Neural Network in snow avalanche dynamics

MAJOR TECHNICAL PROJECT (DP 402P)

to be submitted by

Hrishikesh Sagar (B16029)

for the

END-SEMESTER
EVALUATION

under the supervision of

Dr. Gaurav Bhutani
Dr. Aditya Nigam

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MANDI

KAMAND-175005, INDIA

SEPTEMBER, 2019

Contents

ABSTRACT

Avalanche Simulations are computationally expensive. The Simulations can be done by using fluid solvers or using Discrete-Element-Modelling(DEM) both of which involve sequential calculations that are done on CPU. Our work aims to decrease the computation time of these simulation using neural networks in critical and computationally expensive parts of the Simulations. Our work is divided into two parts, the most recent work demonstrating the use of convolution neural networks to accelerate fluid solvers(Chapter 5 to 8), Where we have achieved successful fluid simulations using CNN, compressed feature-maps and gradient velocity function. The earlier work discusses the DEM methods computational limits and pitfalls of using Neural Networks for DEM(Chapter 2 to 4).

0.1 List of Symbols

m, m_p	Mass of the body
k	Spring constant
c, η	Damping coefficient
x, r , $\vec{r}$	Position vector
F, $\vec{F}$,f ,$\vec{f}$	Force on the body
v, $\vec{v}$	Velocity of the body
a, $\vec{a}$	Acceleration of the body
I	Moment of Inertia of the body
θ	Angular position of the body
U	Potential energy
O_{xyz}, O_{cm}	Coordinate origin, Centre of mass
ρ	Density
r_p	Base radius
μ	Friction coefficient
δ , $\vec{\delta}$	Overlap
ω , $\vec{\omega}$	Angular velocity
$\hat{n}$	Unit vector
u, u_t	Velocity field, velocity field at time t
$E(u)$	Encoder Network
$D(z)$	Decoder Network
$T(z)$	Dense Network

Introduction

1.1 Avalanche

Avalanche or snow-slide is an event that occurs when a mass of snow starts to move due to gravity (or any other trigger), this can trigger a chain reacting and soon a larger amount of snow can start to fall down the slope, accelerating rapidly. The artifacts and living beings in the path of this avalanche face drastic damages. Hence it is of importance that one should be able to study the effects of these avalanche's beforehand so as to take preventive measures. Traditionally Avalanche has been modelled using two method's continuum based fluid simulations and discrete-element-method(DEM), both of these methods have become computationally expensive when simulating for snow avalanches this work demonstrates the use of Neural Networks in Accelerating the Fluid Solvers, explores the use of Neural Networks in DEM and puts forward the limitations of using Deep Learning in DEM.

1.2 Avalanche fluid simulation

The technique to model snow avalanches in fluid simulations as described in [1] shows that the snow avalanches of the type powder which are more violent can be modelled using Navier Stokes equation of incompressible fluid. An incompressible model of two miscible can successfully be used to model the powder flow of avalanche by allowing mass diffusion between two phases according to Fick's law. The governing equations are discretized in a finite volume space. The flow of powder avalanche is similar to the flow of the water column in the classical dam break problem where the wall is removed and the flow of the water is investigated. The avalanche flow is a two fluid flow formed by the air and snow particles in suspension similar to the water flow in the dam break problem where the two fluids are water and air. On both the flows move under the effect of gravity. For simplicity the avalanche flow is considered as the mixture of Newtonian Fluid and the flow has a very high Reynolds number (Re $\sim 10^9$). The solving of Navier Stokes equation is computationally expensive and by using deep learning techniques we can generate faster simulations governed y Navier Stokes equation.

1.3 Avalanche and DEM

The current technique for studying granular flow is by using Discrete Element Method (DEM). It is inspired from the Molecular Dynamics approach and captures particle to particle interactions in an iterative process by using various numerical methods and is hence computationally intensive. It helps us in tracking the local density changes (flow viscosity) and the flow velocity as each particle interacts with one another.The advantage of modelling every particle in the system, however, comes at a huge computational cost. This demands the need of high performance computing cluster to deliver simulation results in a comparatively less time frame using partitioning and parallelization. Moreover, further computation is needed for post-processing, supporting the previous statement.

1.3.1 Neural Network in fluid solvers

Fluid simulators and solvers take a huge amount of time to calculate and iterate between timesteps, Many calculations done during this iteration procedure are not needed such as grid regions where no fluid is present. The time required between two timesteps often changes with the current state of the fluid solver which adds to the complexity. When the amount of fluid increases the complexity increases. The effect of these issues can be reduced by inculcating Neural networks within the simulation process. The time complexity of a Neural network is agnostic to the simulation state, having a proper loss function can lead the network to focus on grid elements with fluid present. Since the grid size produced by the fluid solvers are agnostic to the fluid amount it is easier to use convolutional Neural networks to complete the task.
Convolutional neural networks have shown great results in accelerating graphics and fluid solvers[2]. Convolutional networks are able to learn the representations of fluids in a given domain and reconstruct such simulations[3][4]. We use one such approach where we use two neural networks, one to generate fluid state from a compressed representation and another to iterate over these compressed architectures for each timestep.

1.3.2 Report organisation

The thesis can be understood in two different parts, Chapter 2 to 4 exploring the Avalanche Simulations through DEM methods using deep-learning to regress some of its aspects. Chapter 5 to Chapter 7. Explore the avalanche simulations using fluid simulations and deep-learning methods to accelerate the simulations itself.

- Chapter 1 gives an intro to the task at hand, providing a basic overview of the objectives and approaches followed.
- Chapter 2 Discusses the discrete element method and some mathematics behind it.
- Chapter 3 Analyses the time complexity of DEM simulations with respect to factors such as number of particles, simulations time, number of cores.

- Chapter 4 Analyzes the results of DEM simulations and proposes a preliminary model for Regressing the end results of a DEM simulation, this chapter also discusses the limits of using Deep Learning methods for DEM.
- Chapter 5 Fluid simulations how they work
- Chapter 6 This chapter discusses how fluid simulation data can be learnt by deep learning models, it discusses our used architecture in detail, the data processing involved, the loss used, deep learning model used with velocity field.
- Chapter 7 Discusses the results obtained after using a neural network for the fluid simulation for to types of problems: smoke simulation and dam break simulations
- Chapter 8 Explains a similar architecture of neural network used with Pressure field of the Fluid Simulations.
- Chapter 9 Gives pointers about the future aspects of the project.

Avalanche and DEM

2.1 Avalanches as granular flow

Snow can be modelled as a pack of granular particles. Whether an avalanche releases depends on terrain, meteorological, and snow-pack parameters. They are dangerous natural hazards in alpine regions. Traditionally, continuum models are used for modelling snow-debris flow where velocity distribution through the depth is assumed to be insignificant. The Discrete Element model works well for complex geometries and contours compared to the continuum shallow flow models and has shown a viable and even preferable alternative in recent studies.

2.2 Discrete Element Method

Discrete Element Method (DEM) is a numerical method which captures particle to particle interactions in a system with dynamics driven by external forces. These forces cause the particles to interact and move with respect to one another. Since, it is a numerical method, it is iterative in nature which means it is computationally intensive for the repeated operations.

A DEM model of two interacting particles is mathematically defined as a 2nd order non-linear system as shown in the Figure 2.1:

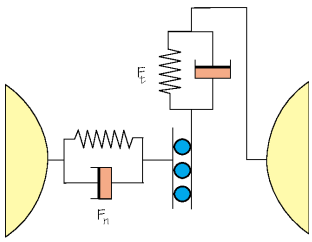

Figure 2.1: Model of interaction of two particles.

Equation 2.1 describes the motion with m being the mass of the particle, k, the spring constant, c, the

damping factor, and x, the position vector.

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0 \tag{2.1}$$

A basic DEM simulation contains five steps:

- **Initialise** - Read initial state of the particles
- **Ensembles and Interactions** - Calculate the forces acting on each particle based on the interaction with its neighbours.
- **Calculate forces** - Compute the acceleration of each particle.
- **Integration** - Obtain new velocity and the position of the particles after each time-step.
- **Analyse** - Compute magnitudes of interest and measure observables.
- Steps 2 to 5 are repeated for the required number of time-steps.

2.2.1 Force Calculation

The particles in a DEM simulation move due to the forces acting on them as governed by Newton's second law of motion given by the equation 2.2

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{r}}{dt^2} \tag{2.2}$$

$$m_i\vec{\ddot{x}_i} = m_i\vec{g} + \sum_j \vec{F_{ij}} \tag{2.3}$$

$$I_i\vec{\theta_i} = \sum_j \vec{r_{ij}} \cdot \vec{F_{ij}} \tag{2.4}$$

where i$\neq$ j.

Calculating the potential energy between the individual particles helps in determining the particle interactions that are described by the inter-particle forces. The sum of the potential energy associated with all the particle interactions gives the total energy of the system. The forces acting between a pair of particles are given by:

$$\vec{F_i} = \Delta U \cdot \vec{r_{ij}} \tag{2.5}$$

2.2.2 Interaction Between Particles

A particle can be defined as a small localised object with physical properties like volume and mass with two volume boundary surfaces. One is the physical surface of the particle S_b and the other is the virtual or effect surface boundary which affects the neighbours even before physical contact (like in electrostatic conditions).

