DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Neural Network Driven Particle-based Fluid Simulation

Robert Brand

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Neural Network Driven Particle-based Fluid Simulation

Partikelbasierte Fluid-Simulation mit Neuronalen Netzwerken

Author:	Robert Brand
Supervisor:	Prof. Dr. Nils Thuerey
Advisor:	Kiwon Um, Ph. D.
Submission Date:	15.03.2017

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

Munich, 15.03.2017 Robert Brand

Acknowledgments

I would like to thank Prof. Dr. Nils Thuerey for providing me with the opportunity to work on this interesting topic. Secondly and especially, I want to thank Kiwon Um for putting up with my endless questions, for the weekly meetings and sometimes overlong discussions.

Also a big thanks to friends and family for supporting me during the process of creating this thesis and for providing me with feedback on ideas and drafts.

Abstract

Particle-based fluid simulations are often limited by their need for per-particle neighbor calculation. Inspired by other data-driven approaches we investigate the use of neural networks to predict the pressure forces within a fluid while avoiding neighborhood calculation. We present an approach that uses a grid to distribute the particle's quantities over space instead. Experiments show that the method can be used to believably simulate a fluid, although artifacts remain. We discuss the residual errors the approach contains and its performance compared to WCSPH. In conclusion our method proves to not be optimal in its current implementation, due to not resulting in a performance benefit. However, the method contains a lot of potential for performance improvement which might lead to it performing better than WCSPH. We therefore suggest ways this could be done as well as an alternative way of modeling the learning which predicts the entire pressure force field at once instead of each grid cell's force in isolation. The approach presented in this thesis can thus be used as a basis for further research into speeding up particle-based fluid simulations using neural networks.

Keywords: SPH, Fluid Simulation, Neural Networks, Particle-based

Contents

1 Introduction

The Smoothed Particle Hydrodynamics approach is a widely used method to simulate the movement of a fluid by discretizing its mass to a quantity of particles. To compute their movement, it is necessary to look at their positions, velocities, densities and other quantities relative to each other. The computational complexity would therefore in theory be quadratic with the number of particles. In practice, methods such as acceleration grids are used to improve this, yet it is always necessary to calculate a list of the neighbors of each particle. Large amounts of particles can thus not be simulated in real-time without extremely capable hardware, compromising the simulation quality in applications such as interactive medical simulations, video games or even film production, where preliminary simulations should not take too long to render.
Since physical correctness is often less important than overall visual believability and evaluation speed, methods could be explored that approximately predict the forces acting on a particle from data about its surroundings. Neural networks lend themselves well to this notion, as they can be used to approximate any function, given that there is a relation between the input data and the expected output values. A trained neural network can also be quickly evaluated on a Graphics Processing Unit (GPU), as it consists primarily of matrix multiplications, especially in the case of a simple fully-connected architecture.
This thesis aims to use a simple neural network to predict the pressure forces acting on each particle without explicitly calculating its neighbours. We present a grid-based approach as a substitute in chapter 4 which interpolates the particle's quantities to a grid, uses a neural network to predict each cell's pressure force and interpolates them back to the particles. The experiments in chapter 5 show that the method can be used to simulate a fluid in different scenarios, albeit with remaining artifacts and errors. We discuss the performance and residual flaws of our method. Though the approach does not result in the desired performance improvement, this is mainly down to implementation details and we show various ways this may be improved. In the end, we think an approach that predicts the entire pressure force field at once would be a promising direction to further improve the method described in this thesis.

2 Related Work

At the time of writing the published research record on the combination of machine learning and fluid simulation in computer graphics is sparce. In this section we highlight two earlier publications in particular that have a strong relation to this thesis.

2.1 Data-driven Fluid Simulation Using Regression Forests

In their 2015 paper, Ladicky et al. proposed a data-driven approach to particle-based fluid simulation [Lad+15]. They formulated the simulation as a regression problem, training a regression forest to predict the acceleration of each particle. For this, they designed an input feature vector based on so-called integral volumes. These essentially calculate features such as density cumulatively for a discretized space, i.e. grid. This means that at each grid cell, the integral volume of such a feature is based on the combined area of the cells to its bottom left. This can then be used to compute the feature of any box within the grid in constant time. The feature vector of each particle is comprised of the integral features calculated on a large fixed randomly sampled set of boxes placed relatively to the particle. From it, the trained regression forest can then predict the particle's acceleration.

In many ways, our approach is similar to and inspired by this work. Both methods avoid explicit calculation of nearest neighbors by making use of a grid. Particle properties are interpolated to the grid, the prediction is performed using them and the result interpolated back to the particles (see [Lad+15] and chapter 4). Ladicky et al. even note that their trained regressor has problems predicting a perfectly still fluid, similarly to our findings in chapter 5.
The most significant difference is the choice of machine learning approach. We used a neural network instead of a regression forest with the intention of investigating how neural networks might be used for such a task. One noteworthy difference between neural networks and regression forests is that the latter are make their prediction based on the most discriminative feature, while neural networks factor all input features into

their prediction.
Additionally, though we experimented with a version of integral features our grid-based approach did not incorporate them in the end. The input features we used are therefore not evaluated on a set of regions around a grid cell but only on its immediate neighbor cells. Lastly, our approach differs in scope with respect to the training. While Ladicky et al. trained on a cluster of ten computers for more than four days, using more than 600 billion samples to do so, our training was performed on a single computer and rarely took longer than two hours. This was done to enable quick testing of our approach and iterate on it.

2.2 Neural Networks for Accelerating Eulerian Fluid Simulation

Tompson et al. investigated the use of neural networks for fluid simulation by taking a Eulerian, i.e. not particle but fully grid-based approach [Tom+16]. They managed to train a convolutional neural network (see [Nie17]) to replace the computationally expensive step of solving the Poisson equation to enforce the incompressibility constraint of the fluid. The trained network predicts the whole pressure field each time step. The pressure field's gradient is then used to predict the velocity field and thereby the movement of the fluid. Tompson et al. present results which show their method to significantly improve the performance of this step of the Eulerian fluid simulation.
Though also using neural networks to predict the pressure and model the incompressibility of a fluid, the approach is less similar to this thesis than that of Ladicky et al. [Lad+15]. Apart from being a Eulerian method and therefore not requiring the back-and-forth interpolation between particles and grid, Tompson et al. do not predict the value of each grid cell individually but the whole pressure field at once. This allows them to incorporate not only the difference of predicted and expected pressure into their cost function but also the divergence of the velocity field resulting from the predicted pressures. The approach therefore has the advantage of predicting overall more coherent fluid movement compared to the method discussed in this thesis, which only uses the immediate surroundings of each grid cell to make predictions.

3 Algorithms

This section describes the main algorithms used in this thesis. As a fluid simulation method, we chose Weakly Compressible Smoothed Particle Hydrodynamics (WCSPH). This method is fairly simple and predicts forces for each particle based on the current state of all particles without iterative solving. It therefore proved to be a good starting point for attempting to predict the forces within a fluid using a neural network. We opted to use a neural network as our machine learning method since they are theoretically able to learn any possible function [Nie17] and to investigate their viability for particle-based fluid simulation.

3.1 Weakly Compressible Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) is a Lagrangian, i.e. particle-based, method of fluid simulation. The fluid is approximated by a set of particles which move with it, providing the advantage of implicit mass conservation. Each particle distributes its physical properties, e.g. its mass, over a spherical space of a fixed radius h (see Figure 3.1). This can then be used to interpolate a continuous property $A(x)$ of the fluid at a position x [BT07]. This interpolation is performed as:

$$A(x) = \sum_j (m_j \cdot \frac{A_j}{\rho_j} W(x - x_j, h))$$

To calculate the value at a position x, the contributions of the particles within the radius h of that position are summed up. Each contribution depends on the mass m_j, density ρ_j and quantity to be interpolated A_j of the particle j. Additionally it is weighted by the value of the so called smoothing kernel function W given the distance of the particle from the position x. The smoothing kernel thus essentially determines how a particle distributes its quantities over space. While in Figure 3.1 a linear radial gradient is used to visualize this, we used a cubic spline as depicted in Figure 3.2 in this thesis.
Using the SPH interpolation described, forces can be calculated that move the particles

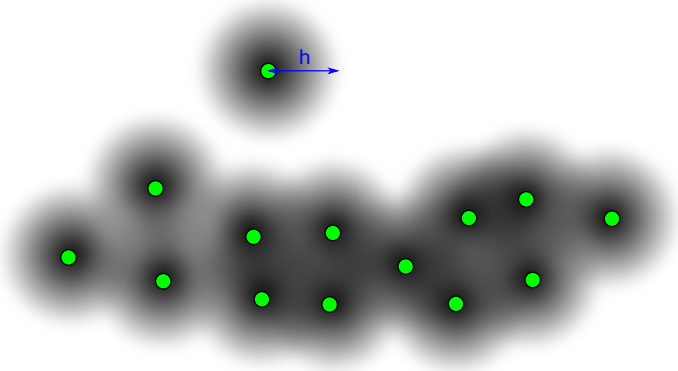

Figure 3.1

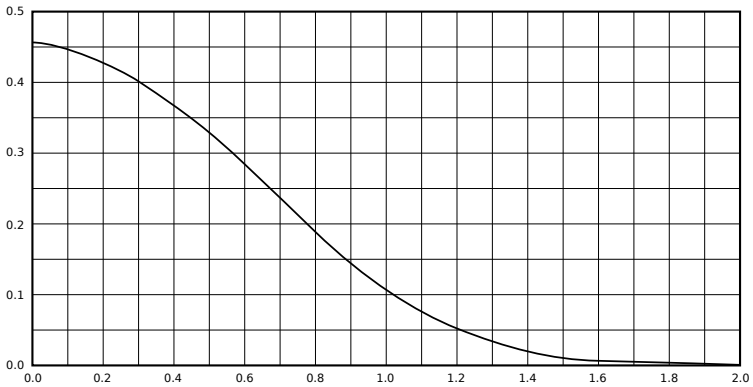

Figure 3.2: The smoothing kernel function W used for this thesis. The horizontal axis represents the distance fed into the function, the vertical axis the resulting kernel value. The function is a cubic spline depending on the support radius chosen, here 2.

