the lower bound of the validation loss of the model.

4.3.3 Validating Adaptability

We conduct a third search to validate the hypothesis that a sufficiently large ConvNet can adapt and produce good results without a window function. We show that such a model exists, outperforming the already strong baseline model in predicting the SPH density gradient. We find that shorter filter extents and more conservative learning rates are

advantageous for performance under the given circumstances. The number of channels output by each layer is less influential than other hyperparameters.

This search employs Ray Tunes Bayesian Optimization as an adaptive algorithm that implements the method given by Nogueira [Nog14]. The algorithm is configured to minimize the validation loss by using ten random search steps for each trial. This configuration promises a better exploration of the objective function and better-performing models. The total number of trained models is 100. We do not use ASHA this time but a First-in-First-Out (FIFO) scheduler. ASHA can be applied to Bayesian Optimization but is not easily applicable to our scheduler. Additionally, not using ASHA might provide a different perspective on parameters, revealing patterns previously concealed. Each model is trained over six epochs to compensate for the increase in training time.

The Gaussian processes-based probability model assumes that the approximated function is continuous. Bayesian Optimization, therefore, expects its parameters to be continuous. The recommended workaround for non-continuous parameters is to allow floating-point inputs and floor them to the nearest integer value in the model instantiation. Within the respective boundaries, this renders all integer values accessible as potential model dimensions.

Parameter	**Search Space**
Learning rate	$[0.0002, 0.006]$
Hidden layers	$[1, 2, 3]$
Input layer channels	$[8, 9, \ldots, 32]$
Hidden layer channels	$[16, 17, \ldots, 64]$
Activation function	`ReLU`
Kernel size	$[3, 4, 5, 6]$
Filter Extent	$[0.12, 0.25]$
Interpolation	`trilinear`
Window function	`None`
Coordinate mapping	`identity`
Ignore query points	`False`
Dense Layer	`True`
Align corners	`True`
Normalize	`False`

Table 4.6.: The third search space.

We want to prioritize lightweight models. We make the following changes while considering the results of the previous search. The number of hidden layers is set to a minimum of one and a maximum of three. The number of channels for each unit of the input layer is 32, and for hidden layers, 64. The learning rate is limited between 0.0002 and 0.006. To show that the network can adapt without a window function, we use the simpler form of the continuous convolution shown in Equation CC2. The window function, coordinate mapping, ignore query points, dense layer, align corners, and normalize parameters are set to `None`, `identity`, `False`, `True`, `True`, and `False`, respectively. The resulting search space is in Table 4.6.

The HPO finished in 10 hours. 86 % of the explored models had less than 500,000 parameters. Several networks were generated that exceeded the performance level of the baseline at the 6-epoch mark. The hyperparameter configuration of the best model consisted of five message passing steps. It utilizes very short filter extents of 0.157, corresponding to a search radius of 3.14 particle lengths. After 32 epochs of training, the modified architecture outperformed the baseline architecture on the DPI dataset, resulting in a lower loss of 0.0635 compared to 0.0758, as shown in Table 4.7. The order of training runs remained relatively consistent, with e.g., the best run being first place after 1,500 updates and until the end.

Learning Rate. Figure 4.8a shows that an increase in learning rate is linearly related to an increase in validation loss. There is also a linearly increasing lower bound. This bound indicates that the optimizer has difficulty properly minimizing the loss function for bigger learning rates. The vertical spread increases with larger update sizes. This effect is particularly pronounced for learning rates above 0.003, where some runs fail to improve the loss below 0.9 for their entire training duration and remain practically stuck. For learning rates above 0.045, more than 50 % of models enter this failure state.

These are clear indicators of a too-high learning rate. The previous searches likely also had too big update sizes. The effect was less pronounced for them, possibly due to the window function. Additionally, ASHA partly concealed the problem. The optimal range for the learning rate is likely below 0.002. Using a window function may have helped guide the learning process for earlier runs, thus allowing higher learning rates.

The following hypothesis may explain why 14 of the 100 trials failed to improve. These runs used relatively long filter extents and high learning rates. When initializing the values of the grid of the CConvs, values are likely set relatively constant throughout the grid. This lack of differentiation is a problem when there is no window function. A window function would artificially scale down the effect of the value of specific grid cells.

Without a window function, the initial state is problematic. All values are about equally far apart from zero. The relative relationships between values given by the density gradient target data do not exist. Values, therefore, are misaligned. A learned gradient update can correct this misalignment of values by shifting respective values closer to zero or away from zero, thereby intuitively relearning the window function. Due to the symmetry property of the original SPH kernel, one expects certain symmetries to reappear in these updates.

However, a learned gradient update could ignore these symmetries. For example, it could improve average loss by pushing the values on one side of the grid toward the negative and slightly more toward the positive on the opposite side. Such an update becomes a

problem when the magnitude of the value updates is too large. The update causes a new misalignment of values in the grid. This misalignment then requires a correction in the opposite direction. If the learning rate is high, this correction will be approximately the same size or larger than the original update. It essentially undoes what was previously learned. Consequently, the learning process oscillates, and the optimizer fails to minimize, which is a typical problem with too high learning rates [Smi18].

Filter Extents. An increase in filter extents is linearly related to an increase in validation loss, ignoring the previously examined 14 outliers. Figure 4.8b shows this. This relationship may be due to the artificial cutoff introduced by the continuous convolution combined with the small number of training epochs. Two particles that are far enough apart exert no effect on each other in the context of an SPH simulation. Correctly weighting particles far enough apart as zero may be challenging without a window function. The qualitative analysis in the next chapter explores this issue further.

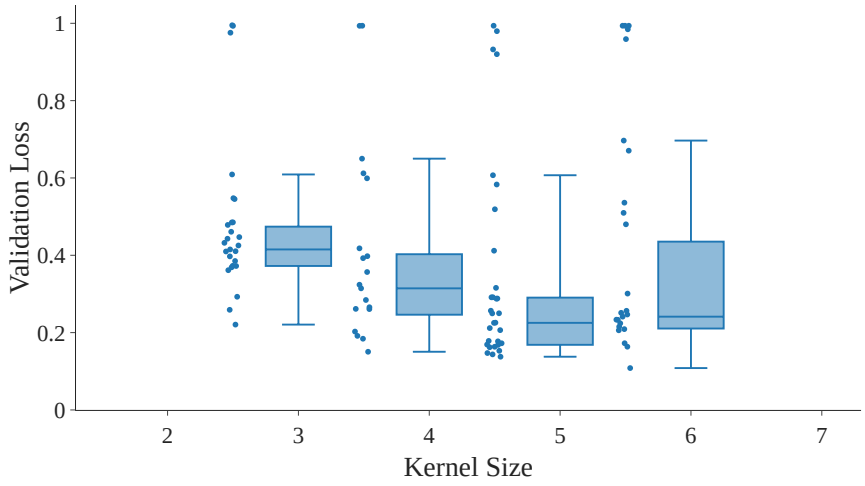


Figure 4.7.: With Bayesian Optimization, a kernel size of 5 produces the best results, closely followed by a kernel size of 6. Boxplots ignore stuck trials.

Kernel Size. Of the kernel sizes, shown in Figure 4.7, a size of 5 consistently gives the best results. Size 3 models perform the worst on average. A grid of this size is likely too low in its resolution with its three values for any direction. Sizes 4 and 6 yield low loss values but less consistently. Their lack of a central grid element could explain this lack of consistency. It may make it more challenging to represent the underlying smooth fluid dynamics. One can imagine a smooth curve, zero for $x = 0$ and $x \geq \kappa h$, but not in between. Such a curve is similar to the derivative of the SPH kernel when moving only in a single direction, which the density gradient formulas use. When no separate grid cell exists for $x = 0$, properly discretizing such a curve becomes challenging. Consequently, other odd sizes, such as 7, 9, 11, ..., could also yield positive results. Their main drawback is the increased model size.