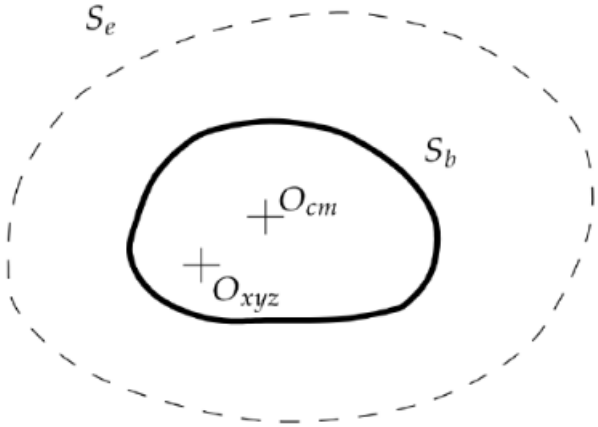

Figure 2.2: Definition of a computation particle

The use of complex geometry increase the computation time drastically so these boundaries are defined as simple spheres with the same coordinate origin and centre of mass (O_{xyz} and O_{cm} respectively).
The mass of the particle P is thus given by

$$m_p = \frac{4}{3}\pi\rho r^3 \tag{2.6}$$

where r_p is the base radius and its density is ρ .
There are three collision states for two particles as show in Figure 2.3

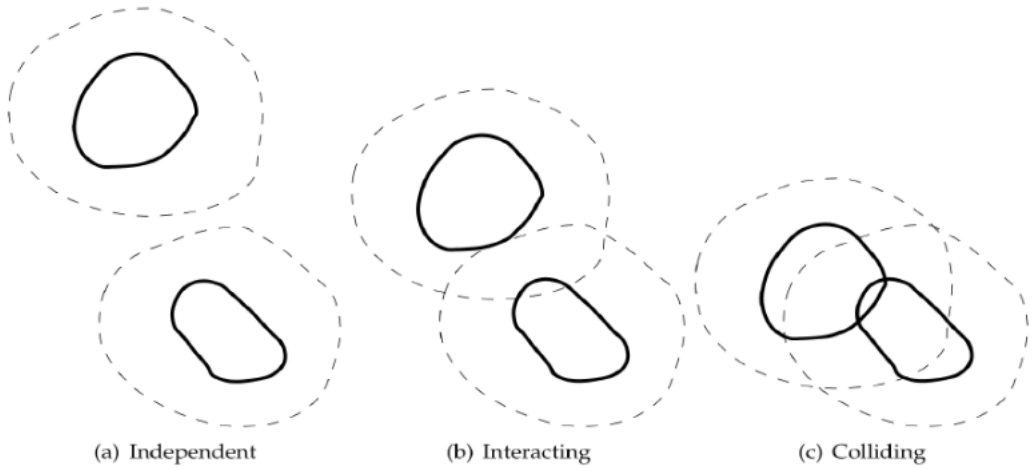

Figure 2.3: Different possible collision states

In Figure 2.3(a), no force is exerted by the two particles on each other, hence, they are independent of each other.
In Figure 2.3(b), the effect surface boundaries of the particles interact with each other although no physical contact occurs. This is analogous to two charged particles exhibiting an attractive or a repulsive force on each

other even before they are in con- tact. This interaction may also take place when the effect surface boundary interacts with a physical boundary.

In Figure 2.3(c), the particles come in physical contact with each other and are subjected to the reactionary forces during this collision state. These forces may also be coupled with the forces due to the interaction of the effect surface boundaries or the effect surface boundaries with the physical boundary.

When two particles i and j are in the state of collision, their relative position is described as

$$\vec{r_{ij}} = \vec{r_i} - \vec{r_j} \tag{2.7}$$

and their relative velocities as its time derivative.

$$\vec{v_{ij}} = \vec{v_i} - \vec{v_j} \tag{2.8}$$

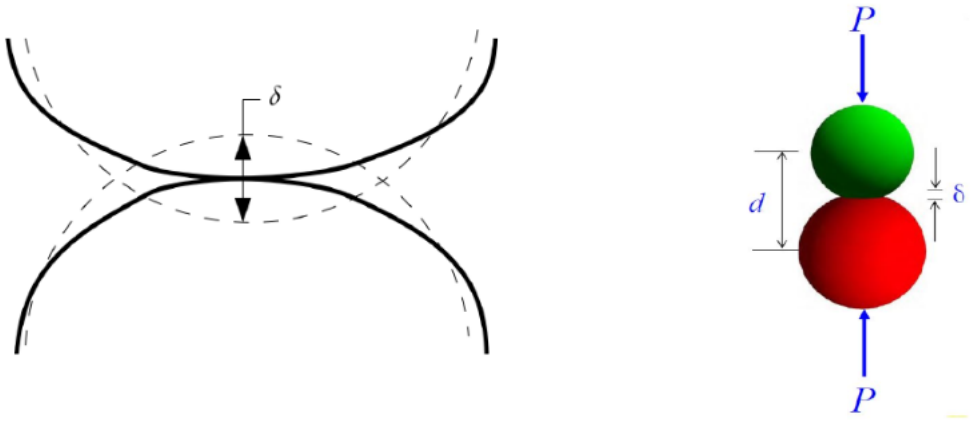

Figure 2.4: Contact between two spherical particles

The contact model given by Cundall and Strack [5] as shown in Figure 2.4 is used to define the interaction of two particles upon collision and is affected by stiffness (k), damping coefficient (η) and friction coefficient (μ). When particle i is in contact with particle j, the normal component of the contact force is given by the sum of forces due to the spring and dash-pot as [6]

$$\vec{f_{n_{ij}}} = (-k_n \delta_{n_{ij}} - n_n v_{n_{ij}}) \hat{n_{ij}} \tag{2.9}$$

where $\delta_{n_{ij}}$ is the normal overlap, $\vec{v_{n_{ij}}}$ is the normal component relative velocity, and $\hat{n_{ij}}$ is the unit vector from centre of particle i to j.

Similarly, the tangential component is given by

$$\vec{f_{t_{ij}}} = (-k_t \vec{\delta_{t_{ij}}} - \eta_t \vec{v_{s_{ij}}}) \tag{2.10}$$

$$\vec{v_{s_{ij}}} = \vec{v_{r_{ij}}} - (\vec{v_{n_{ij}}} + r_s(\vec{\omega_i} + \vec{\omega_j})) \times \hat{n_{ij}} \tag{2.11}$$

where $\delta_{t_{ij}}$is the tangential overlap, $\vec{v_{s_{ij}}}$ is the slip velocity, $\vec{v_{r_{ij}}}$ is the relative velocity of the two particles, r_s is the radius of the sphere, and ω_i and ω_j are angular velocities of said particles.

The resultant frictional force that resists sliding is given as the sum of these two forces by Hertz theory[7].

$$\vec{f} = \vec{f_{n_{ij}}} + \vec{f_{t_{ij}}} \tag{2.12}$$

These forces act when the distance between the centres of these two particles d is less than the sum of their radii r_i and r_j . Otherwise, there is no force.

Heavy-duty discrete element modelling of dry snow avalanches for varying slope angles: A computational study

3.1 Introduction

DEM simulations are costly, an extensive study of the computation time was done and the results were contributed to a paper titled: *Heavy-duty discrete element modelling of dry snow avalanches for varying slope angles: A computational study*, which was submitted 7^{th} International Congress on Computational Mechanics and Simulation (ICCMS 2019) and is currently under review.

3.2 Motivation

This study investigates the feasibility of using high-performance computing (HPC) cluster to model dry snow avalanches using DEM in LIGGGHTS, an open-source DEM solver written in C++ and parallelised using OpenMPI. Simulations for different slope angles, different number of particles, different width of slope, different number of cores and different number of simulation time were carried out. The forces delivered to a rigid wall and nature of the splash were studied. The results are physically consistent and can help us to design better impact defence structures from the force distribution information. This computational modelling methodology can also be extended for industries dealing with granular flows, such as food, pharmaceutical and mining.

3.3 Methodology

All the simulations were done in LIGGGHTS and other softwares were used during the whole process. The pipeline can be understood by figure3.5. All these software were controlled by creating a python wrapper to run scripts for each of them. The setup is elaborated further

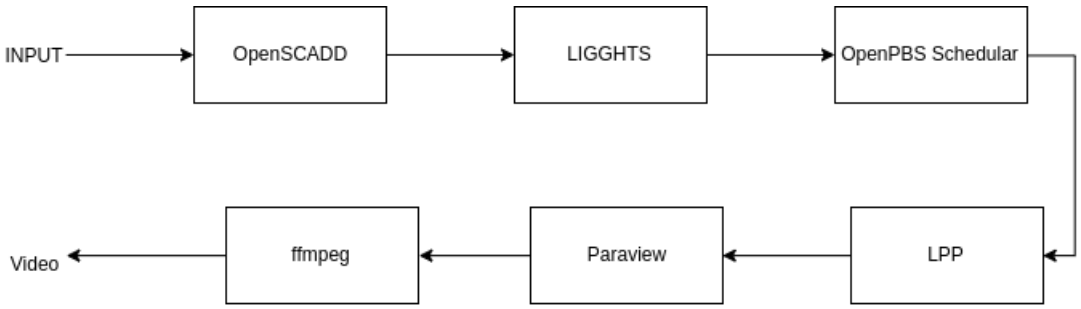

Figure 3.5: Flowchart of the simulation

3.3.1 OpenSCAD

OpenSCAD is an open-source command based computer-aided design (CAD) software.OpenSCAD uses its own feature rich language to generate and transform geometries and takes**.scad** file containing command as input. A python library SolidPython makes easier to make and modify input scripts.OpenSCAD is used to generate slope for the system.

3.3.2 LIGGHTS

For generating the DEM simulation ,LIGGGHTS has the following procedure:

- **Initialisation of the solver**- Here in the input script of LIGGGHTS, the information about the type of particle morphology , boundaries,and interaction between the processes is defined.
- **Domain**- Here we define the domain of the system, which includes region,bounding box, distance between a particle and its neighbour, and neighbour modification time delay.
- **Material Properties**- The material properties of the particle like young modulus(γ), coefficient of restitution(e),coefficient of friction (μ)and poissons ratio.
- **Global Parameters** - Here we define , the parameters like timestep , gravity , and other external forces that are actin g on the system.
- **Boundaries**- Grannular Boundaries parameter are defined here. The parameters include x ,y and z coordinates of the bounding box.
- **Particle Distribution**- Particle geometry and their distribution is defined here.
- **Mesh** - This part of the input script include the type of slope we want to add to the system. The slope properties are defined in the the file of the **.stl** format.

3.3.3 LPP

The output files from LIGGGHTS are in **.atom** format. To visualise the output , this **.atom** file is post-processed to **.vtk** using the utility called LIGGHTS Post Processing(lpp). The lpp works on pizza.py. The output of lpp generates two types of **.vtk** files, one is for the particles/atom and the other file contains the information of the system boundaries.

3.3.4 Paraview

ParaView is an open-source multiple-platform application for interactive, scientific visualization. It has a client–server architecture to facilitate remote visualization of datasets, and generates level of detail models to maintain interactive frame rates for large datasets.