as the fluid would. The three most important forces are viscosity, surface tension and pressure force. In this thesis, we will focus on the pressure forces, as they are what keeps the fluid from compressing and they are arguably the most important forces

to make the particles move believably as a fluid. To compute them, the first step is to interpolate the density at the position of each particle. Since the quantity A to be interpolated is the density ρ, this shortens the equation to $\rho(x) = \sum_j (m_j \cdot W(x - x_j))$. The resulting density at each particle's position must then be related to a pressure value using an equation of state. These pressure values indicate how the density at the point compares to the rest density of the fluid - in particular if it is too high. This then forms the basis of calculating the pressure forces which should push particles away from high pressure positions. Weakly compressible SPH (WCSPH) uses Tait's equation [BT07] as equation of state:

$$P = B \cdot ((\frac{\rho}{\rho_0})^\gamma - 1)$$

where γ is chosen to be 7 and B is a fixed pressure (see [BT07]). The resulting pressure P is therefore dependent on the ratio of the resting density of the fluid ρ_0 and interpolated pressure ρ. Tait's equation of state results in very weak compressibility since it gives high pressure values for low density variations. WCSPH does not directly enforce incompressibility based on these pressure values each simulation time step as other approaches such as Predictive-Corrective Incompressible SPH (see [SP09]) do. Instead the state of pressure values is used to directly predict the pressure forces. The equation for this is [BT07]:

$$f_i^{press} = -\sum_j (m_i \cdot m_j \cdot (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \nabla W(x_i - x_j))$$

Here the pressure force f_i^{press} acting on the ith particle is calculated using its mass m_i, position x_i, density ρ_i and pressure p_i as well as those of the particles j in its neighborhood. The gradient of the smoothing kernel ∇W essentially provides a scaled vector connecting particles i and j [Mon05]. Once these pressure forces have been calculated for all particles, the particles are then advected using euler integration. This then leads to a new state of particles in the next time step, meaning the densities, pressures and pressure forces have to be calculated again.
The neighborhoods for each particle must also be calculated at each time step. Without them, the WCSPH algorithm would have to consider all particles when interpolating density or pressure force, even if they are further away than the kernel W's radius and therefore have no influence. This would result in an n^2 complexity of the method, where n is the number of particles. The neighbor data calculation is realized using a grid storing references to all particles in each cell. This can then be used to compute a list of neighbors for each particle where only particles in cells close to the current

particle have to be considered as potential neighbors.
Though the neighborhood calculation speeds up the WCSPH approach overall, it is itself the most computationally expensive operation. Our approach therefore seeks to avoid explicit per-particle neighbor calculation and instead use a neural network to predict the pressure forces from input data that is less expensive to compute.

3.2 Neural Networks

A program usually specifies steps on how to compute a desired output from given input values. The programmer explicitly tells the computer what to do. Neural networks take a different approach to this. Instead of writing down an exact computation, the network is fed example data consisting of input values and the corresponding desired outputs. The network gradually infers the underlying rules through training with this data. In the best case, it can then generalize to data not contained in the example set. Like this, a neural network can theoretically learn any mapping of values. Images of handwritten numbers can be mapped to their numerical representation, mathematical mappings such as $x \rightarrow sin(x)$ can be approximated.
A neural network is made up of nodes called neurons. They take their inputs x and compute their output as $\sigma(w \cdot x + b)$ [Nie17]. The weights for each input w and bias b are variables that are usually randomly initialized, then gradually adjusted during training. Their values are what a network actually learns. The activation function σ finally calculates an activity for the neuron, usually a value in $[0, 1]$ or $[-1, 1]$. If chosen to be smooth, the activation function ensures that a small change of weights and bias will result in a small change of activation [Nie17]. This is crucial for successful learning. When changing w and b to better fit a new piece of training data, the neurons overall output and behavior should not change drastically since it should still perform similarly on previous data samples.
Connecting these neurons via their inputs and outputs forms the actual network. In the standard feedforward structure (Figure 3.3), they are arranged in layers. The input and output layer are used to feed data into the network and receive its prediction for it. The actual computation is performed by the neurons in the hidden layers. If fully-connected, each receives the outputs of all the neurons in the previous layer, which could be the input layer or another hidden layer, as its inputs. The network can be evaluated by successively computing $\sigma(X \times W + B)$ for each layer, where σ is applied element-wise and X, W, B are matrices containing inputs, weights and biases of the layer.

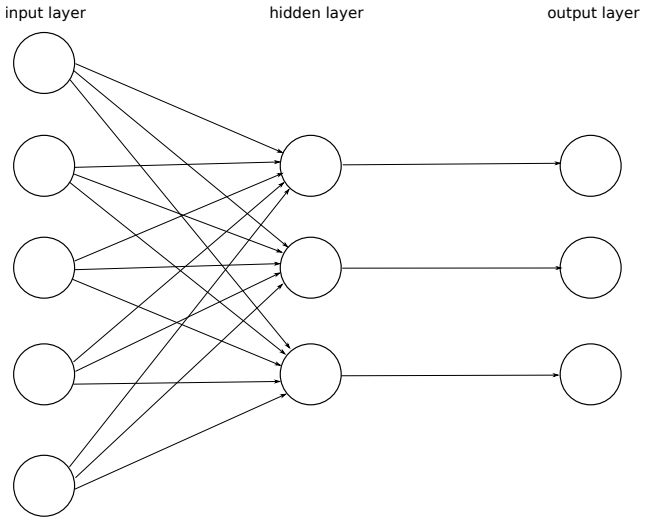

Figure 3.3: Fully-connected feedforward neural network structure

Evaluating a network yields a prediction for the output. In order to learn, it is necessary to specify how the prediction compares to the expected output, also referred to as the ground truth. For this purpose, a cost or loss function must be defined. The weights and biases can then be adjusted to minimize the cost function, making the network approximate the ground truth more closely.
The gradient of the cost function is used to determine how exactly they should be changed. Since it consists of partial derivatives $\frac{\partial C}{\partial w}$, it gives insight into how changing a weight w (or bias) would impact the cost function C. To compute the gradient quickly and with sufficient accuracy, the backpropagation algorithm is used. Its application to learning is what enabled neural networks to be used in the capacity they are today.

4 Approach

The goal of this thesis is to train a neural network using the data from several pre-computed WCSPH simulations. The trained neural network should then replace the actual computation of pressure forces in a separate fluid simulation. There are therefore three distinct tasks that should be solved: Generating the training data, training the network and applying the trained model to a simulation (see Figure 4.1).

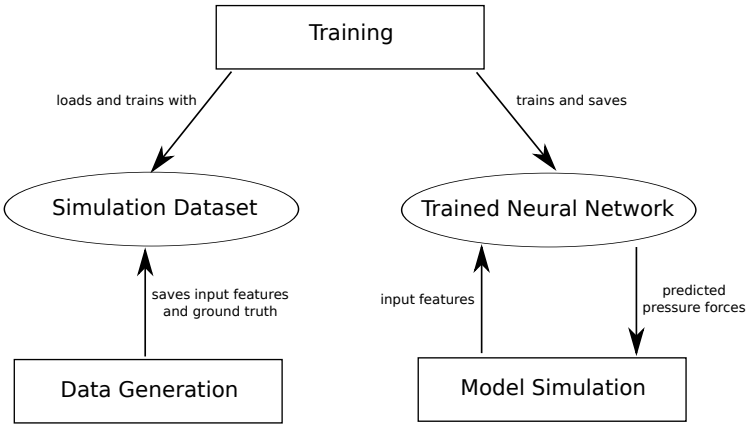

Figure 4.1: Interaction between the main components

The Data generation component runs an example WCSPH simulation. For each frame, it calculates the input features and corresponding ground truth values that the neural network should be trained with. It then saves this data to disk. This can be repeated multiple times with different example simulations to yield the full simulation dataset. This data is loaded in training, preprocessed (see subsection 4.3.1) and used to train the network model. The trained model is saved to disk. Finally the model simulation loads and uses it to predict the pressure forces acting on the fluid particles. To do so,

input features are calculated at each time step, fed into the network and the resulting predicted forces used to advect the particles.

4.1 Input Features and Ground Truth

In WCSPH, in order to compute the pressure force acting on each particle, information about the distribution of fluid mass in the area surrounding the particle is required. The density value of each particle describes how much of the fluid's mass is concentrated at that particle's position. Applying the equation of state yields the particle's pressure value. It indicates how much higher the density is than it should be for the fluid to be uncompressed. Due to its list of neighbors, each particle thus has information about where the fluid is being compressed in its vicinity. The pressure force can then be calculated so that it moves the particle away from those high pressure positions.
For the neural network to learn and predict the pressure forces, it needs similar information as input features. To meet our goal, this has to be achieved without explicitly calculating a list of neighbors for each particle.

The proposed approach calculates input features and ground truth on a grid instead of per particle. On a grid it is much easier to check the neighbor values, as one simply needs to look at the surrounding grid cells - no explicit neighbor calculation is required. The predicted pressure forces for each grid cell can then be interpolated back to the particles (see section 4.4) to obtain forces that affect the movement of each particle. The input feature vector of a cell consists of the pseudo pressure values and flag values of itself and the surrounding eight cells, making for eighteen values in total. The following sections describe how the pseudo pressure grid and flag grid are calculated in detail.

4.1.1 Density Grid

To avoid per-particle neighbor data for density computation, we use a grid to distribute the mass of the particles over the simulation space.
Each grid cell's density value ρ_i is calculated as:

$$\rho_i = \sum_j (m_0 \cdot W(||x_i - x_j||))$$

The sum iterates over all the particles j that are within a radius around the grid cell's midpoint x_i. For this, the same radius is used as when computing density normally.

For each particle j, the distance from the cell center x_i is fed into the kernel function W which returns a corresponding weight in $[0, 1]$. The particle mass m_0, which is in our case the same for all particles, is multiplied by this weight and added to the grid cell's density value. Each particle therefore contributes to the cell's density proportionally to its distance from the cell center.