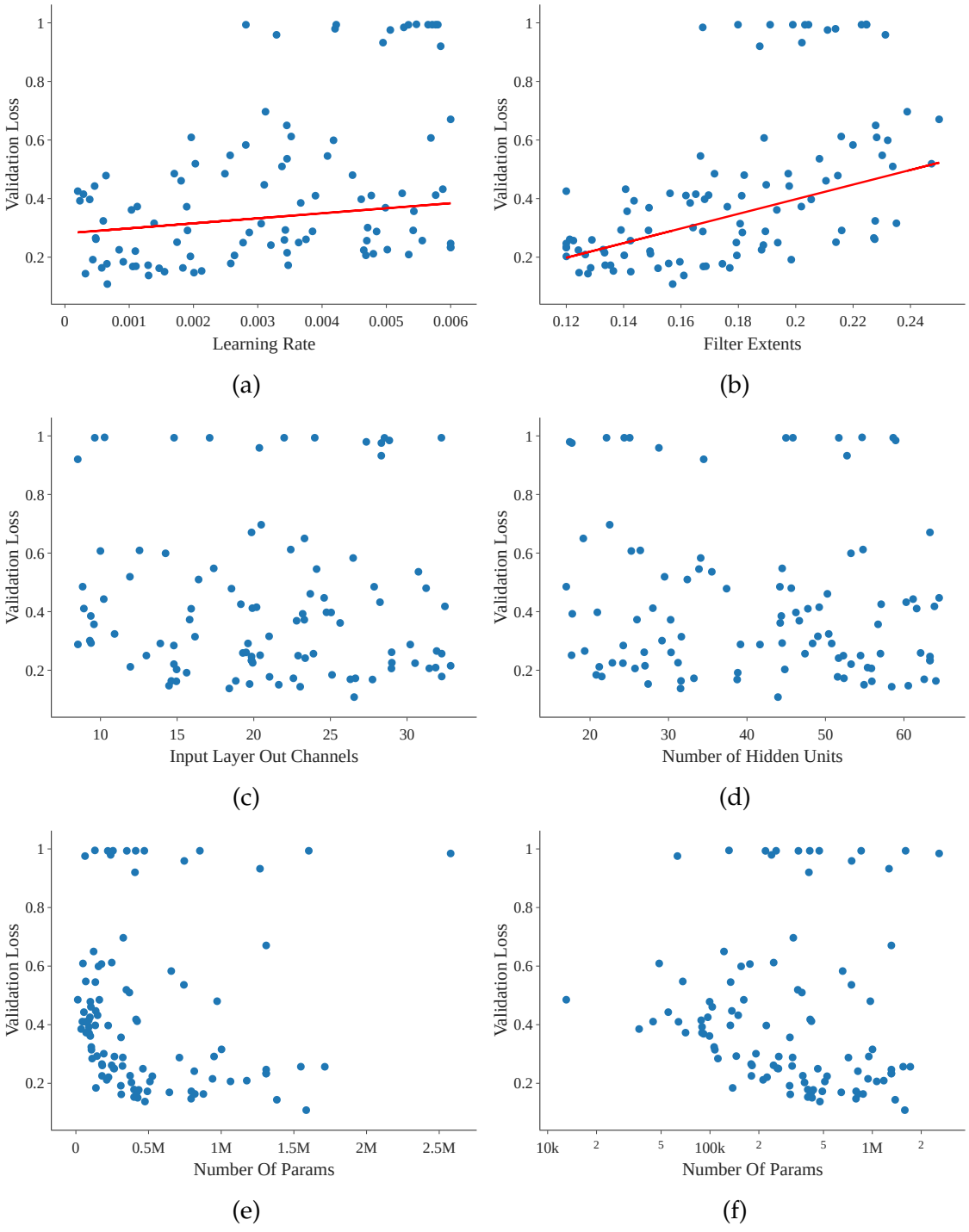


Figure 4.8.: The results of the third, adaptive search. Regression lines ignore outliers. Figure f log scales the results of e.

Hidden Dimensions. The number of output channels of the input layer and channels of the hidden layers plots show no clear pattern for the loss values. Less than 15 channels for the input layer and 30 channels in the hidden layers introduce a lower bound for the validation loss. Below that, the representational power of the feature vectors is not sufficient for the underlying physical dynamics. Other parameters are more important for the performance.

Message Passing. Deeper networks with more rounds of message passing perform better, as Figure 4.9 shows. The increased particle communication may help the network learn the density gradient data. Alternatively, more layers add more parameters, which, as we see next, is also beneficial for performance. The best run used three hidden layers, corresponding to five rounds of message passing. However, more message passing steps are associated with more non-improving trials. This increase in failed runs may be related to the issue of features converging closer together for more message passing rounds. Alternatively, problems with the loss gradient of the deeper networks, such as vanishing or exploding gradients, could also be an explanation. Figure 4.10 retrospectively examines the message passing for the previous, second search. The results are similar.

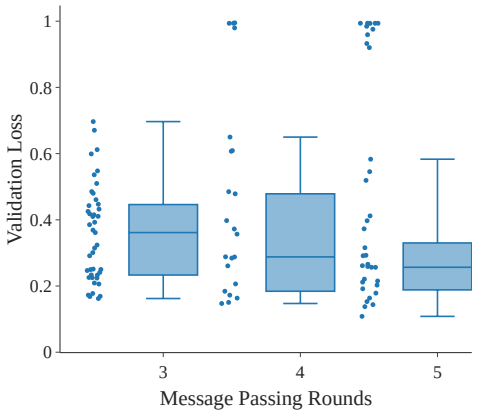


Figure 4.9.: Validation loss and message passing rounds. Boxplots ignore stuck trials.

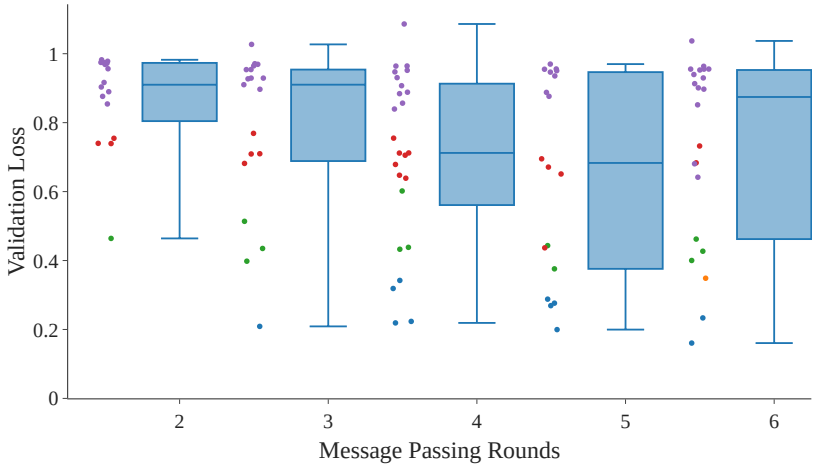


Figure 4.10.: Validation loss and message passing rounds of the previous, second search.

Number of parameters. The goal was to find a lightweight model. 85 % of the examined models had less than 500,000 parameters, as seen in Figure 4.8. Despite this, the best model is found within the top 5 % quantile of the number of parameters. It is 1.6 million parameters heavy. The search only evaluated two other models with more parameters. The high number supports the previous hypothesis that the parameter count improves with higher values and is a key performance factor. The baseline architecture had 686,658 parameters. It performed almost as well, making it a considerable outlier in this and the previous search.

Table 4.7 concludes this chapter. It shows the discovered, refined hyperparameter configuration compared to the baseline model it outperforms. The next chapter investigates possible reasons for this.

	Parameter	**Refined**	**Baseline**
1	Learning rate	0.00066	0.002
2	Batch size	2	2
3	Optimizer	Adam	Adam
4	Hidden layers	3	2
5	Dimensions	$[26 \cdot 3, 43, 43, 43, 1]$	$[32 \cdot 3, 64, 64, 1]$
6	Activation function	`ReLU`	`ReLU`
7	Kernel size	$[6, 6, 6]$	$[4, 4, 4]$
8	Filter Extent	0.157098	0.225
9	Interpolation	`trilinear`	`trilinear`
10	Window function	`None`	`Poly6`
11	Coordinate mapping	`identity`	`ball_to_cube`
12	Ignore query points	`False`	`True`
13	Dense Layer	`True`	`False`
14	Align corners	`True`	`True`
15	Normalize	`False`	`False`
	Trainable parameters	**1,585,011**	**686,658**
	Validation loss 32k	**0.0635**	**0.0758**

Table 4.7.: The final selected hyperparameters and their default values given by the reference design.

Chapter 5

Evaluation

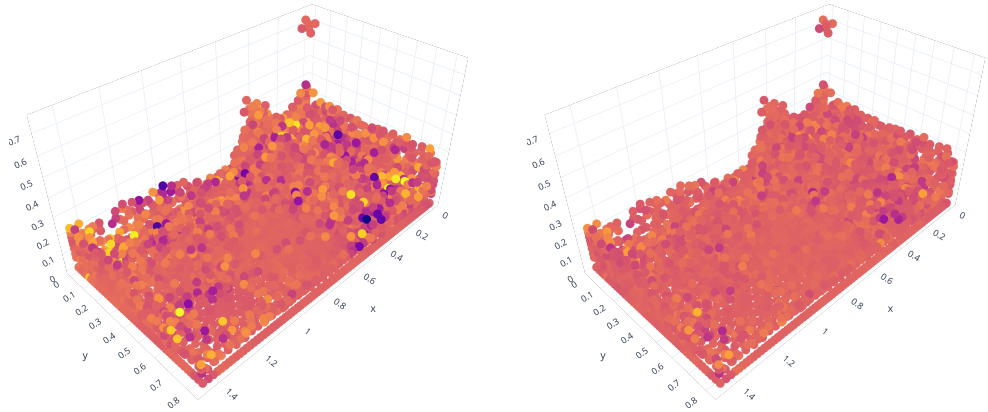
This chapter explores why specific changes in hyperparameters lead to improved performance. We examine behavioral similarities and differences between the baseline and the refined model. Identifying these gives us insight into how previously identified performance-critical components influence performance. It may also provide ideas for further avenues of research. Section 5.1 qualitatively evaluates both models by visually assessing their performance and analyzing learned responses in the context of the SPH method. In Section 5.2, we understand the models on a quantitative, more macroscopic level by analyzing the density gradient distributions they both produce. This knowledge helps us to assess whether the MSE is an appropriate metric for the training and performance of SPH-focused GNNs.