To visualise DEM simulation on Paraview, **pvserver** is runned on HPC. With the help of port-forwarding Paraview is running on client machine. Data on the server is loaded on the pvserver by opening the files ***.vtk** and ***.stl** from the OpenDialog of the Paraview.The animation of the simulation can be saved as a series of ***.png** images by selecting File $\longrightarrow$ Create Animation. The timestamped images of the simulation are saved on the directly user's machine. The images can be converted to a **.mp4** video using **ffmpeg** command where the animation speed can be adjusted by tweaking frame-rate of the video.

3.4 Results and Discussion: Computational complexity analysis

Heavy-duty simulations are analysed for the computational complexity in this section. The computation time for DEM simulations depends upon various factors, the number of particles being a major one. For analysing computational performance of the avalanche simulations in the preset work, certain parameters were fixed as given already in Table 3.2, and varying other parameter over a given range as shown in Table 3.1.

Table 3.1: Range of parameters for computational complexity analysis

Parameter	Start	End	Unit
Number of CPU threads	2	24	-
Number of particles	10^4	10^7	-
Number of iterations	10^4	10^5	-
Length of slope	100	10^4	mm
Width of slope	100	1000	mm
Angle of slope	10	60	deg

Table 3.2: Material Parameters

Parameter	Value	Unit
Shape	Spherical	-
diameter	5	mm
density	500	kg/m^3
Coefficient of restitution	0.89	-
coefficient of friction	0.1	-
Young's modulus	$5x10^6$	-
Poisson Ratio	0.32	-
Gravity Acceleration	9.81	ms^{-2}

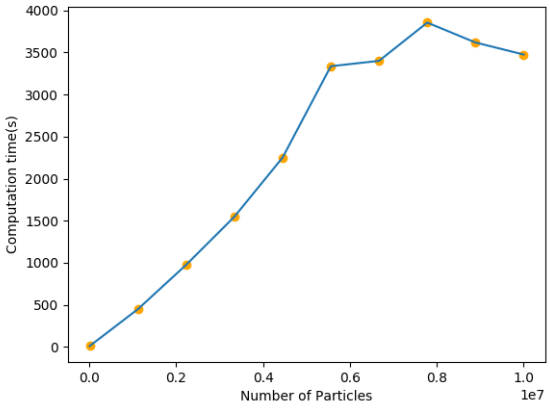

Figure 3.6: Computation time vs Number of particles.

Fig 3.6 demonstrates the computation time for a range of number of particles between 10^4 to 10^7. All simulations used 10^4 iterations, with a time-step size of 10^-5s, i.e. a total simulation time of $0.1s$. The general trend suggests that computation time increases with the number of particles, however there is a plateau noticed as the number reaches the right side of the range. This plateau effect can be attributed to a steadiness that is reached in the number of collisions.

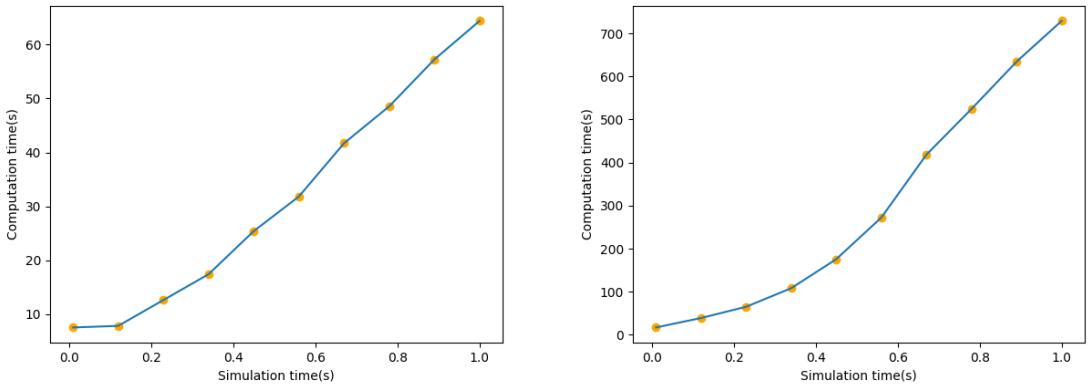

Figure 3.7: Computation time vs simulation time for (a) 10^4 particles and (b) 10^5 particles.

The avalanche simulations were allowed to continue for different duration and the computation time is plotted in Fig 3.7. Time-step in this case is $10^{-5}s$, i.e. 10^5 iterations represent 1 second of simulation. The computational time with simulation time (or number of iterations) increases drastically when the number of particles is increased 10 fold (10^4 to 10^5), which is clearly visible in Fig 3.7. This increased dependence can be related to number of collisions, which increase as the iterations increase.

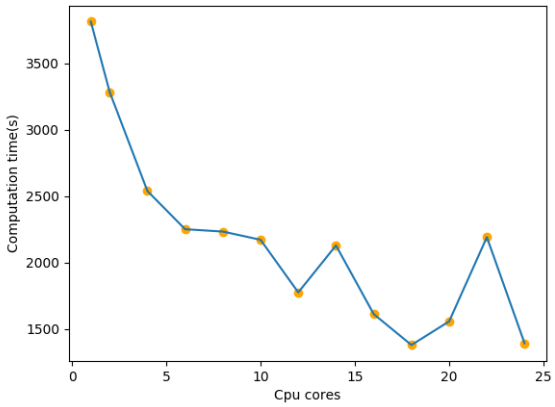

Figure 3.8: Computation time vs number of CPU cores

A simulation with 10^4 particles and 10^5 iterations (1s simulation time) was conducted using different number of CPU cores (or threads) and the computation time was captured, as displayed in Fig 3.8. One would expect the processing time to halve with doubling the number of cores, however, that was not noticed in the present granular flow DEM simulations. This can be attributed to the extent of parallelisation of the code and to the latency in communication between the cores.

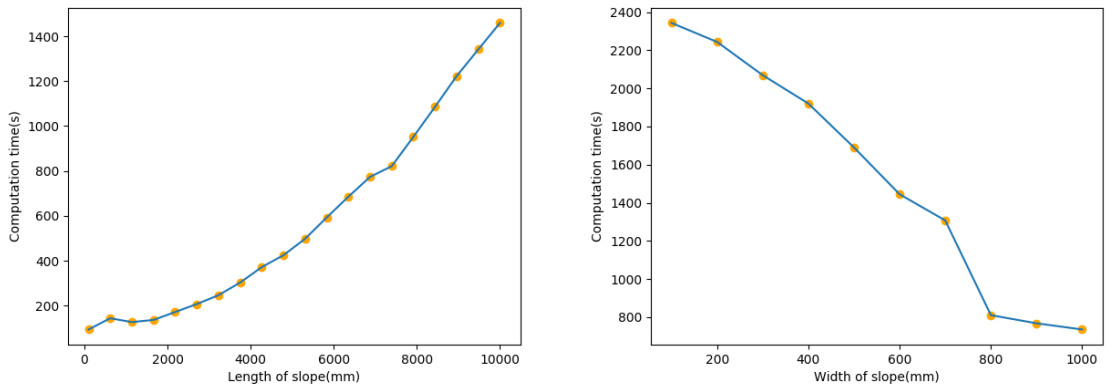

Figure 3.9: Computation time for different slope parameters for various (a) slope lengths and (b) slope widths.

Interestingly, the computation time for avalanche simulation also depends on the slope geometrical parameters such as its length and width. These factors directly affect the number of collisions, thereby affecting the number of equations to be solved and hence the computation time, as presented in Fig 3.9.

Data Analysis and Deep Learning and Pitfalls

4.1 Identification of Parameters

The data generated by a DEM simulation is stored in the ***.dump** format for each time step contains the x, y, z co-ordinates, velocities in x, y and z direction, force in x, y, and z direction, and the angular velocity in x, y, and z direction for each particle along with the unique id of the particle given by the LIGGHTS. Our main aim is to find the velocity of the front of the snow avalanche , the force acting on the front of the flow of particles and how much time does the front of the flow of particles takes to reach the end of the slope.The data files vary on the parameters of width, length, and angle of the slope and the number of particles in the system.So the input parameters are the length, width, angle of the slope and the number of particles.
In the ***.dump** file of the DEM for each timestep, we only want to study the front of the avalanche, so we take the maximum of all the x-coordinates for each timestep as the input parameter length.

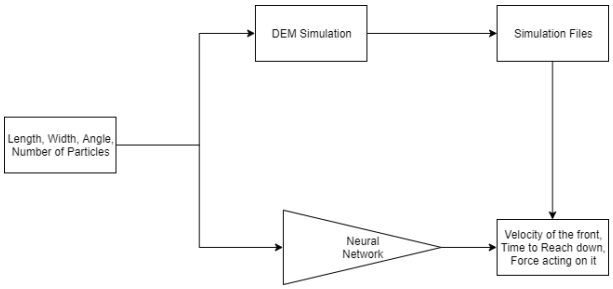

Figure 4.10: Workflow of Deep Learning Model

4.2 Analysis of the Length vs Time

4.2.1 Width of the Slope

Only the width is varied keeping the number of particles and the angle constant. The Figure 4.11 below shows the variation of the length vs time , for different widths.

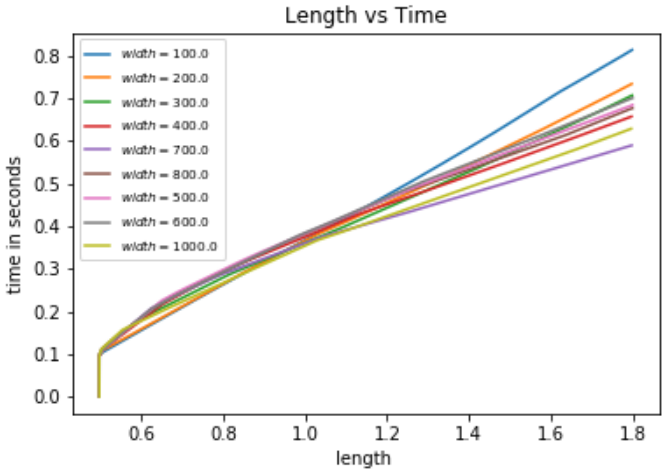

Figure 4.11: Width Variation

The above Figure 4.11 shows that the length has the linear dependence with time.

4.2.2 Angle of the Slope

Only the angle is varied keeping the number of particles and the width of the slope constant.The fig below shows the variation of the length vs time , for different angles.