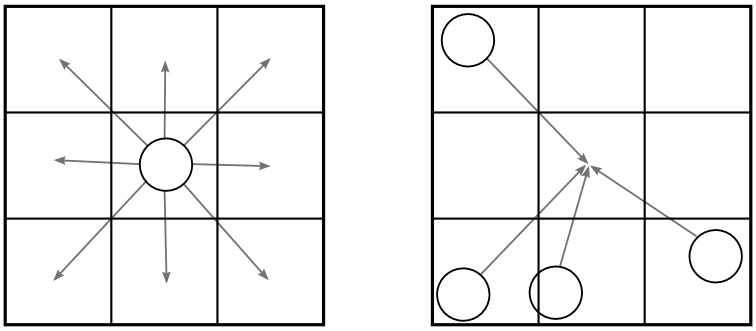

Figure 4.2: Method 1 (left): Distribution, Method 2 (right): Gathering

The two alternative methods of calculating the density value for each grid cell are shown in Figure 4.2. The distribution method iterates over all particles, adding their contributions to the surrounding cells' density values. In contrast method 2 iterates over all grid cells, gathering the particles from the surrounding cells and adding up their contributions. Both methods require the same number of computations, since they consider the same pairs of cells and particles. As can be seen from method 1, this number is $n \cdot c$, where n is the number of particles and c a small fixed number of grid cells, depending on the chosen radius. It is not dependant on the state of the particles, i.e. it does not change even if the particles are all very close to each other. By contrast, the normal WCSPH density calculation would have to perform n^2 operations in the worst case scenario, where all particles are within each other's radius.
Although we used method 1 since it is more intuitive, method 2 has the advantage of being more easily parallelizable. Method 1 has the issue of race conditions where multiple particles try to write their mass contribution to a cell's density value at the same time. By contrast, the gathering method operates per cell and only has to read the particle's contribution values, hence avoiding race conditions.

4.1.2 Pseudo Pressure

Normally the pressure is calculated directly from each density value. The density grid however cannot directly be used to compute a pressure value by simply applying the equation of state to each cell's value. This is because a cell's density value models the concentration of particle mass at its midpoint (see subsection 4.1.1). Calculating a pressure from it would only indicate whether too many particles are close to the center, rather than close to each other. In the example in Figure 4.3 a high pressure value

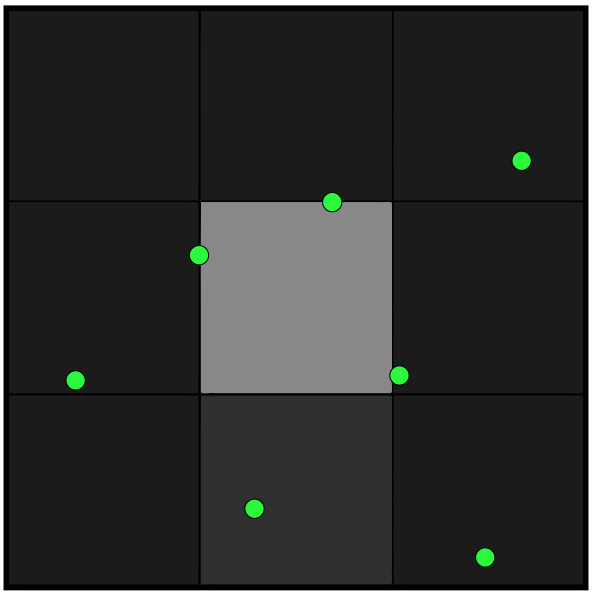

Figure 4.3: A pressure grid calculated directly from the density grid. Lighter colors for cells indicate higher values. Fluid particles are represented as green dots.

is falsely calculated for the middle grid cell. Three particles are relatively close to its center, thereby increasing the cell's density value. However, the particles are not too close to each other. As such, there should be no or little pressure.
To avoid this issue, we calculate the pressures at the actual particle positions. The densities at these positions are obtained by bilinear interpolation of the density grid. Applying the equation of state then yields a pseudo pressure value at each particle position:

$$p_j = p_0 \cdot ((\frac{\rho_j}{\rho_0})^\gamma - 1)$$

ρ_j is the interpolated density at the position of particle j. The resting pressure p_0, exponent γ and resting density ρ_0 are the same as in normal WCSPH pressure computation. If $\frac{\rho_j}{\rho_0}$ is smaller than one, the equation results in a negative pressure. This does not make physical sense, since there are no negative pressures. The value is therefore clamped at zero.
The pseudo pressure values at the particle positions need to be interpolated back to a grid. We do this in the same manner in which the particle's masses were interpolated to the density grid in subsection 4.1.1. Once this is done for all particles, the resulting pseudo pressure grid provides the first nine input features of each cell.

4.1.3 Obstacle Flags

In addition to the pseudo pressure values of the surrounding cells, the input feature vector of each grid cell should also contain information about obstacles in the vicinity. This helps to prevent the network from predicting pressure forces that would push the particles into an obstacle. This might otherwise occur in a situation such as in Figure 4.4.

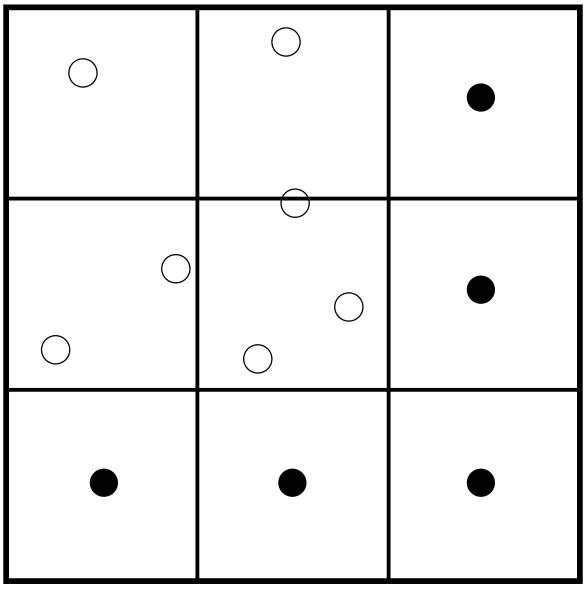

Figure 4.4: Fluid particles (white) congregating near an obstacle (black).

As fluid particles are pushed against an obstacle, the pseudo pressure of the cell may

become higher than that of the obstacle cells next to it. Without the information that these cells are part of a boundary, the network might predict a pressure force pushing the particles towards the boundary cells as their pseudo pressures are lower.
We therefore include a flag value of each surrounding cell in the input feature vector. This flag value is zero if the cell is an obstacle and one if not. The flag values make up the last nine values of a cell's feature vector.

4.1.4 Interpolated Pressure Forces

To predict with the trained network, the input feature vector of each cell is sufficient. But for training, we also need to calculate a ground truth value to calculate the cost function. In our case, the ground truth is the pressure force. This means a pressure force vector must be calculated for each grid cell. This only has to be done during data generation. It is thus not necessary to avoid neighbor computation here. The pressure forces can simply be calculated using the normal WCSPH method for each particle, then interpolated to a grid.

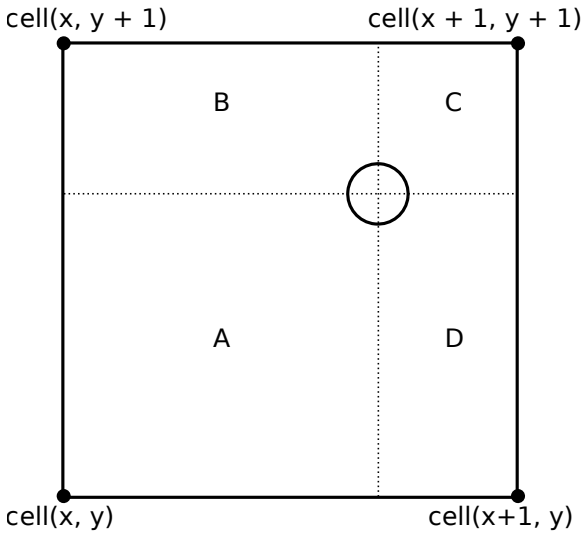

Figure 4.5: Bilinear interpolation is used to map the per-particle pseudo pressures to a grid.

To do so, we use bilinear interpolation, though in reverse. The weight of each cell corre-

sponds to the area opposite to it (see Figure 4.5). The particle's pressure force is then multiplied by the cell's weight and added to the cell. For instance, in Figure 4.5 the particle's pressure force would be multiplied by the area A, then added to cell (x + 1, y + 1).

4.2 Neural Network Structure

The neural network we use has a fully-connected feedforward structure (see section 3.2). It consists of three hidden layers that contain twenty neurons each. This choice is a trade-off between depth and training speed. A greater quantity of hidden layers can enable the network to learn the solution to more abstract problems more easily. To train a deeper network, however, much more training time is needed.
We also confirmed the network to be able to learn different simple functions such as $f(x) = sin(5 \cdot x)$ and lower order polynomials before attempting to train it with simulation data.

4.2.1 Cost Function

The cost function should model how far off the network's prediction is from the expected result, the ground truth. For our network, the ground truth y and prediction y_p are three-dimensional vectors. To compare them, it is intuitive to use the euclidean distance $||y - y_p||$. It is not necessary to calculate the actual distance, however, since the most important thing is for the value of the cost function to decrease noticeably the closer y_p is to y. We therefore use $C_0 = \frac{(y_1-y_{p_1})^2+(y_2-y_{p_2})^2+(y_3-y_{p_3})^2}{2}$ as our base cost function, omitting the costly computation of the square root.
We further augment the cost function using L2 regularization. This essentially makes the network "prefer to learn smaller weights" [Nie17]. The network will thus be less prone to changing the learned prediction due to single data points and instead learn a function that primarily fits data often seen during training. L2 regularization can hence be seen as a resistance to noise in the training dataset. It is added to the cost function as

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

n is the number of training samples, w represents an arbitrary weight. The factor λ essentially describes the importance of the small-weight-preference compared to minimizing the cost function overall.

4.2.2 Activation Function

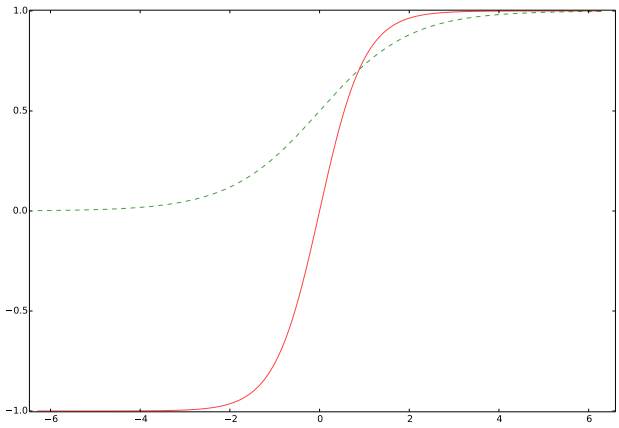

Figure 4.6: tanh activation (red line) vs. sigmoid activation (dotted green line)

As the activation function for each hidden layer we chose the hyperbolic tangent $tanh(x)$. It is differentiable at all points, meaning it can be used for gradient based optimization. Its limits $\lim_{x\to\infty} tanh(x) = 1$ and $\lim_{x\to-\infty} tanh(x) = -1$ are well-defined and different, making the network using it universal for computation [Nie17]. These are properties it shares with the widely used sigmoid activation function. As Figure 4.6 shows, the sigmoid function maps its input to the interval $[0, 1]$, as opposed to *tanh* which maps to $[-1, 1]$. This means that the output of a neuron using *tanh* will be zero-centered, which it would not be using sigmoid. Zero-centered input values for hidden neurons are desirable since it can mean faster learning convergence [Hay99]. Moreover, the $[-1, 1]$ range is suitable because our network will be used to approximate a vector which could well have negative components.
Sigmoid and hyperbolic tangent share a drawback, however. Their activation saturates at their limits, making the gradient correspondingly very small. A gradient close to zero can lead to the neuron being effectively stuck, leading to problems during backpropagation [Wei17]. To avoid this, correct weight and bias initialization (see subsection 4.3.2) as well as input data normalization (see subsection 4.3.1) are needed.