5.1 Qualitative Analysis

5.1.1 Visual Assessment

3D Assessment. We evaluate the refined architecture of the last chapter. We generate three-dimensional scatter plots of the samples to assess whether the model successfully reproduces the general trends of the underlying density gradient data. The plots apply a relative and continuous color scheme to the quantities of the particles. We select samples previously unseen by the network from the DPI dataset. The selection of a sample is partly random and partly based on its representativeness to the model behavior.

The new model replicates general trends of the density gradient data. Figure 5.1 shows an example. Distinguishing between model output and target data is visually challenging. Color transitions reappear in the predictions. Areas in the fluid where the target data takes on a particular value and color nearly always correspond to similar values and colors in the predicted model. Samples where the particles are in equilibrium and the normalized density gradient is zero are an exception. In these samples, the predicted values are close to zero but not exactly zero, which the relative color scheme overly emphasizes. Otherwise, no unique patterns appear that one could use to consistently distinguish model prediction from the actual target.



(a) Model prediction **(b)** Model-to-target delta

Figure 5.1.: Predicted temporal density gradient of the refined network

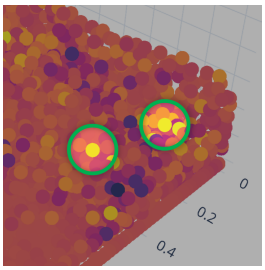


Figure 5.2.: Marked outliers

Most deviations from the target density gradient are minor. When significant outliers appear, they typically only affect individual particles. Outliers were identified visually, for example, by directly comparing prediction plots like the one in Figure 5.1a with the target or by generating prediction-to-target delta plots such as Figure 5.1b. Figure 5.2 highlights the two most extreme mispredictions. The upper-right particle is an uncharacteristic overprediction of a non-outlier target data value by 1.99 as 4.30. Often, mispredicted values are already extreme relative to the normalized density gradient distribution. A similar example is the lower bright yellow particle with an extreme target value of 5.72. The refined model mispredicts it as an even extremer 7.84.

These errors are not unique to our modified model. We do not show the model-to-model delta plot for Figure 5.1 because it is almost monochromatic. Near-monochromatic in a relative color scheme indicates that most values are almost identical compared to some highly distinct values in the data. The plot, however, scales its values on the microscale with 10^{-6}. Value differences between models, therefore, are minimal. For the above-mentioned most extreme error of the previous sample, the refined model's prediction only differs by $9.54 \cdot 10^{-7}$ from the baseline. Both models are similarly highly wrong about the particle.

2D Assessment. We map the particles and their density gradient values onto a two-dimensional grid of values. This representation discards positional data but has advantages. It simultaneously displays all particles and their values, even those in a three-dimensional view, hidden behind other particles. This rearrangement of particles on a two-dimensional grid makes it easier to compare individual particles in terms of their target and predicted

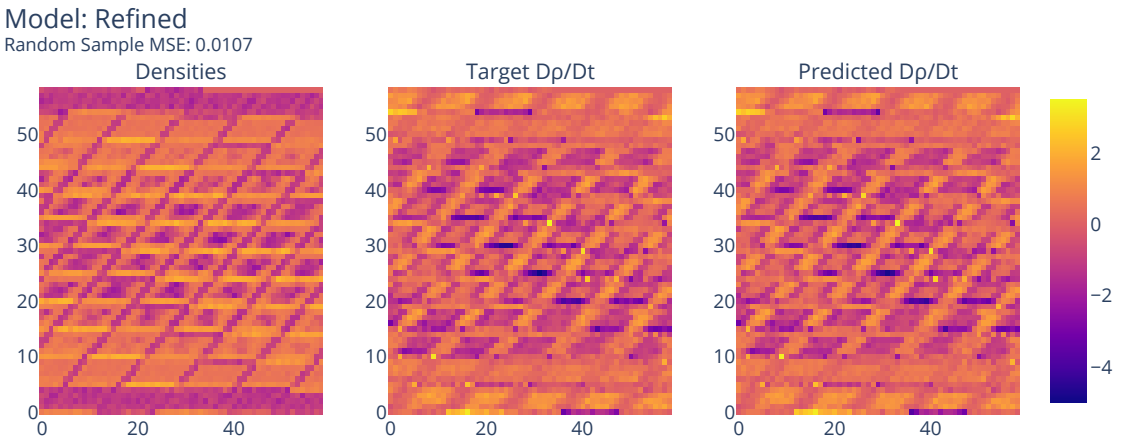


Figure 5.3.: Two-dimensional representation of a sample.

values. Underlying patterns in the fluid target data are transformed and reappear in the plot.

The two-dimensional view of the values confirms the results of the three-dimensional analysis. The network successfully reproduces general trends and patterns. Differences for individual particles now appear more pronounced. There is a slight attenuation of values and their colors. For example, dark values in the target often appear lighter in the prediction, and light colors appear darker. These differences appear extremer for some particles. The view does not disclose relational information, e.g., whether those particles are near the boundary. These observations confirm the quantitative analysis findings that the density gradient gets well replicated with mostly minor errors.

5.1.2 Learned Response

Visualization of the kernel or its gradient is challenging because no SPH kernel is involved in the network's output. As a surrogate, the learned temporal density gradient is analyzed under certain conditions and compared with the analytical solution. We place two particles at a distance x and with opposite velocity vectors $\mathbf{v}$. For each particle distance x, we calculate the analytical solution to the temporal density gradient and the model's prediction. The resulting plot of the analytical solution closely resembles a scaled version of the derivative of the cubic spline kernel, as expected with Equation D2. Figure 5.4 shows it. The learned output of the baseline configured architecture is in Figure 5.5. Figure 5.6 shows the version our 1.6 million-parameter network learned.

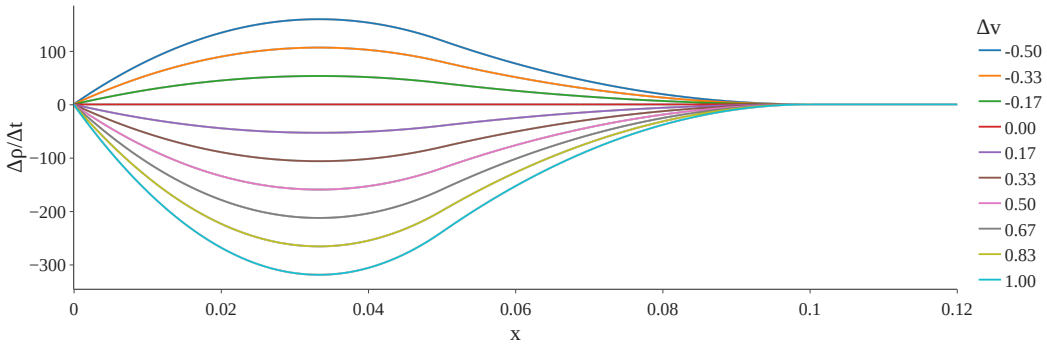


Figure 5.4.: The analytically calculated value of the temporal change in density.

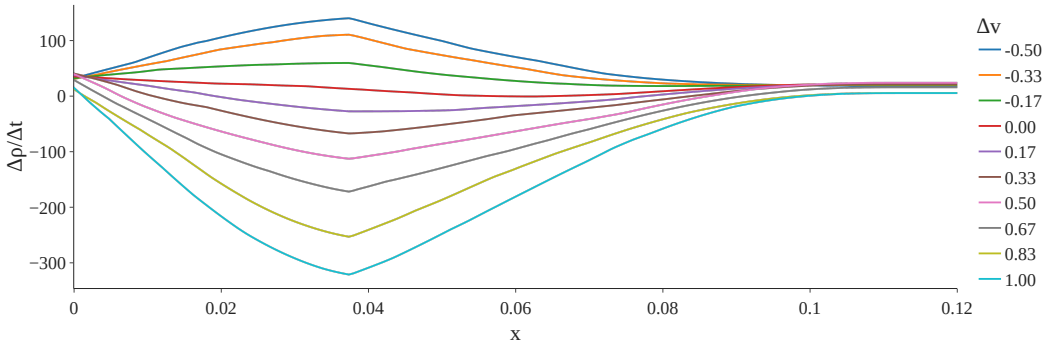


Figure 5.5.: The predicted change in density learned by the baseline network.