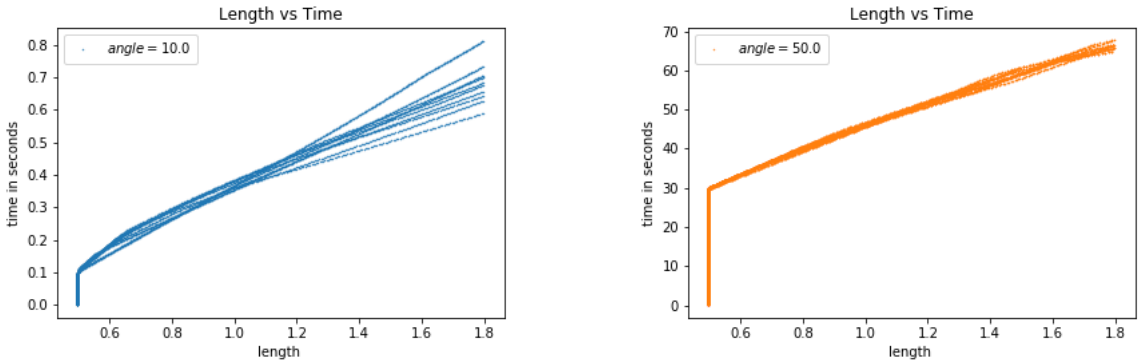

Figure 4.12: Angle Variation

The above Figure 4.12 shows that the length has the linear dependence with time.

4.3 Deep Learning

In order to skip the computation time and data processing as shown in Figure 4.10 and get our required features such as time taken to cover a distance, velocity of the front, force on wall an deep learning based approach was proposed.

4.4 Model

Owing to the linear trend of time required to cover a distance(Figure4.11 and Figure4.12) w.r.t change in angle and width a simple Multi-layer-perceptron was proposed with the following details.

Table 4.3: Model Parameters

Parameter	Value
Hidden Layers	1
Number of nodes	7
Activations	reLu
Loss Function	MSE
Learning Rate	0.001

As the model had to learn these linear trends the model learned quite easily following were the training and validation loss represented by Figure 4.13

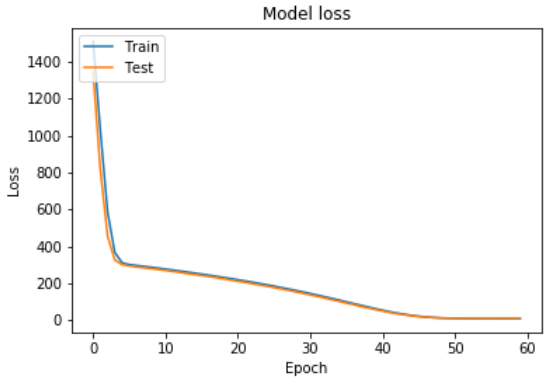

Figure 4.13: Training and Validation loss of the model

4.5 Limitations of Deep Learning for accelerating DEM simulations

4.5.1 Information retainment and ambiguity w.r.t to inputs

The workflow mentioned in 4.10 largely relies on the values of length, width, angle and number of particles to predict the values of the velocity at the front, but these 4 inputs to the network itself can create ambiguity for velocity values. Consider there exists a function $G(L, W, n, \theta)$ that the model is tries to fit/regress, but at the same time one combination of values of L,W,n,θ can have multiple values of v_f(imagine different orientations of particles at t=0), hence the existence of such a function put quite a limitations on the kind of simulation we are studying.

4.5.2 Limitations due to high number of Particles

If one were to use Neural networks to calculate the velocity of each and every particle, at some point in the network the number of variables would have a direct relation to the number of particles, which when increased would also explode the number of variable in the network. These limitations could be overcome if the input variable(s) was independent of number of particles(or amount of fluid) or the orientation to the particles(or fluid) then one could use Deep learning techniques to accelerate the simulations.

Fluid Simulations using Navier Stocks Equation

5.1 Navier Stocks Equation

5.1.1 Fluid Model as Velocity Flow

In our work, we model the fluid flow as a *velocity field*. The velocity field is a vector that defines the motion of fluids at a set of points in space. In the whole discussion **u** represents the **velocity field** vector. For the simulation, the **u** is evolved over the time and marker particles are moved through space governed by **u**. The velocities are stored at the discrete grid points and by the use of methods of interpolation velocities between the samples are found. The rules to calculate **u** over the time is based on the Navier-Stocks equations.

5.1.2 The Navier-Stocks equation for Incompressible Flow

The Navier Stokes equations are two set of differential equations that can be used to define any flow in this universe. The Navier-Stocks equation describe the flow of incompressible fluid using a system of partial differential equations. These partial differential equations are of the momentum conservation and the continuity equation. The fluid particles are handled by grouping them and represented by the field of velocity vectors $\vec{\mathbf{u}}$ and the pressure field $\boldsymbol{p}$ at time t. Because the fluid is incompressible the density ρ is assumed to be constant. The body forces like gravity $\vec{g}$ governs the behaviour of the fluid.

The continuity equation is

$$\boldsymbol{\nabla} \cdot \vec{u} = 0 \tag{5.13}$$

The continuity equation simply tells the equality between the amount of fluid flowing in any volume of space to the amount of fluid out of that volume.

The equation which specifies **u** change over the time is given by

$$\frac{\partial u}{\partial t} = -(\boldsymbol{\nabla} \cdot \mathbf{u})\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{F} \tag{5.14}$$

where v represents the viscosity of the fluid, ∇^2 represents the Laplacian operator.

This equation is represents the motion of fluid through the space along with any internal and external forces acting on the fluid[8].

The different terms of the equation is describes as follows:

- $\frac{\partial \mathbf{u}}{\partial t}$: This terms represents the derivative of velocity with respect to time and is calculated at all points in the grid during each step of simulation.
- $-(\boldsymbol{\nabla} \cdot \mathbf{u})\mathbf{u}$: This represents the convection term and arises due to the conservation of momentum. Since the velocity of the fluid is sampled at the fixed locations in the space, hence the momentum of the fluid must be moved or "convected" through the space along with the fluid itself.
- $-\frac{1}{\rho}\nabla p$: This represents the pressure term which captures the forces generated by the pressure differences inside the fluid. ρ represents the density of the fluid. The pressure term will always be coupled with the (5.13) so that the flow remains incompressible.
- $\nu\nabla^2\mathbf{u}$: This represents the viscosity term. The viscosity means the internal friction forces which acts between the fluid and resist the flow of fluid. In thick fluid, like honey viscosity **v** is very high and in water **v** is low.
- **F**: shows the external forces like gravity or contact forces with any objects acting on the fluid.

5.1.3 The MAC Grid

The Marker and Cell (MAC) grid first published in [9], has become one of the most popular ways for fluid simulation in computer graphics. The MAC grid method is used to discretize the velocity and pressure fields used in Navier Stocks equations.

The MAC grid methods uses the cubical cells with a width of h to discretize the space. Each cell has a pressure p defined at its center and the velocity $\mathbf{u} = (u_x, u_y, u_z)$ placed at the centres of each face of the cell, like u_x towards the -x axis, similarly u_y towards the -y axis and the u_z towards the -z axis. Along with that a set of marker particles (points in space) is used to represent the fluid volume in the simulation space. As the simulation progress in time, the markers particles are moved through the velocity field and then used to determine which cells contains the fluid.

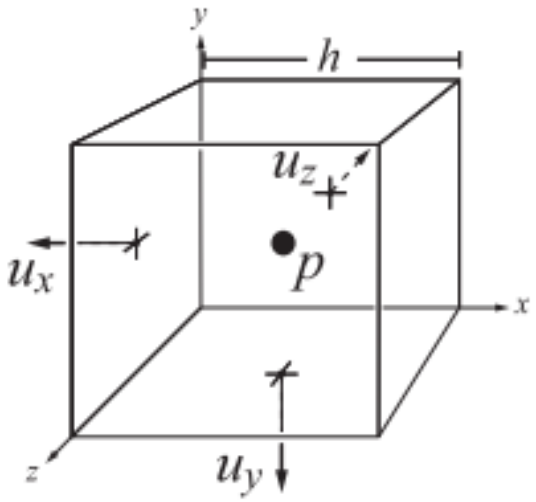

Figure 5.14: A MAC grid cell. Velocity components u_x, u_y, u_z are stored on minimal faces of the cell and Pressure p is stored at the cell center

5.1.4 Mantaflow

Mantaflow is an open-source framework used for fluid simulation research in Computer Graphics and Machine Learning[10]. It has a parallelized C++ solver core, python scene definition interface and plugin system that allows quick prototyping and testing of new algorithms. A wide range of Navier-Stokes solver variants are included to simulate the flow. Our simulations are generated using the Mantaflow framework.
Some features of the mantaflow are as follows-

- Eulerian simulation using MAC Grids, PCG pressure solver and MacCormack advection
- Flexible particle systems
- FLIP simulations for liquids
- Surface mesh tracking
- Free surface simulations with levelsets, fast marching

5.2 Dam-Break Problem

A dam-break problem is the classical problem in which when the wall separating the two sides of water is removed a shock wave occurs and propagates. The behaviour of the shock wave is investigated with respect to the depth of the water and the wave speed. The snow avalanche problem is similar to dam-break problem. Using the Navier Stokes equation the dam-break simulation is generation in the mantaflow framework.

5.2.1 Problem Setup

The dam-break model used is a tank of Length L=3.22 m and height H=2.0 m and the water column has the dimensions of L=1.2 m and H=0.6 m.[11] To train the deep learning network different sizes of water-column is used. Fig 5.15 shows the schematic setup.

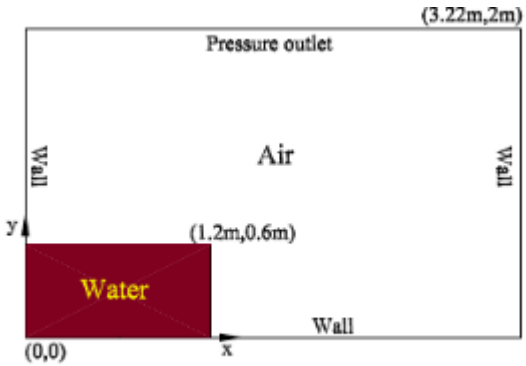

Figure 5.15: General layout of dam break problem

5.2.2 Mantaflow Simulation of Dam-Break

A two-dimensional dam-break simulation is created using the mantaflow framework.