4.2.3 Optimizer

We used the Adaptive Moment Estimation (Adam) optimizer to minimize our cost function. This stochastic gradient descent optimizer uses a form of momentum and adaptive learning rates. The intuition behind this is to mimic a ball rolling down hill and the way its momentum carries it in the direction it has built up in, even if it hits a small bump. In a similar vein, the Adam optimizer builds momentum along those dimensions of the cost function whose gradients do not change directions too much. This enables it to "go down hill" i.e. find the minimum faster and with less oscillation [Rud16].

4.3 Training

Training is performed in epochs. Within an epoch, the whole training dataset is split evenly into batches of a predetermined batch size. For each of these batches, an optimizer training step is performed (see subsection 4.2.3). The training therefore traverses the entire training dataset every epoch. At the end of each epoch, it is shuffled so that the next epoch will not train with the same batches.

4.3.1 Dataset

Before the data can be used to train the neural network, it has to be pre-processed. The most important part of this is normalization. The pseudo pressures and pressure forces are usually very large - far outside the "active range" of the hyperbolic tangent. This can lead to a very small gradient and therefore extremely sluggish learning (see subsection 4.2.2). Also, by comparison, the obstacle flag values of the input feature vector are very small, giving them much less influence. Due to this, the pseudo pressures and pressure forces should be normalized. This is done by looking at the values occurring in the dataset, finding an upper bound and dividing all values by it.
Another step to optimize the dataset for fast learning is to remove data which does not represent relevant information the neural network should learn from. In our case, grid cells whose feature vector only contains pressure values that are zero can be discarded. This can be done without a threshold value, as the pseudo pressure computation already performs clamping.
Once the dataset is loaded it needs to be split into a training and a test set. The training set contains the data the network will actually be fed during training. The test set is

used to check how well the network generalizes to unknown data. This also helps to recognize if the network is overfitting, essentially learning the training data by heart. If the cost on the training data decreases while the predictions for the test data do not improve, it indicates that the network is overfitting.

4.3.2 Weight and Bias Initialization

While biases can be initialized to zero or a small positive value, it is important to randomly initialize the weights so that neurons on one hidden layer do not behave the same during training [Ben12]. How exactly this is done can drastically affect the networks ability to learn, especially in a network with multiple layers. If the weights are initialized too small, this can lead to the input's variance diminishing throughout the layers. For an activation function like the hyperbolic tangent (see Figure 4.6), the activation is almost linear around zero. If the variance of the input shrinks enough, the network thus acts as it would with a linear activation function [Jos16], losing its ability to approximate non-linear functions. If the weights are initialized at high values, this can lead to the neurons becoming saturated and the gradients becoming close to zero (see subsection 4.2.2).
To avoid these problems during weight initialization, we use Xavier Glorot and Yoshua Bengios's method [GB10]. This method chooses the variance of the random distribution the weights are drawn from depending on the number of inputs n_{in} and outputs n_{out} of the neuron. In our case a uniform random distribution $U(-r, r)$ where $r = \sqrt{6/(n_{in} + n_{out})}$ is used.

4.4 Simulation using Trained Network

During data generation, the fluid is simulated using WCSPH and the input features are calculated and saved without having any influence on the particle's movement. In the model simulation, this is different. It loads the trained network and uses it to predict the fluid's behavior. Each simulation time step the input features for the grid cells are calculated and fed into the network. Since the network is trained with normalized ground truth values (see subsection 4.3.1), the network's estimates need to be denormalized using the same factor. The resulting predicted pressure forces per grid cell are then bilinearly interpolated to the particles. Using euler integration these per-particle forces then change the particles' velocities and thus movement.

4.4.1 Corrective Measures

The neural network can only approximate the pressure forces, with the predictions always containing a small error. We hence introduce two corrective measures to keep the predicted simulation stable and believable.

Enforcing Obstacle Collisions

As the predicted forces are never fully accurate, they should not be relied upon to resolve collisions between the fluid and obstacles. Doing so might result in a portion of the particles moving into the obstacles. This would then lead to input features that the neural network would have difficulty predicting from, in turn leading to less accurate predictions for the forces.
We therefore use a simple version of collision detection and resolve to prevent this. The method is applied each time step, after the particles' velocities have been updated but before their position update. For each particle, it checks whether its updated velocity would move it into an obstacle cell. If this is the case, the dimensions of the velocity moving the particle towards the obstacle are set to zero. This allows the particle to move along the obstacle, but not into it.
This method has several disadvantages. First, it can only deal with obstacles that can be fully described using the flag grid, as this is used to determine whether the cell the particle would move to is an obstacle. This means the approach only covers rectangular, static obstacles. Furthermore, setting the velocity to zero is a very crude form of collision response, featuring no form of impact or projection of the velocity onto the normal of the obstacle surface. For our purposes though, it is sufficient, as it is easy to implement, fast to evaluate and the simulated scenes do not contain any obstacles the method cannot cope with.

Removing Prediction Bias

For any feature vector containing pseudo pressures that are all zero, the predicted pressure force should be zero as well, regardless of the flag values. Preliminary tests of the approach showed that this is in practice not the case. As neural networks only approximate the solution more and more closely, yet never completely reach it, each cell has a prediction bias - a small pressure force that is predicted even if all pseudo pressure features are zero. This prediction bias is particularly large for cells near the boundaries. This is likely the case due to high pressure forces occurring much more

often there. The bias is not only a part of the prediction for zero pseudo pressure values, but is contained within all predictions. We therefore have the neural network predict the pressure forces for each cell twice: Once normally and once with all pseudo pressure values set to zero. We then subtract the second from the first to obtain a prediction without bias.

5 Experiments

This section details the experiments performed to test the approach described in chapter 4. For each experiment a network was trained using the same architecture (see section 4.2) and hyper parameters for better comparability. We used an initial learning rate of 0.001, a batch size of 1000 and an L2-regularization factor of 0.001. Training was performed for 300 epochs. These values were mainly chosen through experimentation. We trained on a single Central Processing Unit (CPU) only, specifically an Intel Core i7 3770 clocked at 3.40 GHz complemented by 24 Gigabytes of DDR3 RAM.
All simulation data was generated using a 64 by 64 grid and a simulation time step of 0.001, with one simulation running for 300 frames at a frame rate of 30 frames per second. The simulation scenes were encased by a boundary for the fluid to be pushed against.

To determine how well each trained model simulates the fluid in various situations, we chose three specific setups for the model simulation as shown in Figure 5.1. They test

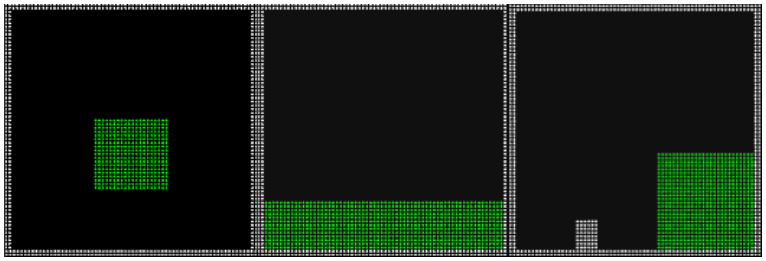

Figure 5.1: Falling fluid, resting fluid and dam break simulations (from left to right)

whether the trained model can cope with sudden high pressure from falling onto the ground, low pressure which keeps the fluid from compressing in the resting state and punctual pressure to the sides due to obstacles.

5.1 Randomized Input Data

The first neural network was trained with randomized simulations of one body of fluid falling onto the floor. For this, six example simulations were computed during data generation. The bodies of fluid were centered horizontally and placed 0 to 10 percent of the simulation space height above the lower boundary. Their height and width were generated between 15 to 20 and 20 to 30 percent of the domain length respectively. Each simulation consisted of 300 frames with 64^2 samples - one per grid cell. After data pre-processing, this resulted in 941260 samples to be trained with. Training took

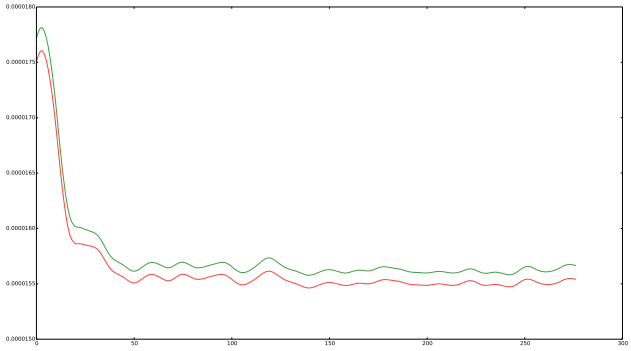

Figure 5.2: Cost on training (red) and test data (green) while training the first model

29.803 minutes, measured using Python's process time [Pyt17]. Figure 5.2 shows the cost on test and training data over the training epochs. The curves were smoothed using a moving Gaussian algorithm [Har08] to make the result more legible. Apart from some jitter, the cost seems to converge around the 150 epoch mark, indicating that the network's predictions do not improve thereafter.
Applying these predictions to the three test model simulations showed the network to be able to keep the fluid from compressing too much. While encouraging overall, the predicted fluid did not behave as desired in two regards in particular. The first was primarily noticeable in the dam break and resting fluid test cases.
If the fluid was pressed against an obstacle the trained model predicted pressure forces moving the particles upwards. This can be seen quite clearly at the beginning of Figure 5.3, where the particles resting against the right wall fly up. It is also visible in Figure 5.4 where the fluid "flows up" the left wall. The phenomenon can be explained