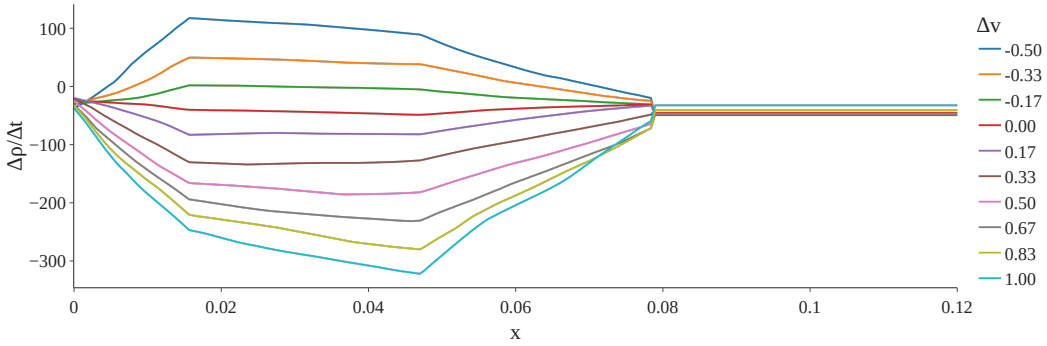


Figure 5.6.: The predicted change in density learned by the refined network.

Baseline Model. We first compare the predictions of the baseline model to the analytical solutions in Figure 5.5. The output captures the general shape of the analytical solution well. It is very smooth due to the forced smoothing of the `Poly6` window function. It is jagged and shows a triangular peak in the output at a distance of $x = 0.39$. The location of the peak is consistent for all speed differences tested. It is close to the peak position of the analytical solution but does not match it perfectly. The equidistant spacing between the curves is approximately, but not perfectly, reproduced. Therefore, a change in particle velocity is not precisely linearly related to a change in the density gradient.

The analytical gradient vertically centers at zero for $x = 0$. It approaches zero for $x > 0.1$. The learned solution adds a small positive bias to the y-axis for $x = 0$. It adds a slightly different one for $x > 0.1$. This bias varies for different speed differences. The model's output for the red, $|\Delta \mathbf{v}| = 0$ curve is not zero as expected with Equation D2. Instead, it is assigned small, non-constant values greater than zero.

Subsection 2.2.2 discussed properties of SPH kernels, which influence simulation behavior. The model does not use such a kernel, as mentioned. An SPH smoothing function can be reverse-engineered from the output in Figure 5.5 for Equation D2. The curve it shows describes a linearly transformed section of the derivative of the SPH smoothing function. From this derivative, we infer properties such a smoothing function would satisfy.

For most desirable properties, it is debatable whether they are satisfied. Such a kernel does not satisfy the precise definitions in Subsection 2.2.2, such as the unit property, as they give no space for errors. One can weaken these definitions to introduce a concept of approximation, for which the concrete formalization is beyond our scope. Some of these weakened properties are satisfied, depending on the definition. For example, the kernel can be approximately smooth, unit, symmetric, and decaying with increasing particle distances. The compact support property is more difficult to decide. The continuous convolutions have a cutoff implemented by their filter extents, outside of which no particles are found. However, there is no distance x beyond which the network outputs zero.

Refined Model. The function learned by the refined, windowless network shares many characteristics with the windowed baseline. It is nearly evenly spaced, spiky, and not vertically aligned with zero. However, it is not smooth. Its shape approximately follows that of a piecewise linear function. Individual piece segments display a small degree of smoothness. Each predicted curve has two peaks at about $x = 0.016$ and $x = 0.047$. The peaks have different y-values for differing speed differences. They do not match the location of the peak of the analytic function. The shape of the curve shows three identifiable edges, ignoring the cutoff at $x = 0.08$.

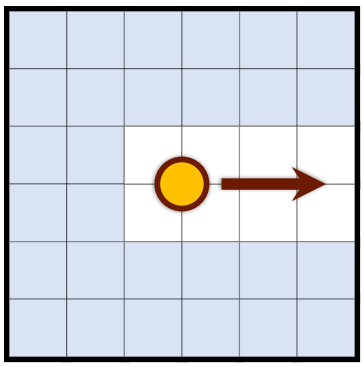


Figure 5.7.: Two-dimensional view of particle moving along the grid.

Figure 5.7 illustrates why the structure of the $6 \times 6 \times 6$ grid pattern reemerges in the output. We consider the convolution operation from the location of the stationary particle. The second particle moves along a single dimension, with its trajectory starting at the center where $x = 0$. Along its way, the convolution obtains only values of the marked cells. The CConv layers linearly interpolate values, resulting in hard edges when the process shifts to the next set of accessed cells.

The cutoff is exactly half the length of the filter extents of 0.16. They gave the diameter for the particle-weise neighbor search. The cutoff point was optimized during HPO. Reducing filter lengths and, thereby, the cutoff point linearly reduced validation loss. A shorter cutoff means the same number of linear parts must approximate less of the analytic smooth curve. The artificial cutoff may also be closer to the solution than the linear reconstruction of Figure 5.6. The approximation quality improves as the problem becomes simpler. Consequently, predictions improve, and loss values decrease. The spiky approximation describes the target curve better. The curve's shape causes it to overpredict in some cases and underpredict in others, on average attaining better results. This behavior also compensates for the fewer neighboring particles found with the shorter search distance. Alternatively, the model outperforms the baseline because the smoothing window function of the baseline is slightly misaligned. It causes the predicted curve to slightly but consistently mismatch the target curve.

5.2 Quantative analysis

The refined model developed in the previous chapter performs with an MSE of 0.0628. This section explores the meaning that this number encodes by conducting a quantitative analysis of the data produced by the network. It also provides additional information about

the behavior of the GNN on a macro-scale. We compare the distribution of the predicted gradient values to the distribution of analytical values. Any systematic misbehavior of the network, such as over- or under-prediction or attenuation of extreme values, will become apparent. This analysis uses normalized and for the model novel samples from the DPI dataset.

5.2.1 Distributions

Figure 5.8 shows a representative density gradient histogram and the corresponding model predictions. The model duplicates the normalized mean of 0 and the standard deviation close to 1. It successfully reproduces the shape and skew of the histogram. No systematic tendency to over- or underpredict values is apparent. The GNN has likely found a method of computing the gradient that consists of more than memorization. It successfully replicates distributions even for challenging scenarios, an example of which is shown in the appendix in Figure A.2.

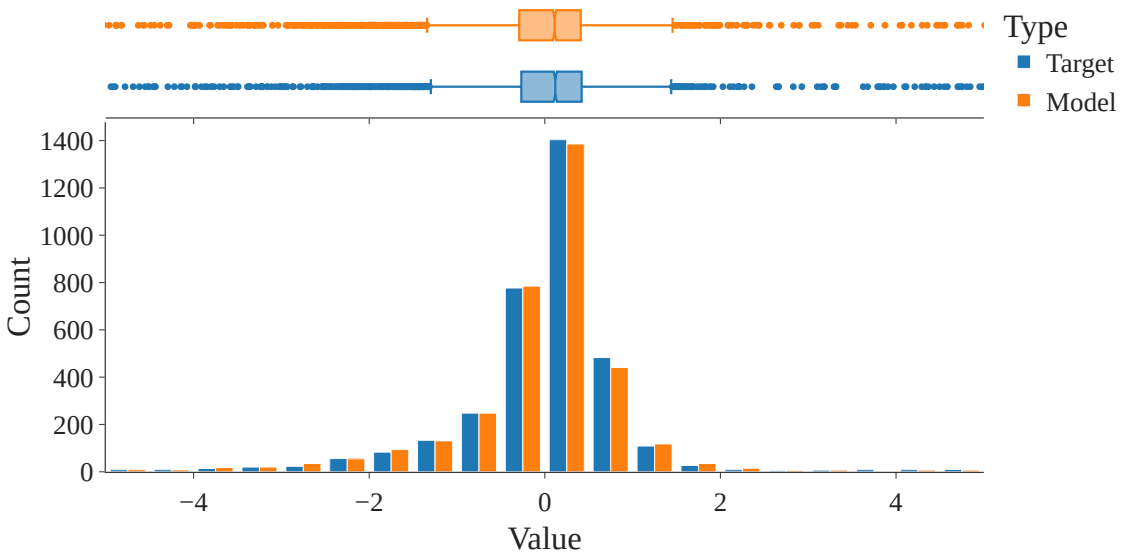


Figure 5.8.: The figure shows a typical density gradient histogram.

Noticeable differences between predicted and analytical values are rare and often similar to Figure 5.9. In this example, general features of the distribution are again reproduced. There is a slight positive overprediction of about 15 % of the values near zero. This misbehavior may not cause a simulation using the neural network to become unstable, although an actual simulation is needed to test this.