The simulation creates a velocity field using the MAC grid, the pressure field using RealGrid, and uses the MAC grid algorithm to solve the velocity fields and the pressure fields and the FLIP algorithm [12] to update the velocity. The Fig 5.16 shows the dam- break simulation generated by using mantaflow framework.

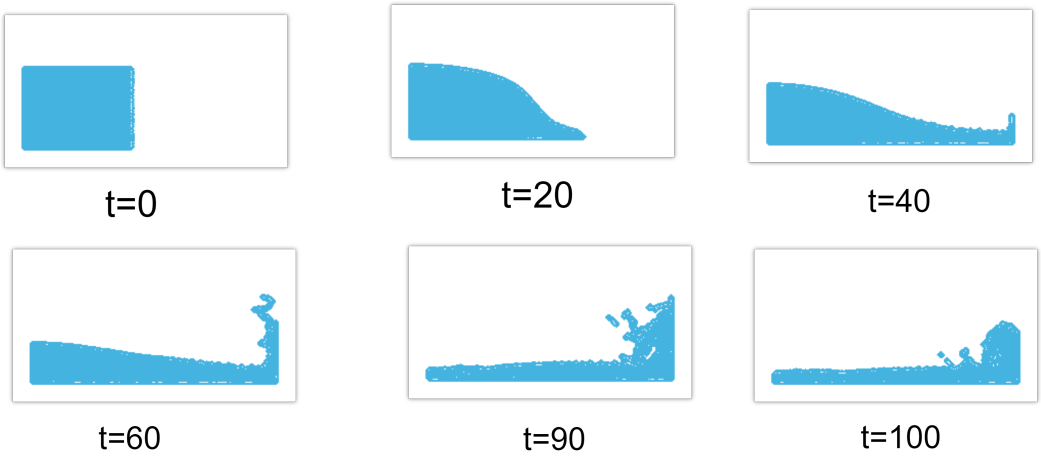

Figure 5.16: Dam Break Simulation using Mantaflow framework

Algorithm 1 Dam break Simulation in Mantaflow

1: $resx \leftarrow 256$

2: $resy \leftarrow 256$

3: $gs \leftarrow (resx, resy, 1)$

4: Create a solver $s \leftarrow Solver(gridSize = gs, dim = 2)$

5: Set the timestep $s.timestep \leftarrow 0.8$

6: $flags \leftarrow s.create(FlagGrid)$

7: Create MACgrid for velocity field $vel \leftarrow s.create(MACGrid)$

8: Create RealGrid for pressure $pressure \leftarrow s.create(RealGrid)$

9: Create the Particle types $pp \leftarrow s.create(BasicParticleSystem)$

10: Create the particle velocities $pVel \leftarrow pp.create(PdataVec3)$

11: Define the particle Index $pindex \leftarrow s.create(ParticleIndexSystem)$

12: Define the dam water column $fluidBasin \leftarrow Box(parent = s, center = center, size = size)$

13: Compute the levelset of fluidbasin $phi \leftarrow fluidBasin.computeLevelset()$

14: Update the Flags $flags.updateFromLevelset(phi)$

15: Sample the paricles in the grid sampleLevelsetWithParticles()

16: Set the initial velocity of particles mapGridToPartsVec3(source=vel, parts=pp, target=pVel)

17: **for** each timestep **do**

18: Advect the particles velocities in the grid pp.advectInGrid(vel=vel)

19: Mark the particle velocities using MAC mapPartsToMAC(vel=vel,partVel=pVel)

20: Extrapolate the velocities extrapolateMACFromWeight(vel=vel, distance=2)

21: mark the cells which contain fluid markFluidCells(parts=pp)

22: Create a levelset surface to resample paricles

a: gridParticleIndex()

b: averagedParticleLevelset()

23: addGravity(vel=vel, gravity=gravity)

24: solvePressure(vel=vel, pressure=pressure, phi=phi)

25: extrapolate the MAC grid velocity field extrapolateMACSimple(flags=flags, vel=vel)

26: update the velocities using FLIP algorithm flipVelocityUpdate(vel)

27: **end for**

Generative Models for fluid Simulation

6.1 Neural Networks for Fluid Simulation

Fluid simulators and solvers take a huge amount of time to calculate and iterate between timesteps, Many calculations done during this iteration procedure are not needed such as grid regions where no fluid is present. The time required between two timesteps often changes with the current state of the fluid solver which adds to the complexity. When the amount of fluid increases the complexity increases. The effect of these issues can be reduced by inculcating Neural networks within the simulation process. The time complexity of a Neural network is agnostic to the simulation state, having a proper loss function can lead the network to focus on grid elements with fluid present. Since the grid size produced by the fluid solvers are agnostic to the fluid amount it is easier to use convolutional Neural networks to complete the task.

Convolutional neural networks have shown great results in accelerating graphics and fluid solvers[2]. Convolutional networks are able to learn the representations of fluids in a given domain and reconstruct such simulations[3][4]. We use one such approach where we use two neural networks, one to generate fluid state from a compressed representation and another to iterate over these compressed architectures for each time-step.

6.2 Generative Model

A typical fluid solver follows the steps shown in algorithm 2, Step 1 and step 2 are constant w.r.t to the grid domains. The most complex step in the solvers's computation is step 3 where velocity field u, is calculated by doing various calculations on the grid. The acceleration of this particular step would implicate many-fold acceleration for the simulation process. Our work does step 3 using a combination of two neural-networks, an autoencoder network and a dense network.

The autoencoder network is responsible to compress the velocity field into a smaller representation and also decompress,the Dense network is responsible to generate these compressed features for next time-step. We also use a loss function that uses velocity field gradient information (∇u) to penalize incorrect velocity field data.

Algorithm 2 Basic Fluid Dynamics Algorithm

1: Calculate the simulation time step, Δt

2: Update the grid based on the marker particles

3: Advance the velocity field **u**

a: Apply convection using a backwards particle trace

b: Apply external forces.

c: Apply viscosity.

d: Calculate pressure to satisfy $\nabla \cdot u = 0$

e: Apply the pressure.

f: Extrapolate fluid velocities into the buffer zone

g: Set solid cell velocities.

4: Move the particles through **u** for Δt time

a: If Δt extends beyond the next displayed frame time then

Advance particles to next displayed frame

Display frame and Repeat step 4a.

b: Move particles through **u** for the remainder of Δt

6.2.1 Auto-Encoder

The encoder is responsible to convert the velocity u ($R^{H\times W\times V}$) to a compressed form c(R^n).
$E^c(u) : R^{H\times W\times V} \rightarrow R^n$ and a Decoder architecture is responsible to reconstruct velocity field i.e. the to convert the compressed features back to a velocity field $D^+(z) : R^n \rightarrow R^{H\times W\times V}$. The compressed form is used by the Dense network to generate compressed features for the next time-step. The encoder is made up of the following blocks as shown in Fig 6.17

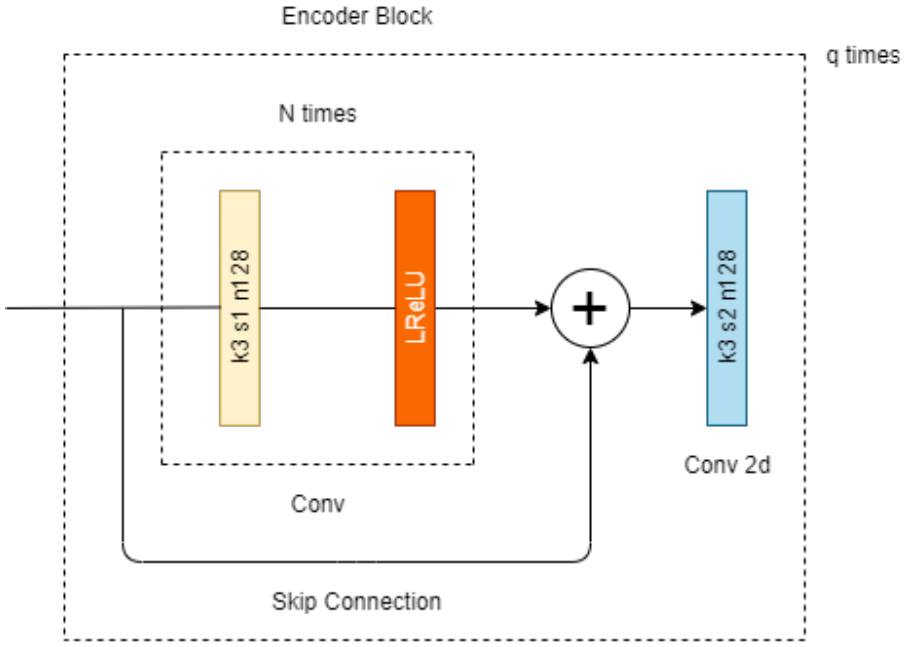

Figure 6.17: Encoder Block

Fig 6.17 shows one block of the encoder, this block has N convolutional layers with kernel size (3x3) and stride 1 and f number of filters per layer, along with same padding, the using leaky ReLu activation. After every such block there is a 2d convolution with stride 2. There are q such blocks with skip connections between every one of them, the feature map m after these blocks is of size: $\frac{H}{2^q} \times \frac{W}{2^q} \times 128$.