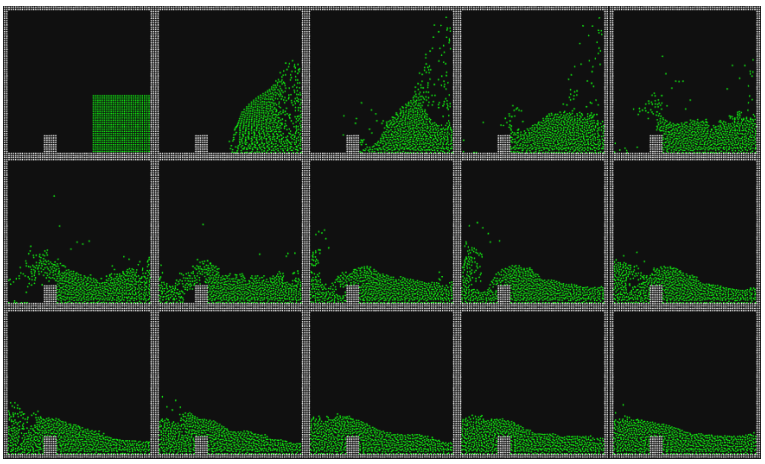

Figure 5.3: Every 40th frame of the dam test using the first trained model

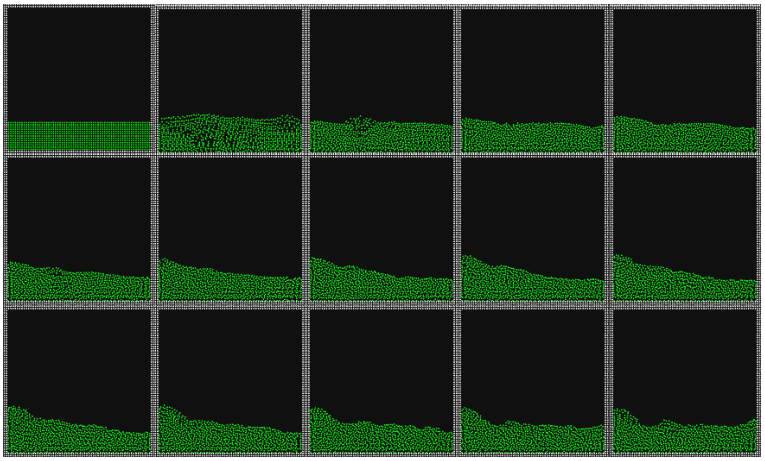

Figure 5.4: Every 40th frame of the resting fluid test using the first trained model

by the fact that the model was only fed data about bodies of fluid falling onto the floor. The training data therefore contained a lot of information on pressure forces acting vertically and very little on horizontal ones. The network could therefore not learn to adequately react to pressure from the sides, instead predicting mainly vertical pressure forces for higher pressure input.

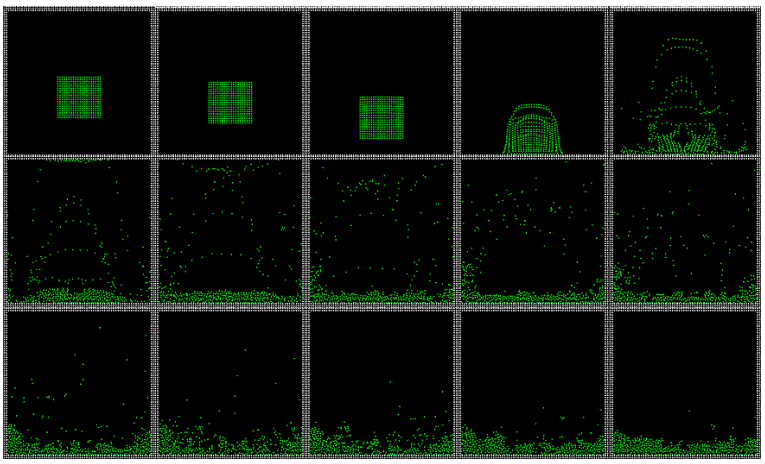

Figure 5.5: Every 20th frame of the falling fluid test using the first trained model

The second visual inconsistency was the strength of the pressure forces predicted for high pressure input. In the falling test simulation the fluid bursts apart upon hitting the bottom boundary (see Figure 5.5, frames 4-6). A possible explanation for this is that the trained model could not keep the fluid from compressing entirely. In the following frames, this then lead to input features containing higher pressures than the network was trained with. The model then predicted very high pressure forces from them due to imperfect generalization.

5.2 Rotated Input Data and Gravity Multiplier

Due to the observations made in section 5.1, we modified the training data for the second neural network in two main ways. First, the network needed to receive more data enabling it to react to pressures from all sides, rather than just from below. We therefore used each training simulation four times in the dataset, incrementally rotated by ninety degrees.
Secondly, more data was required for high pressure cases. In normal simulation conditions, such high pressures only occur for a short time, as the WCSPH solver quickly resolves them. This would necessitate a massive number of simulations to get a lot of data on high pressures, as each simulation would only contain a few frames where it occurs. We instead chose to use amplified gravity to artificially generate high

pressures throughout a whole simulation. Each training simulation was run with three different gravity multipliers: 1, 5 and 15.
These measures meant that one scene would be varied in twelve different ways, resulting in twelve times the amount of training data per example scene. To keep training time low, we therefore only used one such scene which needed to contain enough variation in pressure values to be viable. Surprisingly, a setup with a resting fluid showed to have a good range of pressure values (see Figure 5.6). We used the twelve variations

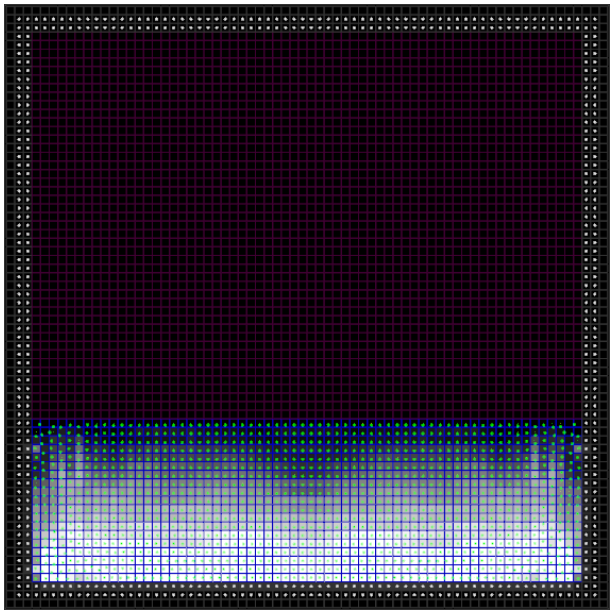

Figure 5.6: Pseudo pressure values in a resting fluid simulation (lighter values equal higher pseudo pressure)

of this scene, resulting in 4123891 samples, to train the second network. Due to the larger amount of data, training took significantly longer than in section 5.1 with 98.169 minutes. The cost developed similarly to the previous training run, though with much more fluctuation (see Figure 5.7), which could be attributed to the higher variation of input features. Notably the cost does not go lower than in the previous training. The costs were made comparable by dividing them by the length of the respective dataset. Otherwise, the cost for the second training would have been much higher since a larger dataset would result in a larger prediction vector y_p and therefore higher cost (see subsection 4.2.1).

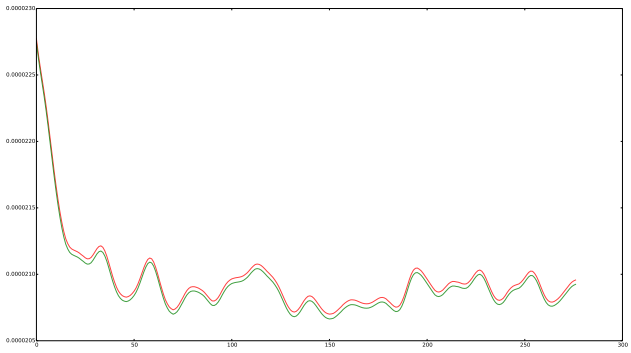

Figure 5.7: Cost on training (red) and test data (green) while training the second model

Although the cost remained at a similar level, the test simulations showed a noticeable improvement over the previous network. The trained model no longer predicted

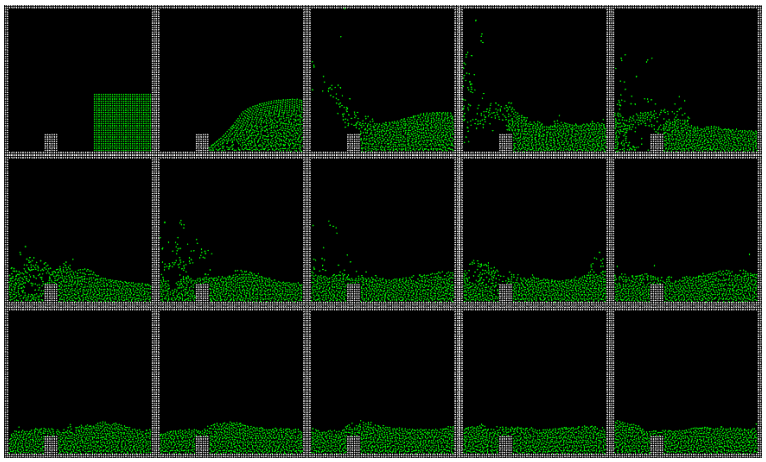

Figure 5.8: Every 40th frame of the dam test using the second trained model

upwards pressure forces due to obstacles to the sides (see Figure 5.8, frames 1-5) and reacted better to high pressures such as when hitting the ground at speed (see Figure 5.9, frames 4-6). Though the resting fluid simulation also improved, the network was still unable to fully keep the fluid calm (see Figure 5.10, frame 2). One possible reason for this is that the training data still lacked variance for low pseudo pressure

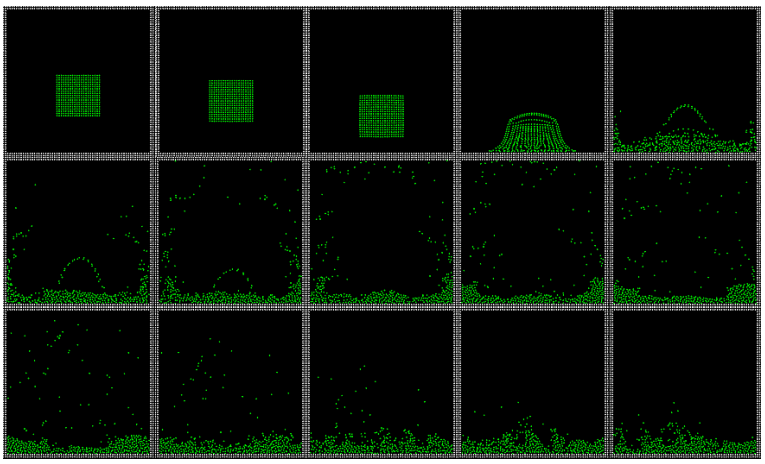

Figure 5.9: Every 20th frame of the falling fluid test using the second trained model