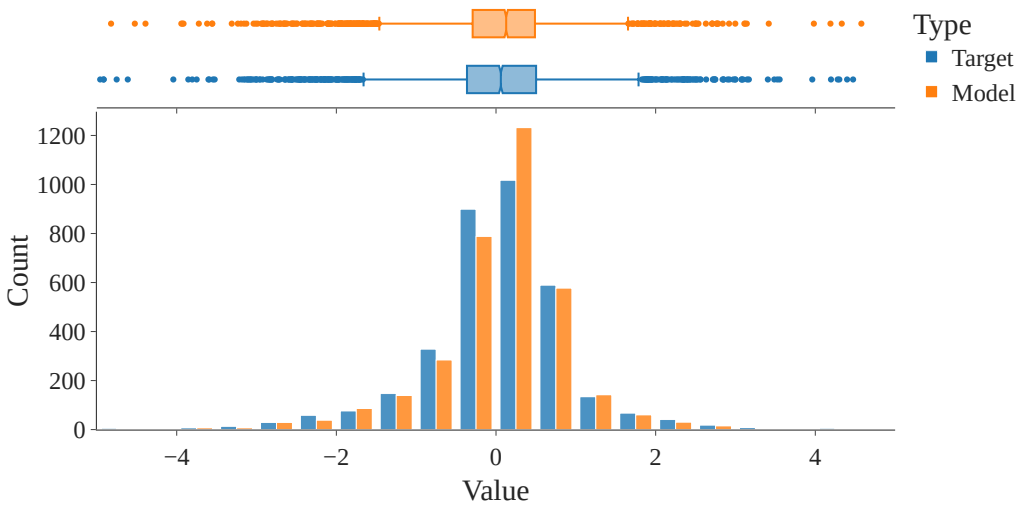


Figure 5.9.: A typical density gradient histogram with minor errors.

A typical distribution of errors, defined by the deviation from the target, is shown in Figure 5.10. The shape of the distribution is approximately normal and centered at zero. The plot shows that about 85 % of the deviations in the predictions are between -0.35 and 0.35. About 95 % are minor errors between -0.45 and 0.45. The numbers differ slightly for different simulation frames, as we see later.

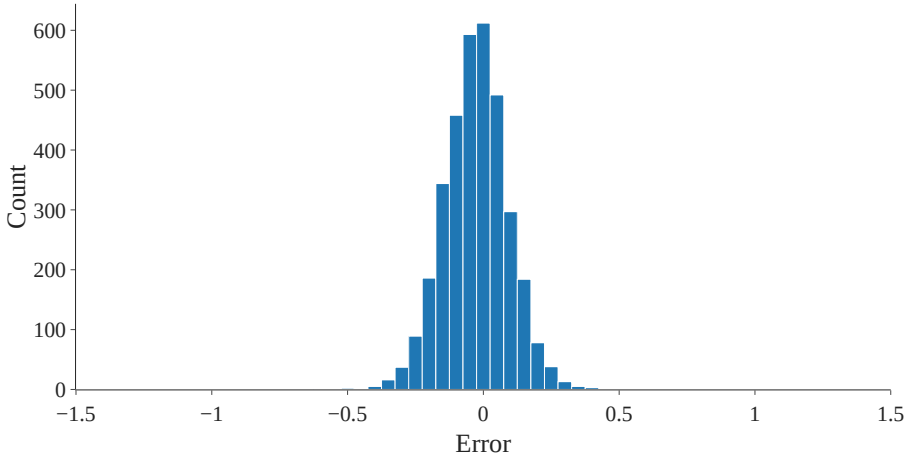


Figure 5.10.: A typical distribution of errors in the prediction. The density gradient data is normalized to a standard deviation of 1.

5.2.2 Rollout Error

In a conventional rollout test, such as the one performed by Sanchez-Gonzalez et al. [San+20], one examines the simulation's behavior over multiple time steps to assess whether the network accurately replicates physical phenomena. We have no simulator. Instead, we analyze the MSE loss computed over each successive sample of an unseen, contiguous simulation. This metric quantifies how much the networks' outputs deviate from the analytical solution. We compare the refined network to the baseline model and see the differences per frame.

An example simulation unroll is in Figure 5.11. The loss shows similar behavior for both the baseline and improved models throughout the simulation. The MSE lines for both models are equally unsmooth despite the previously observed much smoother output of the baseline model. Both models start with a low loss value below 0.02. In the initial stage of the simulation, particles are arranged in a grid and have yet to gain much velocity due to gravity. Until frame 60, both models produce nearly identical MSEs. After this point, the simulation encounters more complex scenarios where the fluid collides with the boundaries, resulting in droplet formation. The refined model outperforms the baseline model in these situations, with the baseline MSE acting almost as an upper bound. These cases contribute to the reduced loss of the refined model relative to the baseline model.

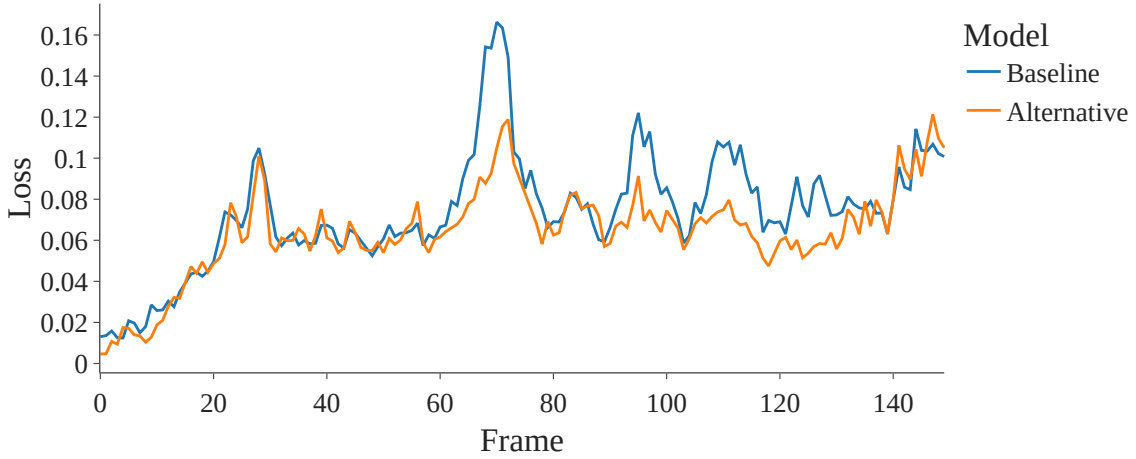
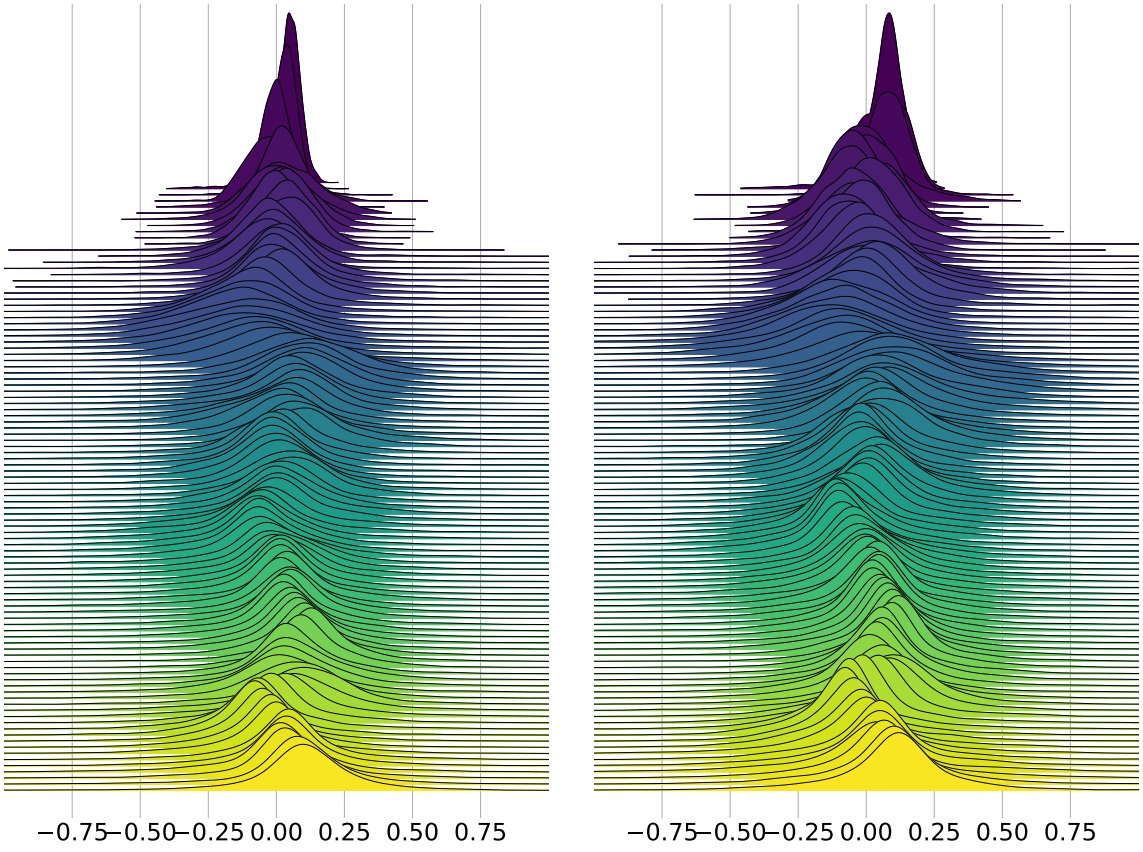


Figure 5.11.: The loss per frame of a contiguous simulation. The new model outperforms the baseline model in cases where its loss values are especially high.