Then a dense layer is applied to bring it down to N features z. The Decoder is responsible to convert this N features back to the original velocity field, $D^+(z) : R^n \rightarrow R^{H \times W \times V}$. The decoder or starts with a dense layer to convert N features to m feature map $\frac{H}{2^q} \times \frac{W}{2^q} \times 128$. The Decoder is made up of the following blocks as shown in Fig 6.18

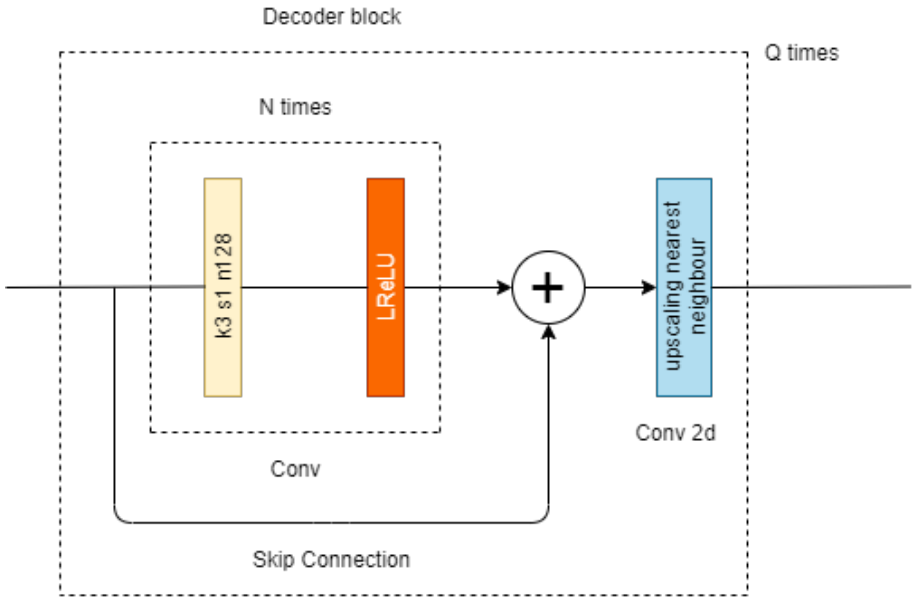

Figure 6.18: Decoder Network

It is quite similar to the encoder block Fig 6.17, this block has N convolutional layers with kernel size (3x3) and stride 1 and f number of filters per layer, along with same padding, the using leaky relu activation. After every such block there is an up-sampling layer of tf.image.resize_nearest_neighbor(). There are q such blocks with skip connections between every one of them, the last layer has activation sigmoid. The output of the decoder is the original velocity field.

6.2.2 Dense Network

The dense network is responsible for iterating once over time step it does so by advecting our compressed feature map z and out putting Δz, T(**z**):$R^n \rightarrow R^n$, where $\Delta z = z_{t+1} - z_t$. It is an MLP with two hidden dense layers each of them with batch normalization (Exponential Linear Unit activation) and a dropout of 0.1, The first dense layer has 1024 nodes and the second 512 the output is Δz. This Δz is used to calculate z_{t+1}.

6.2.3 Loss Function

The final output of the architecture is the velocity field u_{t+1}, a preliminary part of loss function is L1 loss between the u_{pred} and u_{gt} Hence:

$$L_1(u) = \mid u_{pred} - u_{gt} \mid \tag{6.15}$$

But minimizing the velocity field using only 6.15 loss can lead to problems,even though the predicted velocity values are near to the ground truth, the gradients of the velocities may not match, for example refer image:

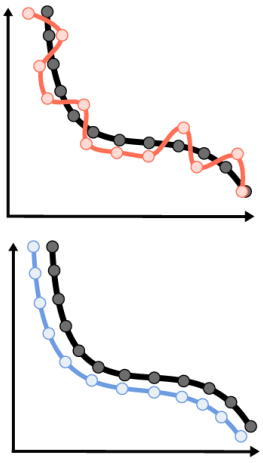

Figure 6.19: Velocity field with same values of equation 6.15

In equation 6.19 the black line is the true velocity the red and blue are the predicted velocities, In both cases the losses are same but the gradient of the predicted velocity in bottom case in same as that of ground truth, hence another part of the loss function is L1 loss between the gradients of velocities.

$$L_1(u) = \mid \nabla \cdot u_{pred} - \nabla \cdot u_{gt} \mid \tag{6.16}$$

Equation 6.16 ensures that the gradient of the predicted velocity fields and ground truth remain the same. Combining the two the final loss equation becomes:

$$L(u) = \lambda_u \mid u_{pred} - u_{gt} \mid + \lambda_{\nabla u} \mid \nabla \cdot u_{pred} - \nabla \cdot u_{gt} \mid \tag{6.17}$$

The autoencoder is trained first using the loss given in equation 6.17, along with adam optimizer, The Encoder is then used to create a dataset of compressed feature maps. The Dense network is then trained upon this feature map using mse loss and Adam optimizer.

6.3 Velocity advection using Neural Networks and Inculcation with solvers

Algorithm 3 explains the working of the architecture to calculate the velocity fields. At t=0 velocity field is used to get z(t=0) this z is used to generate Δz which is added with z_t gives z_{t+1}. z_{t+1} is used in two places first with decoder $D^+(z)$ to generate velocity field for next time step and to calculate Δz for next step.

The encoder is used only at t=0, after which all the generation is done by using T($\mathbf{z}$) and $D^+(z)$ in combination, since the velocity fields are generated from compressed feature maps they are called as generative models. The following flow chart 6.20 explains the flow of data/matrices in the architecture.

Algorithm 3 Simulation with the Latent Space Integration Network

$z_0 \leftarrow E(u_0)$

while simulating from t to t+1 **do**

$\Delta z \leftarrow T(z_t)$ // z_t from previous step, **p** is given

$z_{t+1} \leftarrow z_t + \Delta z$

$u_{t+1} \leftarrow D(z_{t+1})$ // velocity field reconstruction

end while

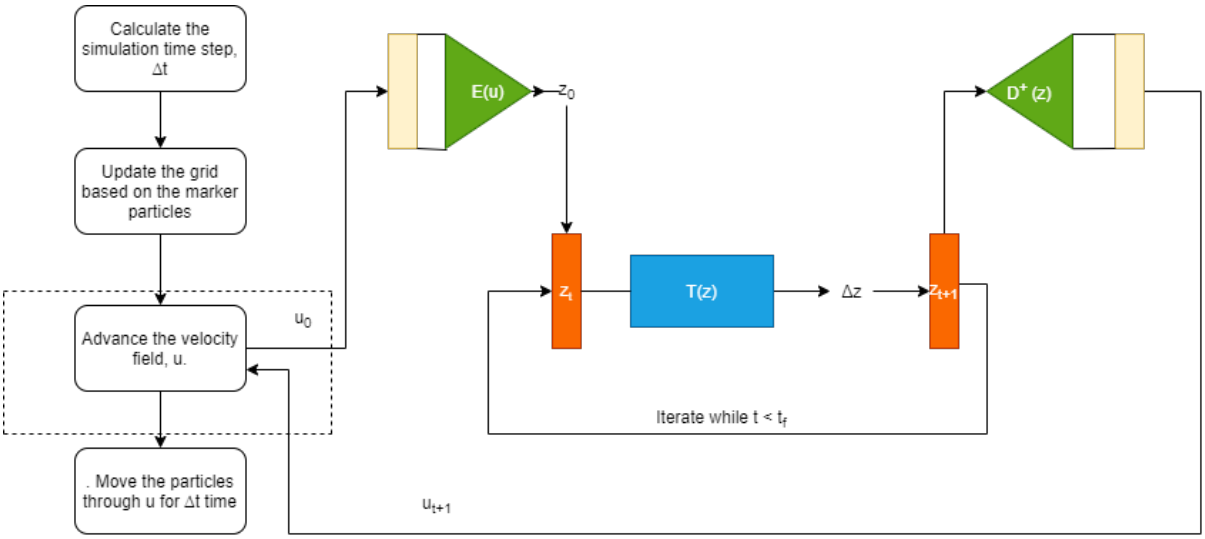

Figure 6.20: End to end architecture for Fluid simulations using generative networks.

The solvers are initialized in a manner mentioned in chapter 5, then instead of using the solver to advect velocity we use architect mentioned in the above section to generate velocity field for each time step and then return this velocity back into the solvers for updating the fluid position, and other grids.

Results and Discussion

7.1 Simulations

We have train and test the architecture on two different types of fluid simulation:

- **Dam Break:** This problems closely resembles a avalanche because Avalanche is type of dam-break problem, In a typical dam break simulation there is a water column at t=0 the water starts falling due to gravity, and its flow is analysed w.r.t time.

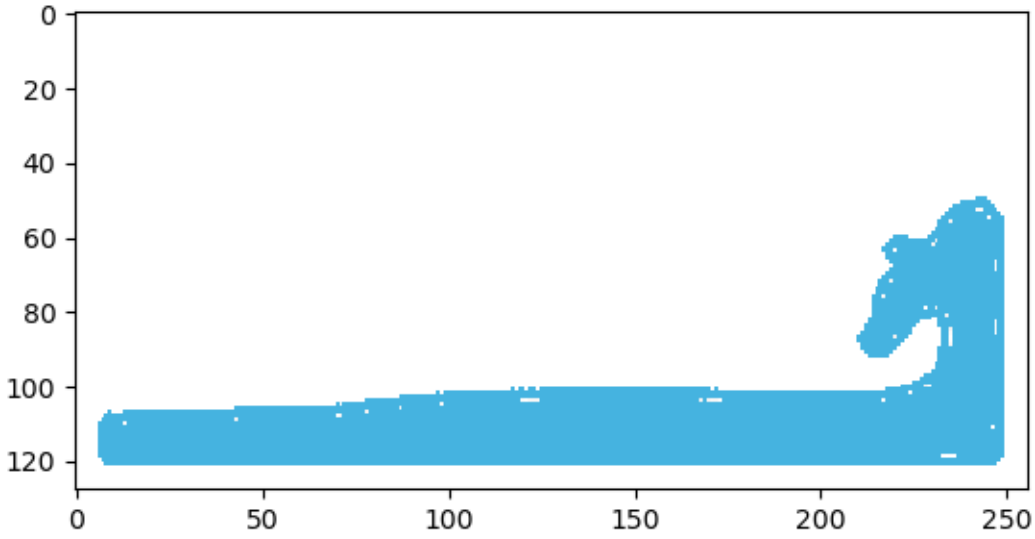

Figure 7.21: Dam Break simulation

- **Smoke Plume:** This type of simulation has a smoke source located at a particular x,y coordinate in the grid/domain which continuously adds some fluid(smoke) to the domain in consideration.

Figure 7.22: Smoke Plume simulation

All of the simulations were done in 2D hence the depth of all the generated velocity fields, density fields, etc. would be 1 or not to be considered.