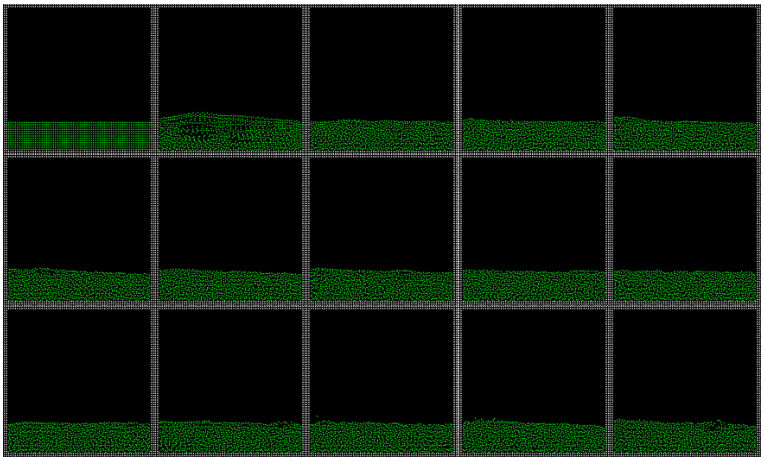

Figure 5.10: Every 40th frame of the resting fluid test using the second trained model

values. This means that the input features did not contain enough unique values to cover the range of ground truth values that should be mapped to. As Figure 5.11 shows, it is problematic if the training data contains samples where the input features are very similar or the same, yet there is greater variance in the expected output. This inhibits the network from learning to predict a nuanced range of values.

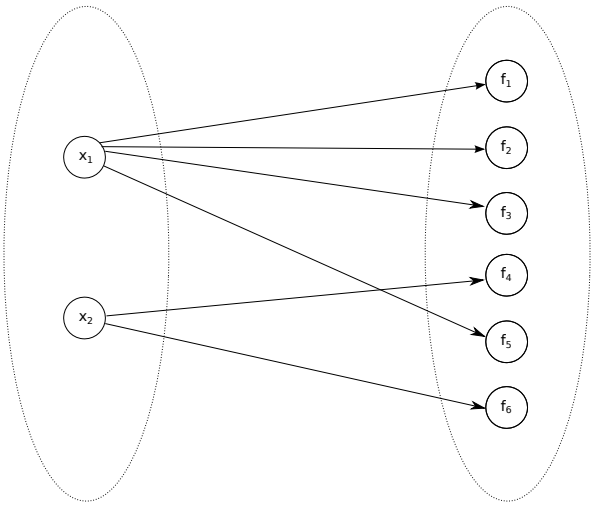

Figure 5.11: Mapping of insufficiently variant input features (left) to much more varied ground truth (right)

5.3 Training with Actual Pressure

To further investigate the variance of low pressure values in the training data, we created a histogram of the occurring values in the dataset used to train the second neural network. This is shown on the left in Figure 5.12. We compared this to the actual

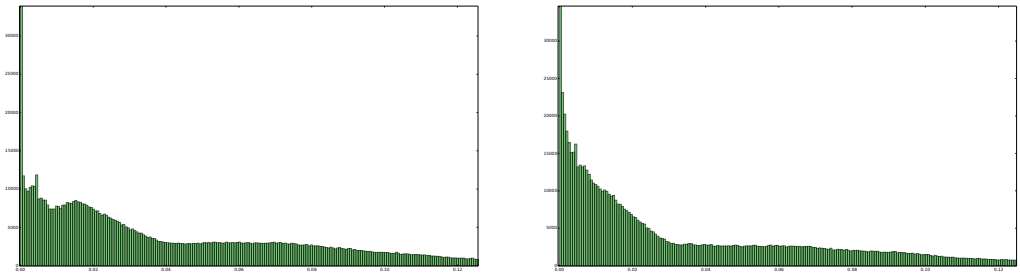

Figure 5.12: Histograms of the occurring normalized pseudo (left) and actual pressure values (right) in the training dataset

pressure values for the same set of simulations, shown on the right. For this we did not use the pseudo pressure calculation described in subsection 4.1.2. Instead, we calculated the pressure values per-particle using the WCSPH approach, then interpolated these pressures to the grid. Though the histograms look very similar overall, there is a distinct difference in the lower range, specifically between 0.00 and 0.02. The right histogram's columns are significantly higher there. This suggests that the actual pressures have more variance for low values than the pseudo pressures. We therefore trained the third network with actual pressure values. The same twelve simulations were used as in section 5.2, with the input feature vector of each sample containing actual instead of pseudo pressure values. However, since the actual pressure values require neighbor data calculation, they could not be used to predict during the model simulation. As such, although we trained with the true pressures values, we used pseudo pressure values to predict as before.
The actual pressure training data resulted in 4273956 samples after data pre-processing. This was an increase from the 4123891 samples in section 5.2, even though the example simulations were exactly the same. This suggests that pseudo pressure samples were often discarded due to being zero when their actual pressure counterpart was in $[0.0, 0.02]$. Training the third neural network took 118.759 minutes. The cost during

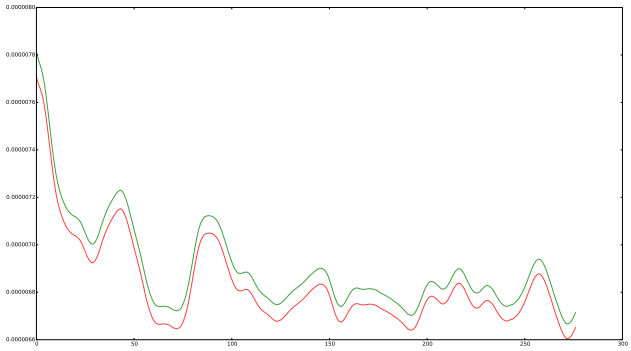

Figure 5.13: Cost on training (red) and test data (green) while training the third model

training fluctuated even more than in section 5.2, yet was overall lower. It is possible that here, training for more than 300 epochs might have further improved the network's performance. The test model simulations showed mixed results. The resting fluid simulation improved over the second neural network, yet still could not keep the fluid

fully at rest (see Figure 5.14, frame 3).

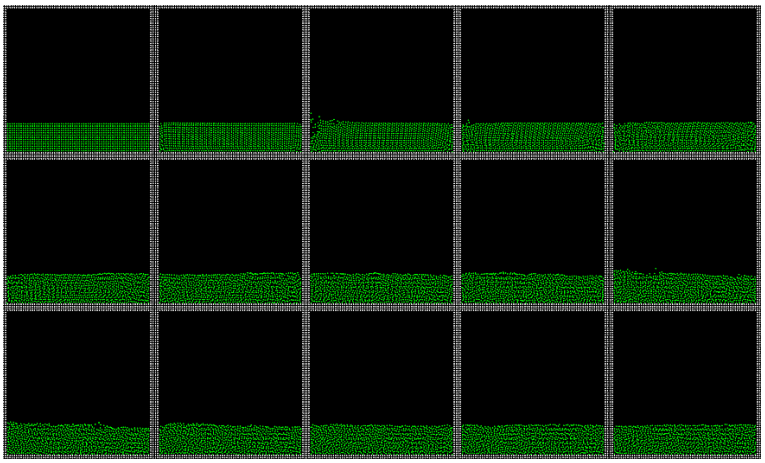

Figure 5.14: Every 40th frame of the resting fluid test using the third trained model

The dam break and falling fluid simulations performed slightly worse than using the second trained model. The simulated fluid seemed less cohesive, with the pressure forces more likely to push particles apart instead of just keeping them from becoming too close. This can be seen by comparing the 4th frame of Figure 5.15 and the 5th frame of Figure 5.16 to their counterparts in Figures 5.8 and 5.9. The reason for this is likely the disconnect between the actual pressure data used to train and the pseudo pressure data used to predict. Due to this, even though this approach improved the stability of the resting simulation, it seems overall unsuited.

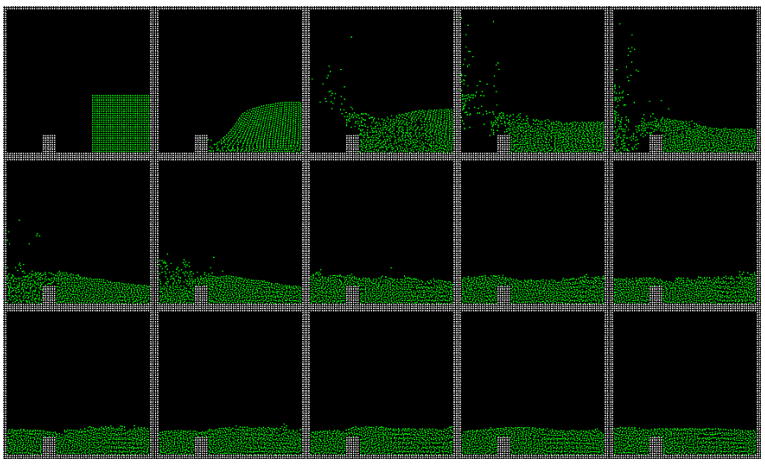

Figure 5.15: Every 40th frame of the dam test using the third trained model

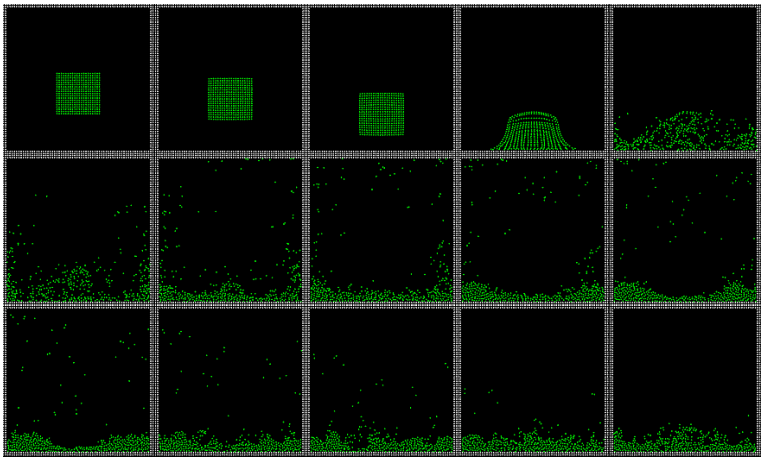

Figure 5.16: Every 20th frame of the falling fluid test using the third trained model

5.4 Other experiments

Before implementing and testing the main approach described in this thesis, we tested several other related approaches. Initially, we intended to have the neural network learn and predict pressure forces per particle, not per grid cell. To do so, a we sampled the input feature grids, e.g. the pseudo pressure grid, at nine fixed positions relative to

the particle. This means bilinear interpolation of the grid's values to these positions. This did not lead to reasonable results for any input feature grid we tried - not for the pseudo pressure or density grids as described in subsection 4.1.1 and 4.1.2, nor for a grid containing the number of particles per cell. The most likely explanation as to why this worked for per-cell but not for per-particle learning is again related to the variance of the expected output. Interpolating the particle's pressure forces to a grid smooths them. This reduces their variance and allows the neural network to find a much better mapping.
We attempted to exploit this by training our network per grid cell and then using it to predict per particle. This produced artifacts near obstacles, however. Due to the flag input features (see subsection 4.1.3) only being 0 or 1 during training per grid cell, the network could not generalize to interpolated flag values in between which occurred when predicting per particle.