We take the first 100 frames of the contiguous simulation from above and plot the model prediction error distributions per frame, starting at the top with the first frame. This stacking of distributions gives us a ridge plot for each, the baseline, and the refined model, shown in Figure 5.12. The MSE compresses each error distribution into a single number. These new plots contain the uncompressed and partially lost information of the 200 error

distributions of Figure 5.11. A narrow, tall error distribution indicates a close match between the target gradient and the low loss value. A wider distribution indicates a poorer fit with more extreme errors. A distribution with a non-zero peak corresponds to a positive or negative misprediction for most values in a frame. Due to the mismatch in the target density gradient, these frames have higher MSEs.



(a) The refined model's errors

(b) The baseline model's errors

Figure 5.12.: The error distributions of the first 100 frames of the contiguous simulation of Figure 5.11. The ordering is from the first frame (Top) to the last frame (Bot).

Both models behave similarly. The generated error distributions are unimodal and appear to be normally distributed. Both model's outputs have approximately the same mean and variance for the same frames. For example, yellow frames are positively overpredicted, and dark blue frames are negatively underpredicted, regardless of the hyperparameter configuration. The refined model shows some slightly narrower error distributions, especially for frames where the baseline model performs poorly, e.g., for the dark blue-colored distributions. This similarity is despite the very different configuration of the refined model, suggesting a more structural origin within the architecture. Both

GNNs tend to over- and underpredict for a few frames alternatingly, as can be seen by observing the movement of the peaks of the distributions between frames. Due to the consistency of these fluctuations, a frame with underpredicted gradient values follows one with overpredicted values and vice versa. This oscillation could result in an additional, mild error correction mechanism, as some mispredictions are reversed by an error in the opposite direction. An actual simulation is needed to test this hypothesis.

The MSE balances these oscillations with their means around zero. It incentivizes error distributions to move closer to zero, as this corresponds to fewer extreme errors and more minor errors. Any overall tendency to deviate leads to more errors and a higher MSE. The final loss value one observes after training does not provide information about error distribution shapes or value fluctuations. We, nevertheless, conclude that the MSE in an appropriate performance metric. Reducing it improves all of the above-addressed misbehaviors.

Chapter 6

Conclusion

6.1 Discussion

This work investigated the learning of spatio-temporal density derivatives in Smoothed Particle Hydrodynamics using GNNs, for which we chose ConvNet. We consider it a problem representative of other problems in particle-based fluid simulations. A vital component of this architecture is the continuous convolution. We studied the impact of changes to its behavior and other hyperparameters on the training procedure and network performance.

We recommend starting with a high learning rate of 0.0025 for short training times. In an HPO scenario, it should be combined with SHA to eliminate non-converging and non-promising trials. A stepwise learning rate decay or batch size increase schedule [Smi+18] likely helps to find high-quality models by slowing the parameter descent once the learning process is near good minima in the loss landscape. Due to the noisy gradient approximation, the learning process does not get stuck on unfortunate trajectories to suboptimal optima. The initial configuration of weights and biases can cause statistically but practically irrelevant performance differences. A single additional training epoch will often overcorrect any possible suboptimal initialization bias.

We found the MSE to be a good measure of model performance. A low value corresponds to a good fit of the density gradient distribution with rare extreme errors. While it does not capture whether the model is over- or underpredicting for a single input, it describes current performance sufficiently well. Zeroing out the MSE is a good approach to ensure high-quality output. The MSE offsets any negative behavioral tendencies, as they increase it on average.

The kernel size and the correlated number of parameters were identified as the most important predictors of the potential performance of a ConvNet in hyperparameter optimization. Subsequent analysis of the network's learned density gradient revealed that this is primarily due to the linearly interpolated grid structure of the kernel. Learned features are too low in resolution and linearly interpolated. The model's learned response ends up being approximately piecewise linear. However, physical relationships involved in a fluid continuum scenario, as described by the Navier-Stokes equations, are all continuous and smooth.

Window functions address this issue by adding smoothness retroactively, not at a feature level but at the response level. These functions have several characteristics, such as their general shape and curvature. All of these can be misaligned or skewed. We saw, for example, how the `Gaussian` outperformed the `Poly6` window function during HPO. The former matches the cubic spline smoothing function more closely, which was used to generate the density gradient target data. Window functions cannot be learned. The work showed that bigger networks can adapt quite successfully, albeit with room for improvement in optimizing the results.

6.2 Further Research

We propose deprecating the window function for an alternative, learnable solution operating at the feature level. For example, the CConv kernel could encode splines or Bezier curves instead of linearly interpolated values. Depending on the implementation, this would allow learning the characteristics of smoother curves. These, in turn, can better describe the underlying physical relationships. There is already promising research in this area with spline convolutions [Fey+18] and Fourier-based approaches instead of linear functions. However, many questions about their applicability in the SPH context still need to be answered.

One problem that arises from a higher feature resolution or grid size is an increase in the number of parameters of such a network. Luckily, many issues related to SPH and fluid dynamics, such as water and air flows, display a degree of symmetry. One would, for example, expect a simulation to behave the same regardless of how all particles are rotated around the axis of gravity. With the current architecture, this is not the case. It first has to learn a notion of symmetry. This relearning of features for each spatial dimension makes the training and use of some parameters inefficient and error-prone. Information is duplicated but imperfectly due to the approximative nature of neural networks. A solution that considers these symmetries could be more efficient and predictable. Prantl et al. [Pra+22] already show a promising approach, where they enforce this symmetry directly in the grid of the continuous convolution.

There are other possible ways to modify the ConvNet architecture that this work did not explore. It is open whether the order of the layers is optimal. There may be better ways to introduce information into the network than concatenating features in the first layer. For example, one could add features slowly, step by step. An additional encoding and decoding layer analogous to Sanchez-Gonzalez et al. [San+20] could be added to learn a more efficient latent space graph representation. Redundant, irrelevant, or suboptimally represented information introduces noise into the learning process. This noise potentially

affects the model's performance. Consequently, the feature selection should be examined. One may achieve more physically stable networks by pre-training the networks on highly corrupted data and later fine-tuning them on more precise data. Networks might learn an error correction mechanism that minimizes the risk of encountering undefined states during simulation. There are other GNN architectures. The Graph Isomorphism Network (GIN) [Xu+19] architecture promises potential in the SPH context. It was designed to be simple while providing maximal expressive power for graph components [Xu+19].

Many past deep learning advances resulted from astutely and pragmatically leveraging the problem's underlying structure. An example is the traditional convolutional layer for pixel-structured data instead of a fully connected layer. It has the structure that otherwise must be learned already built in. This constraint drastically limits the problem space. Thereby, it reduces training times and improves delivered performance. For analogous reasons, desirable solutions for fluid simulations should exploit common, shared elements found in viscous flows, like smoothness, and enforce them at a structural level.

6.3 From Particles to Patterns

Currently, many particle-based simulators solve the fluid problem in an atomic and recipe-like manner. They calculate quantities and acting forces for all individual particles for each of the thousands to millions of time steps they run. In each step, they apply a set of simple rules from which, over time, phenomena emerge that believably mirror physical reality. As we saw with the density derivative problem, a deep learning model can learn each of these rules individually.

Neural network-based models, however, can operate on a more general level. In recent years, they have consistently surprised their creators by learning to solve problems beyond the capabilities of humans and their algorithms. They learn relationships and optimizations, otherwise hidden or incomprehensible. They, therefore, have potential to unfold in tasks more challenging than these more straightforward problems.

Predictions in the SPH context do not have to be limited to the individual particle or timestep level. Networks given access to the temporal dimension may learn time-based trends within the data. They may find long-term strategies that allow for stable rollouts over long periods. With even deeper access to more information, networks can learn more general fluid phenomena such as complex vortices and flow paths. At the end of this process, given the fluid and the scenario, they can unfold the entire development of the liquid. While currently, such generative, end-to-end predicted simulations seem out of reach, they would follow other breakthroughs in the field that just five years ago were considered unimaginable.