7.2 Dam Break and Neural Network

The simulation of the dam break is done according to the Algorithm 1 Dam break Simulation in Mantaflow , The size of the water column was varied for 100 different values and for 200 frames which generated 20,000 different velocity field frames for the model to train onto. The velocities were normalised between [-1,1] using min and max values $x'' = 2\frac{x-\min x}{\max x-\min x} - 1$

The table shows the parameters of the deep learning model 7.4

Table 7.4: Model Parameters for Dam Break

Parameter	Quantity
q: Number of small blocks	5
N:number of convolution layers	4
f: Number of filters per convolution	128
Kernel size	3 x 3
Activation	leaky ReLU
Optimizer	Adam
Batch Size	8
Decaying Learning Rate	1e-4 to 2.5e-6
Epochs	30

7.2.1 Convergence

Each epoch was accompanied with 1000 steps, of the generator and the model loss started to converge at 6-7 epoch but the images generated out of the saved weights were problematic where some fluid chunks were flying away here and there, some fluid behaviors were unnatural but as the epochs proceeded the model learned the distribution of the simulations and simulations seemed to be more alike ground truth. The Training loss is shown in Fig 7.23.

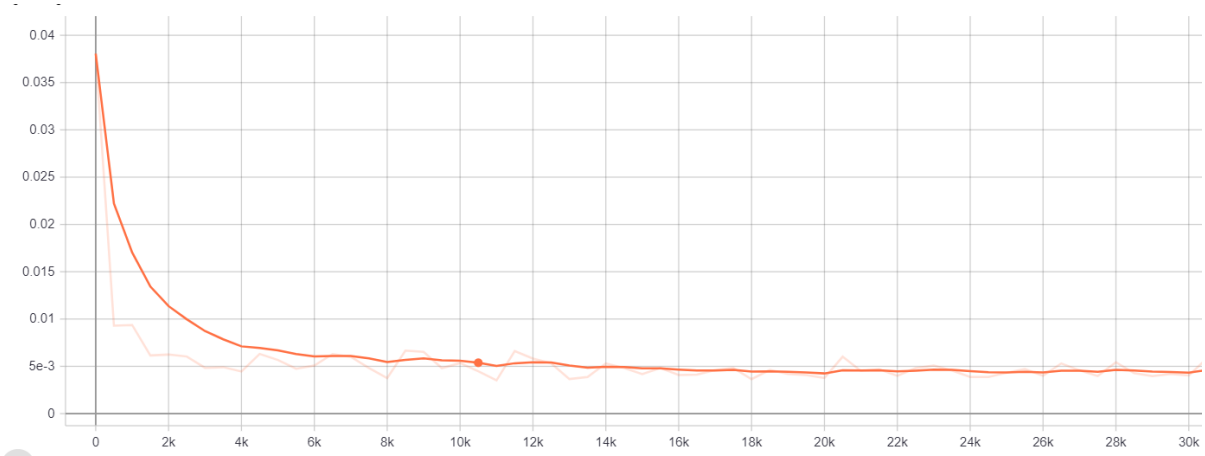

Figure 7.23: Training Loss of Dam Break Simulation

7.2.2 Results

Although we trained for 30 epochs, anything after the 15th epoch was overkill, hence we used weights from 15th epochs and did the simulations using Algorithm 2. Some snapshots of the dam-break simulations are shown in Fig 7.24

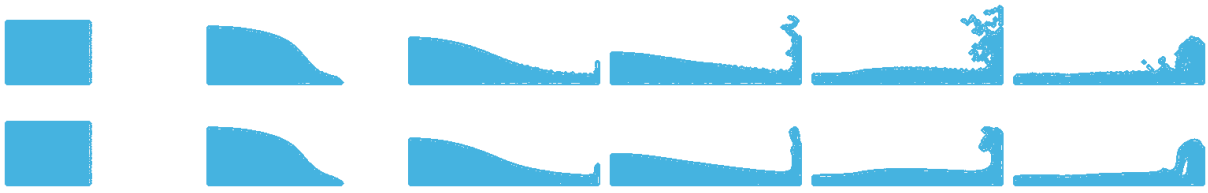

Figure 7.24: Snapshots of the simulation, the first row represents ground truth while the second row represents output from the neural networks

The architecture is able to closely mimic the solvers as seen in the above images, the snapshots were taken at t=0, t=10, t=20, t=30, t=50 and t=90. One can also observe some different behavior in terms of the fluid behavior for snapshot at t=50, in ground truth after crashing with wall some fluid segments fly away from the remaining mass, whereas no such behavior is exhibited by the neural network architecture, this behavior can be attributed to the effects of the velocity field gradient in the loss function, because in majority part of the

fluid simulations the surfaces are smooth.

7.3 Smoke Plume and Neural Network

A simple simulation of smoke plume was done in order to assess the initial capabilities of the model, this too was normalised between -1 and 1.

Table 7.5: Model Parameters for Smoke Plume

Parameter	Quantity
q: Number of small blocks	4
N:number of convolution layers	4 for initial , 2 and 3 for last
f: Number of filters per convolution	128
Kernel size	3 x 3
Activation	leaky ReLU
Optimizer	Adam
Batch Size	8
Decaying Learning Rate	1e-4 to 2.5e-6
Epochs	120

Each epoch was accompanied with 1000 steps, of the generator and the model loss started to converge at 6-7 epoch but the images generated out of the saved weights were problematic where some smoke moving downwards instead of upwards but as the epochs proceeded the model learned the distribution of the simulations and simulations seemed to be more alike ground truth. The Training loss is shown in Fig 7.25

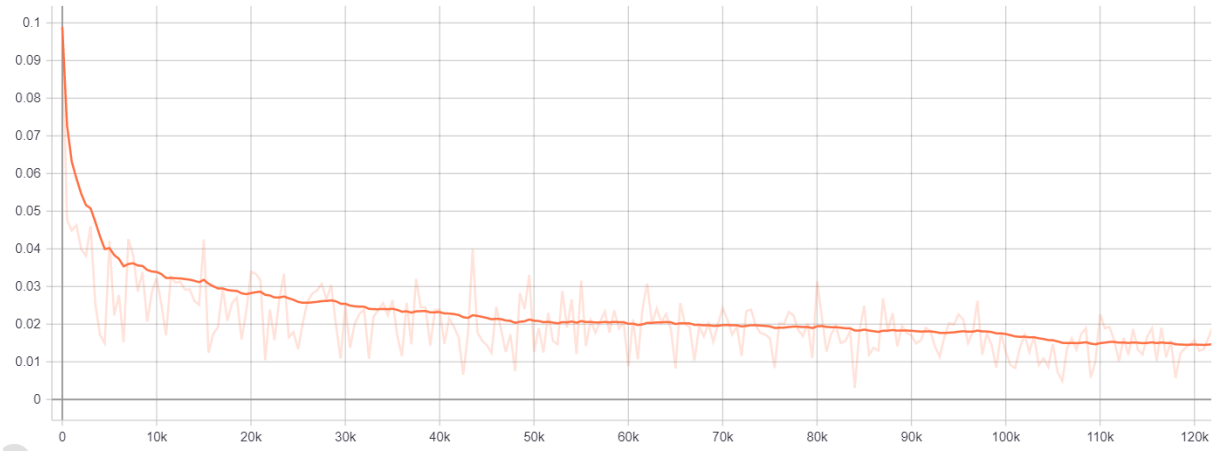

Figure 7.25: Training Loss of Smoke Plume Simulation

7.3.1 Results

To analyse the results we use vorticity plots of the velocity field (∇u) of the smoke simulations, here too we use results from 50th epoch because 120 epochs seemed to be overkill. The vorticity plots at different timesteps look something like this, blue for anticlockwise and red for clockwise is shown in Fig 7.26

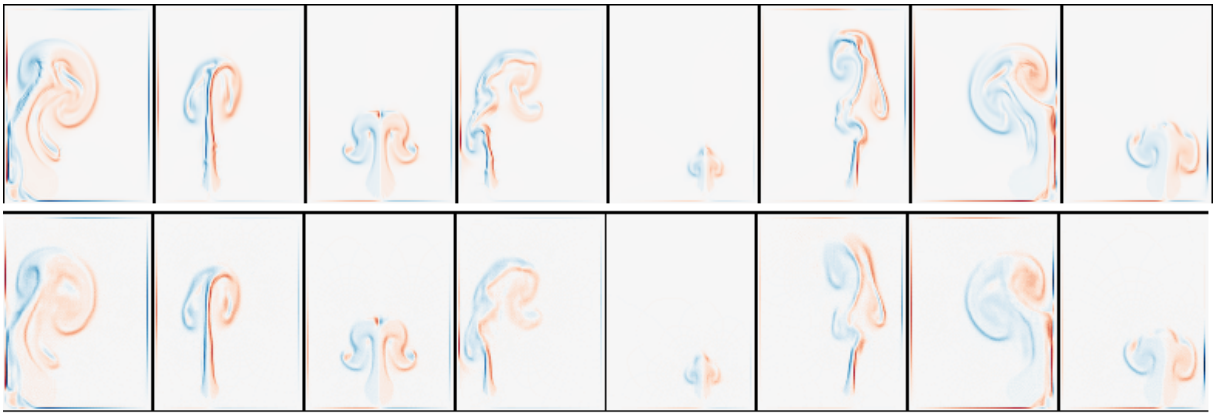

Figure 7.26: Vorticity plots for smoke simulation, top row is ground truth, bottom row is generation

The model is able to replicate the solvers as seen in the above random snapshots of the velocity field, here too one can observe that the plots are not as sharp as those of ground truth which can be attributed to the learning caused by velocity gradient in the loss.

Latent Space Physics: Learning the Temporal Evolution of Fluid Flow

8.1 Neural Network for Temporal Evolution

In the real world problems, the variables which are used to defined the physical systems become complex functions with high dimensionality. The field of computational methods can be used to solve such types of models.Using the deep learning approach we can learn the temporal evaluation of such complex functions like in the Navier Stokes equation describing the fluid flow.

To model the fluid flow as demonstrated in [13], the high dimensionality problem is reduced by using the convolutional neural networks with respect to both time and space. The first method discussed in [13] is to learn to map the original three dimensional problem into a much smaller latent spatial space and the reverse mapping of the smaller latent space to the original dimension. Then using the neural network, we train the neural network that maps the maps a collection of reduced representations into encoding of temporal evolution and this reduced temporal state is used to output sequence of spatial latent space representations. These sequence of spatial latent space representations are decoded to get the full spatial space data i.e in the original dimension for a point in time.
The major features of this paper [13] are :

- LSTM architecture to predict temporal evolutions of dense , physical 3D functions in learned latent space
- Efficient encoder and decoder architecture by which strong compression of data can be achieved to yield fast simulations.