5.5 Used Libraries

To implement our approach and perform the experiments, we mainly made use of two libraries. Mantaflow was used as a fluid solver in data generation and for the model simulation. For training and everything relating to machine learning we used TensorFlow.

5.5.1 Mantaflow

Mantaflow is an open-source framework for fluid simulation developed at the TU Muenchen Games Engineering Group and the ETH Computer Graphics Laboratory by Tobias Pfaff and Nils Thuerey [TP16]. A variety of fluid solving algorithms are supported, though we only used the simple WCSPH approach. These algorithms are, for optimal performance and efficiency, mainly implemented in C++. For this thesis, several functions had to be added to the Mantaflow source code to compute the input feature data the neural network should learn from and predict with.
Mantaflow uses Python scripts called 'scene files' to define an actual simulation and call the fluid solving functions which provide Python interfaces for this purpose. These scenes may be 2D or 3D. For simplicity, this thesis sticks to the 2D case, though the Mantaflow support means it should be possible to adapt the results to 3D as well. The scene files enable easy customization of the setup of a scene. Variables such as obstacle placement, amount of fluid particles or strength of external forces may all be changed

without altering the fluid solving algorithms themselves. This was used to generate the different scene setups for data generation and the model simulation tests.
Mantaflow also provides a GUI which can be used to visualize data being computed per particle or per grid cell. This was very useful for verifying that the neural network input feature data was being calculated correctly. It also helped in checking the pressure forces the trained neural network predicted.

5.5.2 TensorFlow

TensorFlow is an open-source software library [Goo16] which allows the construction and evaluation of computation graphs. It was originally developed within Google for machine learning and deep neural networks research. TensorFlow follows a lazy programming paradigm - it first builds a graph of all the computations and their relationships and only executes it once a so-called 'Session' is run.
We used it to describe and build neural networks as graphs, then train them. Functions for saving and loading the state of such a trained graph are also provided. This enabled us to train a network once, then load the graph into an arbitrary Mantaflow scene and use it to replace the pressure force calculation.
The built in Tensorboard functionality also allowed for visualization of the training process, e.g. by plotting the cost on the test dataset over time.

5.5.3 Other Libraries

Further libraries were used to make data manipulation and representation easier in Python. NumPy [Num16] was used for easier manipulation of the training and test datasets and to transfer data between Mantaflow and TensorFlow functions. To better visualize training, we used Matplotlib [Mat16] in addition to Tensorboard, for example to plot the histograms and cost curves in this chapter.

6 Discussion

The experiments in chapter 5 show that our approach can be used to believably simulate fluid particles using a neural network without explicitly calculating per-particle neighborhoods. However, the model simulations still retain some flaws we were unable to resolve. Since the goal was to build a neural network prediction based simulation that trades correctness for performance, this would in theory be acceptable. This chapter therefore discusses the advantages and disadvantages of our approach regarding correctness of the simulation and performance to establish the overall viability of our approach.

6.1 Correctness

Due to using a grid instead of exactly calculating neighbor data for each particle, our approach produces slightly incorrect density values for each grid cell. This can be seen in the following example.
In Figure 6.1 there are four particles whose density should be calculated using the approach from subsection 4.1.1. In both setups they are positioned identically relative to each other, however they are translated with respect to the grid. Since their relative distances are the same, their density values should also be the same. Calculating the densities using our approach yields different results however.
We first calculate the density grid using the smoothing kernel W from Figure 3.2. Using this kernel, all distances greater than 2 can be considered to have no influence. Thus, there are only a few particle-cell distances that need to be considered here. For the left setup they are 0, 1 and $\sqrt{2}$ with respective kernel values of $W(0) \approx 0.455$, $W(1) \approx 0.11$ and $W(\sqrt{2}) \approx 0.025$. The distances occurring in the right setup are less obvious. They are $|| \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} || = \frac{\sqrt{2}}{2}$ and $|| \begin{pmatrix} 0.5 \\ 1.5 \end{pmatrix} || = || \begin{pmatrix} 1.5 \\ 0.5 \end{pmatrix} || = \frac{\sqrt{10}}{2}$. Their kernel values are $W(\frac{\sqrt{2}}{2}) \approx 0.225$ and $W(\frac{\sqrt{10}}{2}) \approx 0.01$. With these kernel values we can calculate the density grid in Figure 6.2 - we assume the mass m_0 to be 1 for this example. Following our approach from subsection 4.1.2, we interpolate these grid values back to

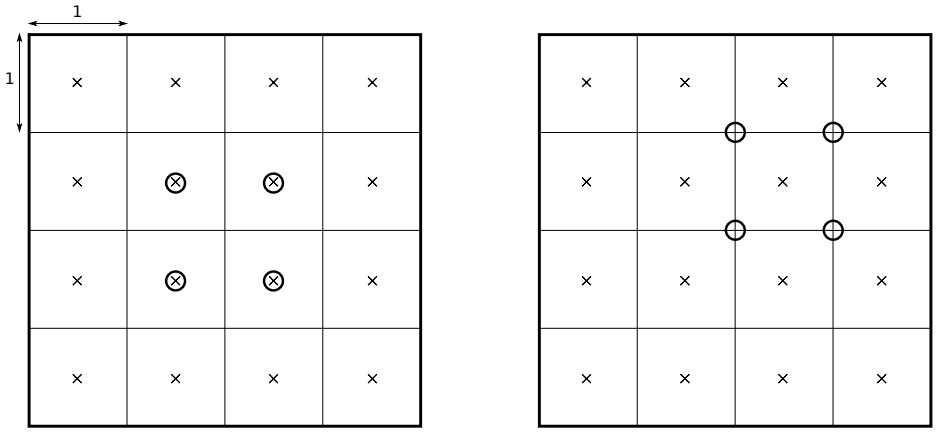

Figure 6.1: Particles translated relatively to the grid

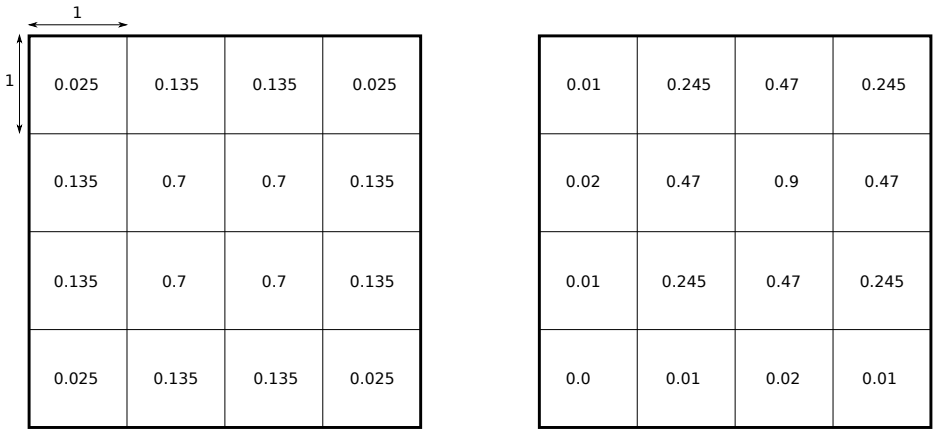

Figure 6.2: Density grids calculated for the setups from Figure 6.1

the particles using bilinear interpolation. This gives us a particle density of 0.7 for the left setup, compared to $\frac{1}{4} \cdot (0.245 + 0.47 + 0.47 + 0.9) = 0.52125$ for the right setup. This example shows that our input features contain an error depending on where, relative to the grid, compression occurs.

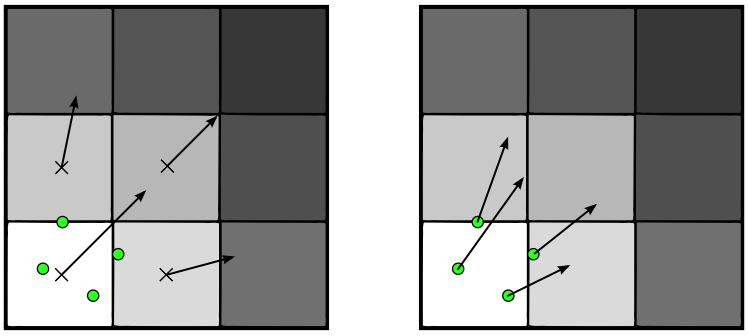

Figure 6.3: Interpolated pressure forces often do not point in opposite directions. Predicted pressure forces per grid cell are shown on the left, interpolated per-particle forces on the right.

In addition to this error in the input features, the grid-based approach also results in an issue when interpolating the predicted pressure forces of each grid cell back to the particles. As shown in Figure 6.3, multiple particles in a cell receive similar interpolated pressure forces. This shows that the interpolation approach has trouble pushing particles apart, i.e. in completely different directions. This error can be minimized by choosing the grid resolution so that there is one particle per cell on average. Overall, our method shows some inherent flaws which result from the grid-based replacement for neighbor-calculation. A method attempting to substitute per-particle neighbor data using a less computationally expensive alternative will inevitably contain some loss of accuracy. But the flaws described still mean that our approach is inaccurate not only due to the neural network's approximation error.