In this work, we gained insight into hyperparameters of the ConvNet architecture and their impact on performance on the density gradient problem. We argue that observed relationships between parameters and predictions suggest potential for improvement lying at the structural level of the architecture. Further, we conjecture that this potential generalizes to both other SPH problems and GNN architectures. Hopefully, these findings provide a hint at a next step in the pursuit of more advanced, neural network-driven fluid simulators.

Abbreviations and Lists

Abbreviations

List of Figures

List of Tables

Bibliography

[Ame18] E. Ameisen. *Always start with a stupid model, no exceptions.* Medium. Mar. 6, 2018. URL: https://blog.insightdatascience.com/always-start-with-a-stupid-model-no-exceptions-3a22314b9aaa (visited on 09/14/2023).

[Ash19] B. Ashwath. *Answer to "What is a latent space?"* Cross Validated. Dec. 27, 2019. URL: https://stats.stackexchange.com/a/442360 (visited on 09/22/2023).

[Bat+18] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. *Relational inductive biases, deep learning, and graph networks*. Oct. 17, 2018. arXiv: 1806.01261[cs,stat].

[BB12] J. Bergstra and Y. Bengio. *Random Search for Hyper-Parameter Optimization*. 2012.

[Ben+17] J. Bender et al. *SPlisHSPlasH Library*. 2017. URL: https://github.com/InteractiveComputerGraphics/SPlisHSPlasH.

[Bis+21] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer. *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges*. Nov. 24, 2021. arXiv: 2107.05847[cs,stat].

[Bis95] C. M. Bishop. *Neural networks for pattern recognition*. Oxford : New York: Clarendon Press ; Oxford University Press, 1995. 482 pp. ISBN: 978-0-19-853849-3 978-0-19-853864-6.

[BK15] J. Bender and D. Koschier. "Divergence-free smoothed particle hydrodynamics." In: *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. SCA '15: The ACM SIGGRAPH / Eurographics Symposium on Computer Animation. Los Angeles California: ACM, Aug. 7, 2015, pp. 147–155. ISBN: 978-1-4503-3496-9. DOI: 10.1145/2786784.2786796.

[Bor+22] J. Borrow, M. Schaller, R. G. Bower, and J. Schaye. "Sphenix: Smoothed Particle Hydrodynamics for the next generation of galaxy formation simulations." In: *Monthly Notices of the Royal Astronomical Society* 511.2 (Feb. 12, 2022), pp. 2367–2389. ISSN: 0035-8711, 1365-2966. DOI: 10.1093/mnras/stab3166. arXiv: 2012.03974[astro-ph].

[Cab+17] A. Caballero, W. Mao, L. Liang, J. Oshinski, C. Primiano, R. McKay, S. Kodali, and W. Sun. "Modeling left ventricular blood flow using smoothed particle hydrodynamics." In: *Cardiovascular engineering and technology* 8.4 (Dec. 2017), pp. 465–479. ISSN: 1869-408X. DOI: 10.1007/s13239-017-0324-z.

[Cao+23] Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. S. Yu, and L. Sun. *A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT*. Mar. 7, 2023. arXiv: 2303.04226[cs].

[Cmg16] Cmglee. *Visual comparison of convolution, cross-correlation and autocorrelation.* 2016. URL: https://commons.wikimedia.org/wiki/File:Comparison_convolution_correlation.svg (visited on 09/22/2023).

[Den68] P. J. Denning. "Thrashing: its causes and prevention." In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*. the December 9-11, 1968, fall joint computer conference, part I. San Francisco, California: ACM Press, 1968, p. 915. DOI: 10.1145/1476589.1476705.

[DM88] J. H. Dymond and R. Malhotra. "The Tait equation: 100 years on." In: *International Journal of Thermophysics* 9 (Nov. 1, 1988). ADS Bibcode: 1988IJT.....9..941D, pp. 941–951. ISSN: 0195-928X. DOI: 10.1007/BF01133262.

[Fal19] W. Falcon. *PyTorch Lightning*. Mar. 30, 2019. URL: https://www.pytorchlightning.ai.

[Fey+18] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. *SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels*. May 23, 2018. arXiv: 1711.08920[cs].

[FS07] M. Firestone and F. Shane. *Flow Visualization Course : Galleries*. 2007. URL: https://flowvis.org/OldGalleries/2007/assignment4.html (visited on 09/22/2023).

[Gao+23] C. Gao, Y. Zheng, N. Li, Y. Li, Y. Qin, J. Piao, Y. Quan, J. Chang, D. Jin, X. He, and Y. Li. *A Survey of Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions*. Jan. 12, 2023. arXiv: 2109.12843[cs].

[GB10] X. Glorot and Y. Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016. 775 pp. ISBN: 978-0-262-03561-3.

[GM77] R. A. Gingold and J. J. Monaghan. "Smoothed particle hydrodynamics: theory and application to non-spherical stars." In: *Monthly Notices of the Royal Astronomical Society* 181.3 (Dec. 1, 1977), pp. 375–389. ISSN: 0035-8711. DOI: 10.1093/mnras/181.3.375.

[HSW89] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators." In: *Neural Networks* 2.5 (Jan. 1, 1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

[Ihm+14] M. Ihmsen, J. Orthmann, B. Solenthaler, A. Kolb, and M. Teschner. *SPH Fluids in Computer Graphics*. 2014.

[Kar19] A. Karpathy. *A Recipe for Training Neural Networks*. Apr. 25, 2019. URL: http://karpathy.github.io/2019/04/25/recipe/ (visited on 09/23/2023).

[Kar23] A. Karpathy. *Neural Networks: Zero To Hero*. 2023. URL: https://karpathy.ai/zero-to-hero.html (visited on 09/14/2023).

[KB17] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. arXiv: 1412.6980[cs].

[Kos+19] D. Koschier, J. Bender, B. Solenthaler, and M. Teschner. *SPH Techniques for the Physics Based Simulation of Fluids and Solids*. 2019.

[LeC+98] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. "Efficient BackProp." In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by G. Montavon, G. B. Orr, and K.-R. Müller. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3.

[LGE08] M.-A. Lavoie, A. Gakwaya, and M. N. Ensan. *Application of the SPH Method for Simulation of Aerospace Structures under Impact Loading*. 2008.

[LHW18] Q. Li, Z. Han, and X.-M. Wu. *Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning*. Jan. 22, 2018. arXiv: 1801.07606[cs,stat].

[Li+19] Y. Li, J. Wu, R. Tedrake, J. B. Tenenbaum, and A. Torralba. *Learning Particle Dynamics for Manipulating Rigid Bodies, Deformable Objects, and Fluids*. Apr. 17, 2019. arXiv: 1810.01566[physics,stat].

[Li+22] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects." In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12 (Dec. 2022). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 6999–7019. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2021.3084827.

[Li18] L. Li. *Massively Parallel Hyperparameter Optimization*. Machine Learning Blog | ML@CMU | Carnegie Mellon University. Section: automl. Dec. 12, 2018. URL: https://blog.ml.cmu.edu/2018/12/12/massively-parallel-hyperparameter-optimization/ (visited on 07/24/2023).

[Lia+18] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. *Tune: A Research Platform for Distributed Model Selection and Training*. July 13, 2018. DOI: 10.48550/arXiv.1807.05118. arXiv: 1807.05118[cs,stat].

[LL10] M. B. Liu and G. R. Liu. "Smoothed Particle Hydrodynamics (SPH): an Overview and Recent Developments." In: *Archives of Computational Methods in Engineering* 17.1 (Mar. 1, 2010), pp. 25–76. ISSN: 1886-1784. DOI: 10.1007/s11831-010-9040-7.

[LLL03] M. B. Liu, G. R. Liu, and K. Y. Lam. "Constructing smoothing functions in smoothed particle hydrodynamics with applications." In: *Journal of Computational and Applied Mathematics* 155.2 (June 15, 2003), pp. 263–284. ISSN: 0377-0427. DOI: 10.1016/S0377-0427(02)00869-5.

[LR15] S. M. Longshaw and B. D. Rogers. "Automotive fuel cell sloshing under temporally and spatially varying high acceleration using GPU-based Smoothed Particle Hydrodynamics (SPH)." In: *Advances in Engineering Software* 83 (May 1, 2015), pp. 31–44. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2015.01.008.

[Mac+14] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim. "Unified particle physics for real-time applications." In: *ACM Transactions on Graphics* 33.4 (July 27, 2014), 153:1–153:12. ISSN: 0730-0301. DOI: 10.1145/2601097.2601152.

[Mau+20] C. Maurel, P. Michel, J. M. Owen, R. P. Binzel, M. Bruck-Syal, and G. Libourel. "Simulations of high-velocity impacts on metal in preparation for the Psyche mission." In: *Icarus* 338 (Mar. 2020), p. 113505. ISSN: 00191035. DOI: 10.1016/j.icarus.2019.113505.

[MCG03] M. Müller, D. Charypar, and M. Gross. "Particle-based fluid simulation for interactive applications." In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. SCA '03. Goslar, DEU: Eurographics Association, July 26, 2003, pp. 154–159. ISBN: 978-1-58113-659-3.

[McL13] D. McLean. *Understanding aerodynamics: arguing from the real physics*. Chichester, West Sussex, United Kingdom: Wiley, 2013. 1 p. ISBN: 978-1-119-96751-4.

[Ndi+19] E. Ndiaye, T. Le, O. Fercoq, J. Salmon, and I. Takeuchi. *Safe Grid Search with Optimal Complexity*. May 27, 2019. arXiv: 1810.05471[cs,math,stat].

[Nog14] F. Nogueira. *Bayesian Optimization: Open source constrained global optimization tool for Python*. original-date: 2014-06-06T08:18:56Z. 2014.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[Pra+22] L. Prantl, B. Ummenhofer, V. Koltun, and N. Thuerey. *Guaranteed Conservation of Momentum for Learning Particle-based Fluid Dynamics*. 2022.

[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*. Sept. 14, 2020.

[San+21] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko. "A Gentle Introduction to Graph Neural Networks." In: *Distill* 6.9 (Sept. 2, 2021), e33. ISSN: 2476-0757. DOI: 10.23915/distill.00033.

[San22] G. Sanderson. *But what is a convolution?* Nov. 18, 2022.

[Smi+18] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le. *Don't Decay the Learning Rate, Increase the Batch Size*. Feb. 23, 2018. arXiv: 1711.00489[cs,stat].

[Smi18] L. N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. Apr. 24, 2018. arXiv: 1803.09820[cs,stat].

[Sto+20] J. M. Stokes, K. Yang, K. Swanson, W. Jin, A. Cubillos-Ruiz, N. M. Donghia, C. R. MacNair, S. French, L. A. Carfrae, Z. Bloom-Ackermann, V. M. Tran, A. Chiappino-Pepe, A. H. Badran, I. W. Andrews, E. J. Chory, G. M. Church, E. D. Brown, T. S. Jaakkola, R. Barzilay, and J. J. Collins. "A Deep Learning Approach to Antibiotic Discovery." In: *Cell* 180.4 (Feb. 2020), 688–702.e13. ISSN: 00928674. DOI: 10.1016/j.cell.2020.01.021.

[Tha+22] S. Thais, P. Calafiura, G. Chachamis, G. DeZoort, J. Duarte, S. Ganguly, M. Kagan, D. Murnane, M. S. Neubauer, and K. Terao. *Graph Neural Networks in Particle Physics: Implementations, Innovations, and Challenges*. Mar. 25, 2022. DOI: 10.48550/arXiv.2203.12852. arXiv: 2203.12852[hep-ex,physics:hep-ph].

[Umm+20] B. Ummenhofer, L. Prantl, N. Thuerey, and V. Koltun. *Lagrangian Fluid Simulation with Continuous Convolutions*. 2020.

[Wan+18] S. Wang, S. Suo, W.-C. Ma, A. Pokrovsky, and R. Urtasun. "Deep Parametric Continuous Convolutional Neural Networks." In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. June 2018, pp. 2589–2597. DOI: 10.1109/CVPR.2018.00274.

[Wei] E. W. Weisstein. *Convolution*. Publisher: Wolfram Research, Inc. URL: https://mathworld.wolfram.com/ (visited on 05/25/2023).

[Wei+18] M. Weiler, D. Koschier, M. Brand, and J. Bender. "A Physically Consistent Implicit Viscosity Solver for SPH Fluids." In: *Computer Graphics Forum* 37.2 (May 2018), pp. 145–155. ISSN: 01677055. DOI: 10.1111/cgf.13349.

[Wu+21] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. "A Comprehensive Survey on Graph Neural Networks." In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. ISSN: 2162-237X, 2162-2388. DOI: 10.1109/TNNLS.2020.2978386.

[Xu+19] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. *How Powerful are Graph Neural Networks?* Feb. 22, 2019. DOI: 10.48550/arXiv.1810.00826.

[YZ20] T. Yu and H. Zhu. *Hyper-Parameter Optimization: A Review of Algorithms and Applications*. Mar. 12, 2020. arXiv: 2003.05689[cs,stat].

[Zha+21] X.-M. Zhang, L. Liang, L. Liu, and M.-J. Tang. "Graph Neural Networks and Their Current Applications in Bioinformatics." In: *Frontiers in Genetics* 12 (2021). ISSN: 1664-8021.

[Zho18] D.-X. Zhou. *Universality of Deep Convolutional Neural Networks*. July 19, 2018. DOI: 10.48550/arXiv.1805.10769.

[ZL21] Z. Zhao and H. Liu. "Simulation of Tsunami Generation by Asteroid Impact Using Smoothed Particle Hydrodynamics Method." In: The 31st International Ocean and Polar Engineering Conference. OnePetro, June 20, 2021.

[ZPK18] Q.-Y. Zhou, J. Park, and V. Koltun. "Open3D: A Modern Library for 3D Data Processing." In: *arXiv:1801.09847* (2018).

Appendix A

Appendix

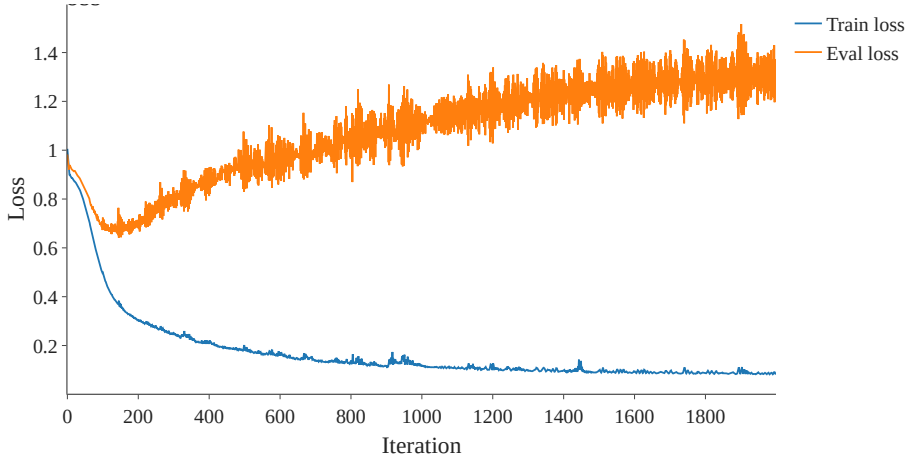


Figure A.1.: A forced overfit of a CConv on a single time frame. This test is traditionally done at early implementation stages to check functionality. In this example, from some initial experimentation, a CConv layer was trained on a single simulation time frame over 5,000 epochs. As validation data, the next timeframe was used.

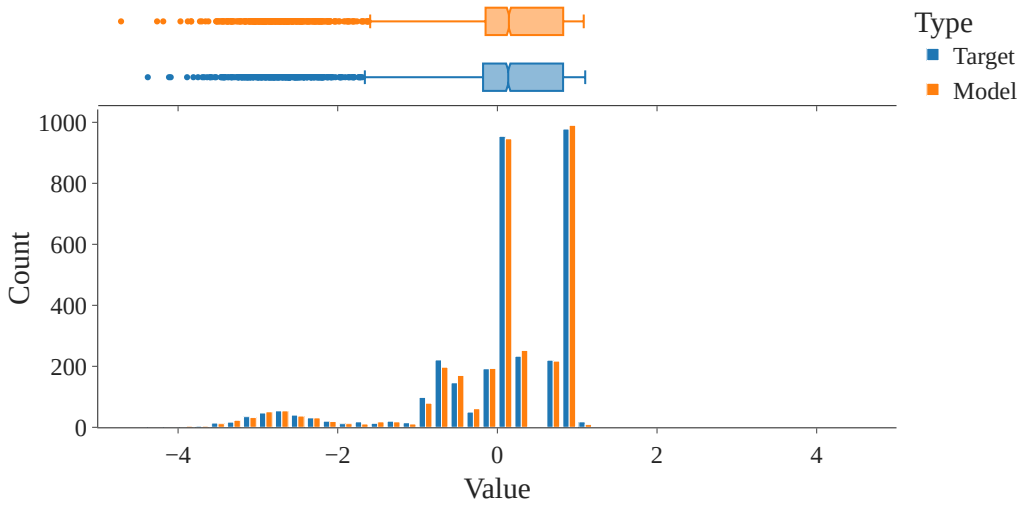


Figure A.2.: The refined network successfully reproduces a complex density gradient distribution.