6.2 Performance

The main goal behind avoiding per-particle neighbor computation was to increase performance. In theory, our approach should be able to perform better than normal WCSPH, due to only computing cell-particle pairs as discussed in subsection 4.1.1. This should lead to a worst case performance linear with the number of particles instead of squared. In practice, this is not fully the case as the grid must be chosen so that there is one particle per cell at rest density. During the experiments in chapter 5 we recorded the model simulations' performance as detailed in Table 6.1.

	WCSPH		Model Simulation
Overall:	3.7342 ms	Overall:	14.8633 ms
Density:	0.5047 ms	Density Grid:	0.5588 ms
Pressure:	0.8880 ms	Pseudo Pressure Grid:	1.0118 ms
		Feature Vector:	0.3719 ms
Pressure Forces:	0.8666 ms	Pressure Force Prediction:	12.7080 ms
		Interpolation to Particles:	0.2128 ms
Neighbor Data:	1.4749 ms		

Table 6.1: Average time taken per operation each time step, measured using Python's process time [Pyt17]. Only operations necessary for pressure force calculation are listed. The dam break simulation from section 5.2 was used to collect this data.

Overall, our implementation did not perform better than normal WCSPH, actually taking longer per time step to compute. As Table 6.1 shows, our approach took 14.8633 ms on average to compute the pressure forces, while WCSPH only required 3.7342 ms. The vast majority of this time was spent on the prediction of the pressure forces, i.e. the evaluation of the neural network. There are several reasons why this is so expensive. First, we only evaluated the network on the CPU (see chapter 5). Since the evaluation of the network mainly consists of multiplications of matrices, a GPU implementation might significantly speed up this step. Additionally, we let the network predict the pressure forces for all grid cells. In this specific case, this means 64^2 network evaluations each time step. It would be much better to discard "empty" grid cells as we did during training data pre-processing (see subsection 4.3.1). This would result in a greatly reduced number of network evaluations per time step.
Apart from the very expensive prediction operation, the most time-consuming operations were the calculation of the density and pseudo pressure grids. Compared to their WCSPH counterparts, they took slightly longer to compute. However, they do not require the neighbor data calculation which is the most expensive operation of WCSPH. As explained in subsection 4.1.1, they substitute it by interpolating the particle's properties to the grid. Herein lies a distinct advantage of our approach. Using the gathering method from Figure 4.2 it is possible to parallelize this process, further speeding up the density and pseudo pressure grid operations. By contrast, The neighbor data computation of normal WCSPH is much less easy to parallelize. Here, each particle has a list of its neighbors and their distances from it. Building these lists

in parallel would result in race conditions as multiple particles attempt to write their information to the same list.
Lastly, the feature vector calculation of our approach could also be sped up using parallelization. The interpolation of the predicted pressure forces to the particles was parallelized in our implementation and, correspondingly, was quite fast. All in all, though our implementation of the method was not as efficient as hoped, there is a lot of potential for performance improvement which might lead to it performing better than normal WCSPH. This is especially relevant when considering that the density grid and pseudo grid calculations could well reach computation times matching density and pressure calculation in WCSPH without needing the additional neighbor data calculation step.

7 Conclusions

In this thesis we presented and investigated a particle-based approach to fluid simulation that uses a neural network to predict the pressure forces acting on each particle and avoids the expensive step of per-particle neighborhood calculation. Experiments showed that it can be used to believably simulate a fluid in many situations. There are however scenarios it still cannot deal with, especially those containing very low or very high pressures. Our approach proved to contain some flaws that impact the correctness of the simulations in addition to the neural network's approximation error. It also did not achieve better performance than normal WCSPH, even though this was primarily down to implementation and it still contains a lot of potential for improvement in this regard. Although the approach is therefore not viable in its current form, its development and examination lead to several findings which will be detailed in this section.

7.1 Prediction per grid cell

Our method predicts grid-based pressure forces from the grid-based input features. As explained in section 5.4 it was not possible to directly predict the particles' pressure forces, as the input features would contain insufficient variance compared to the expected output values. Interpolating the particles' pressure forces to a grid proved to be an effective way of reducing this variance, allowing the network to learn forces per grid cell. A possible alternative to this could be simply smoothing the particles' pressure forces, creating less varied ground truth values per particle.

7.2 Higher pressures in training simulations

As section 5.2 showed, during training it is important for the training simulations to contain higher pressures than the target simulation would if it was computed using normal WCSPH. The reason for this is that the neural network's prediction errors

can lead to temporarily higher compression of the fluid. If the network was never confronted with higher pressure values during training, it will be unable to correctly predict pressure forces based on them. This then leads to a higher prediction error, which in turn can lead to even higher compression and pressure values - the error amplifies.

7.3 Calculating expressive yet inexpensive input features

The main problem with a method that attempts to predict pressure forces for each particle is finding input features that are expressive enough, containing a sufficient amount of variance compared to the expected output, yet are less expensive than particle neighborhoods to calculate. This inevitably forces a tradeoff between correctness and performance: Computing complex input features that the network can learn from is expensive, easily computed input features tend to not contain enough information and variance.
A possible way of avoiding this would be to choose a different approach, where the input features are not based solely on the particle's immediate surroundings. Ladicky et al. [Lad+15] do something similar where they place a large set of box-shaped areas relative to each particle (see section 2.1).

8 Future Work

The results of this thesis provide two main avenues for follow-up research. The first is to further improve upon the approach detailed in this thesis. The other would be to find an alternative way of modeling the network's learning, specifically one that avoids the dilemma of calculating input features that contain enough information about a particle's immediate surroundings while simultaneously being inexpensive to compute (see section 7.3). Which direction is more promising largely depends on the performance improvements detailed in section 6.2. If implementing them improves our methods performance beyond that of WCSPH there is merit in further optimizing the approach - if not, an alternative should be sought.

8.1 Improving the approach

Our method can predict the pressure forces within a fluid without calculating per-particle neighbors. However, to remain stable, it currently needs artificial viscosity forces to be calculated. These are at the moment calculated in the normal WCSPH way (see [BT07]). Like the pressure force calculation this requires considering each particle's neighbors, their densities, masses and positions. Unlike pressure force calculation, it also incorporates the relative velocities of a particle and its neighbors. To transfer viscosity computation to our approach, a third grid would therefore have to be introduced for velocities, similar to those for density and pseudo pressure described in chapter 4.

An additional measure that should be taken to improve the fidelity of our method's predictions is to increase the scale on which training is performed. For our experiments, we used rotated and pressure multiplied resting simulations, resulting in around four million samples (see section 5.2). Though they contained surprisingly varied pressures which were sufficient to get an impression of our method's prediction capabilities, examining the histogram in Figure 5.12 shows that there was still comparatively little data for higher pressure values. Since other approaches such as that of Ladicky et al. took several days and up to 600 billion samples to train their regressor, it stands to

reason that our approach might also show improved results given more varied data on this scale.

8.2 Predicting all forces at once

Though there are several ways to improve the method presented in this thesis, our conclusions in section 7.3 suggest that it could be better to model the input features and learning differently. Specifically, the input features should not try to replace neighbor data by only providing information about a particle's immediate surroundings. Both Ladicky et al. [Lad+15] and Tompson et al. [Tom+16] incorporate information about a much larger neighborhood into their regressor's input features - either by evaluating the features on box-regions in a large area around the particle or by having the network predict the entire pressure field at once.
As such, we think an alteration to our method that does not predict each cell's pressure force in isolation is promising. The most obvious advantage to predicting the whole pressure force field at once would be that the resulting velocity field's divergence could be factored into the cost function as in [Tom+16]. It is also feasible that the additional spatial information provided to the neural network would allow it to learn from less varied input data per grid cell, possibly making it more efficient to calculate. The use of spatial structure could be further strengthened by utilizing convolutional neural networks [Nie17].
A possible issue with this approach would be that the trained model, due to predicting for all grid cells at once, would only be able to be used on a whole grid of fixed size. This would make the approach less flexible since a new model would have to be trained for different grid resolutions.

8.3 Concluding remarks

Though the exact method presented in this thesis proved to be less viable in its current form, it showed that it is possible to use a neural network to avoid the normally required neighbor data calculation in particle-based fluid simulation. It can thus be used as as foundation to further investigate the promising notion of using neural networks to increase the performance of smoothed particle hydrodynamics. Such data-driven approaches might be the key to achieving high fidelity fluid simulations in real time applications.

List of Figures

Bibliography

[Ben12] Y. Bengio. *Practical Recommendations for Gradient-Based Training of Deep Architectures*. 2012.

[BT07] M. Becker and M. Teschner. *Weakly compressible SPH for free surface flows*. University of Freiburg, 2007.

[GB10] X. Glorot and Y. Bengio. *Understanding the difficulty of training deep feedforward neural networks*. DIRO, Université de Montréal, Montréal, Québec, Canada, 2010.

[Goo16] Google. *TensorFlow*. *http://www.tensorflow.org/*. 2016.

[Har08] S. W. Harden. *Linear Data Smoothing in Python*. *http://www.swharden.com/wp/2008-11-17-linear-data-smoothing-in-python/*. 2008.

[Hay99] S. Haykin. *Neural Networks - A Comprehensive Foundation*. Prentice Hall International, Inc., 1999.

[Jos16] P. Joshi. *Understanding Xavier Initialization In Deep Neural Networks*. 2016.

[Lad+15] L. Ladicky et al. *Data-driven Fluid Simulations using Regression Forests*. ETH Zurich, 2015.

[Mat16] MatplotlibDevelopmentTeam. *Matplotlib*. *http://matplotlib.org/*. 2016.

[Mon05] J. J. Monaghan. *Smoothed particle hydrodynamics*. School of Mathematical Sciences, Monash University, Vic 3800, Australia, 2005.

[Nie17] M. Nielsen. *Neural Networks and Deep Learning*. 2017.

[Num16] NumPyDevelopers. *NumPy*. *http://www.numpy.org/*. 2016.

[Pyt17] PythonSoftwareFoundation. *Python Documentation*. *https://docs.python.org/3/library/time.html*. 2017.

[Rud16] S. Ruder. *An overview of gradient descent optimization algorithms*. 2016.

[SP09] B. Solenthaler and R. Pajarola. *Predictive-Corrective Incompressible SPH*. University of Zuerich, 2009.

[Tom+16] J. Tompson et al. *Accelerating Eulerian Fluid Simulation With Convolutional Networks*. New York University, 2016.

[TP16] N. Thuerey and T. Pfaff. *MantaFlow*. *http://mantaflow.com*. 2016.

[Wei17] X.-S. Wei. *Must Know Tips/Tricks in Deep Neural Networks*. 2017.