8.2 Method

The main aim of the paper [13] is to predict the future states of a physical function **x**. Here the **x** is assumed to take the form of dense Eulerian function and can be used to represent spatial-temporal pressure function or the

velocity field. Thus, $x : R^3 \times R^+ \leftarrow R^d$, where when d=1 can represent scalar functions like pressure, and when d=3 can represent the velocity field functions.Thus , the **x** has a high dimensionality on discretizing, typically in order of millions of spatial degrees of freedom.

Given the set of parameters θ and the functional representation f_t, the goal is to predict the q future states $x(t+h)$ to $x(t+qh)$ as accurate as possible when given a current state and series of n previous states as mentioned in equation 8.18

$$f_t((xt - nh), \ldots, x(t-h), x(t)) \approx [x(t+h), \ldots, x(t+qh)] \tag{8.18}$$

8.2.1 Reduce the dimensionality

Due to high dimensionality of **x**, solving the equation 8.18 is very costly. To reduce the computation cost, we use the encoding and decoding functions f_e and f_d to compute the low dimensional encoding. The encoder f_e maps the data into m_s dimensional space $c \in R^{m_s}$ with $f_e(x(t)) = c^t$ and the decoding function do the reverse mapping with $f_d(c^t) = x(t)$, in combination both f_e and f_d represents the encoder and decoder models.

$$f'_t(f_e(xt - nh), \ldots, f_e(x(t))) \approx [c^{t+h}, \ldots, c^{t+qh}] \tag{8.19}$$

$$f_d([c^{t+h}, \ldots, c^{t+qh}]) \approx [x(t+h), \ldots, x(t+qh)] \tag{8.20}$$

$$f_d(f'_t(f_e(x(t-nh), \ldots, f_e(x(t)))) \approx f_t(x(t-nh), , x(t)) \tag{8.21}$$

where f'_t is predicted function

To reduce the dimensionality the autoencoder network is trained to reconstruct the quantity x as accurately as possible w.r.t L_2 norm,

$$\min_{\theta_d, \theta_e} |f_d(f_e(x(t))) - x(t)|_2 \tag{8.22}$$

where θ_d, θ_e are encoder and decoder parameters.

The series of convolutional layers activated with Leaky ReLU are used in encoder and decoder network. Both encoder and decoder consist of 6 convolutional layer to increase or decrease the number of features by a factor of 2, in total yield a reduction of by factor 256.

The autoencoder architecture is shown in Fig 8.27

Layer	Kernel	Stride	Activaion	Output	Features
Input				r/1	d
f_{e0}	4	2	Linear	r/2	32
f_{e1}	2	2	LeakyReLU	r/4	64
f_{e2}	2	2	LeakyReLU	r/8	128
f_{e3}	2	2	LeakyReLU	r/16	256
f_{e4}	2	2	LeakyReLU	r/32	512
f_{e5}	2	2	LeakyReLU	r/64	1024
c_5				r/64	1024
f_{d5}	2	2	LeakyReLU	r/32	512
f_{d4}	2	2	LeakyReLU	r/16	256
f_{d3}	2	2	LeakyReLU	r/8	128
f_{d2}	2	2	LeakyReLU	r/4	64
f_{d1}	2	2	LeakyReLU	r/2	32
f_{d0}	4	2	Linear	r/1	d

Table 8.6: Parameter of Autoencoder layers. Here $r \in N^3$ denotes the resolution of data and $d \in N$ its dimensionality

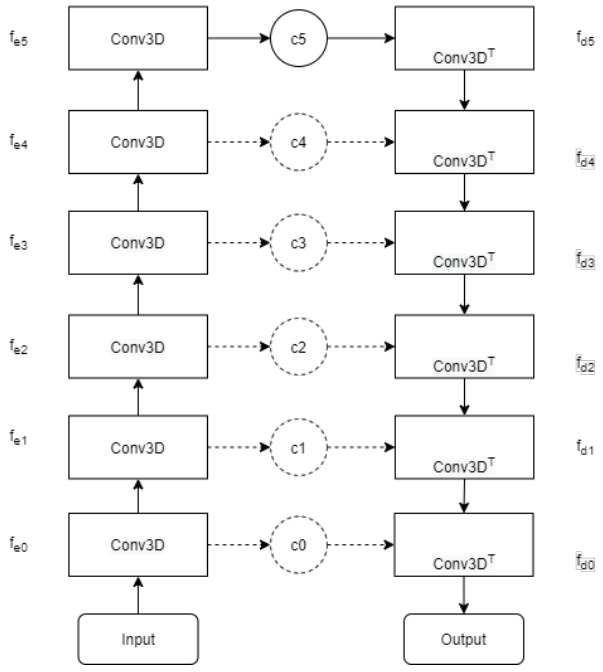

Figure 8.27: Liquid Column Simulation

8.2.2 Prediction of future States

The prediction network which models f_t' transformes the n+1 chronological encoded input states $X = (c^{t-nh}, \ldots, c^{t-h}, c^t)$ into a consecutive list of **q** predicted future states $Y = (c^{t+h}, \ldots, c^{t+qh})$ [13]. During the training the aim is the minimization of the mean absolute error between the q predict states and the ground truth using L_1 norm

$$\min_{\theta_t} |f_t'(c^{t-nh}, , c^(t-h), c^t) - [c^{t+h}, \ldots, c^{t+qh}]|_1 \tag{8.23}$$

θ_t denotes the parameters of the prediction network which is LSTM [. , .] denotes the concatenation of **c** vectors.

The prediction network uses convolutions to tranlslate the LSTM state into the latent state in addition to fully connected layers of LSTM[13]. The prediction network approximates the desired output function f_t' using the internal temporal context m_t denoted as **d**. The first part of the prediction network represents the encoder module which transforms the n+1 latent space into temporal context **d**.The time decoder module takes **d** as input and outputs series of future spatial latent space representations. Also the time decoder module due to internal state is trained to predict q future states on receiving the same **d** repeatedly. The tanh activation is used for all the LSTM along with hard sigmoid functions for efficient internal activations.

The architecture of LSTM is shown in Fig 8.28

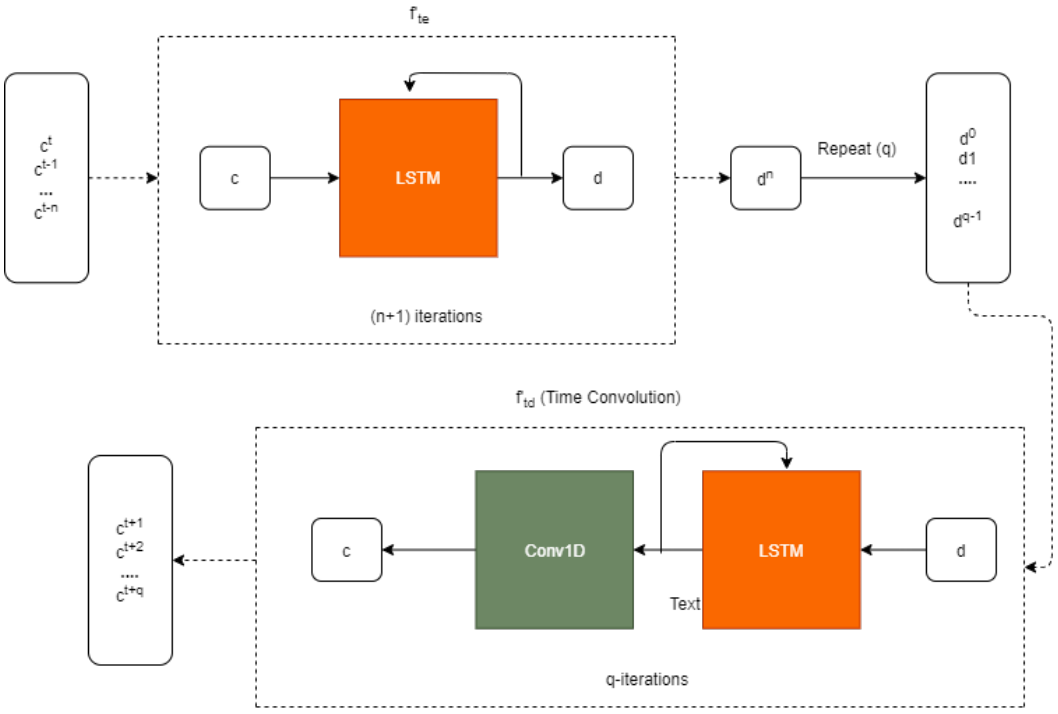

Figure 8.28: Architecture of LSTM prediction network

The encoder actually produces n+1 contexts of which first n are intermediate contexts which are only required for feedback loop corresponding to LSTM layer and are discared afterwards. The last context is kept

and passed to decoder part of network. This context is repeated q times in order for the decoder LSTM to predict desired states. The autoencoder can be applied to inputs of varying sizes but the prediction network is trained for fixed latent space inputs and internal context sizes. The latent space size m_s influences the size of prediction network layers. Thus, the prediction network has to be trained from scratch.

8.3 Fluid Flow Data

The fluid data set is generated using Navier Stokes solver. The steps involved are:

- Computing motion of fluid with the help of advection for velocity field **u**.
- Evaluating external forces
- Computing the harmonic pressure function
- Levelset ϕ representation for free flows is advected parallel to velocity

The solving of elliptic second order PDE typically involved in calculating pressure and the gradient of resulting pressure is used to make the flow divergence free. With the given data driven methods to predict the pressure fields, the hydrostatic pressure model can be used into 3D liquid simulations by decomposing the regualar pressure (p_t) field into hydrostatic (p_s) and dynamic (p_d) components,

$$p_t = p_s + p_d \tag{8.24}$$

With this decompositon, the autoencoder would learn small scale fluctuations in p_d.

8.4 Fluid Simulations and Results

The liquid scene setup is generated by using mantaflow framework. The scene setup shows the low basin represented by the large volume of liquid at the bottom of domain. A tall narrow pillar of liquid drops into the basin.

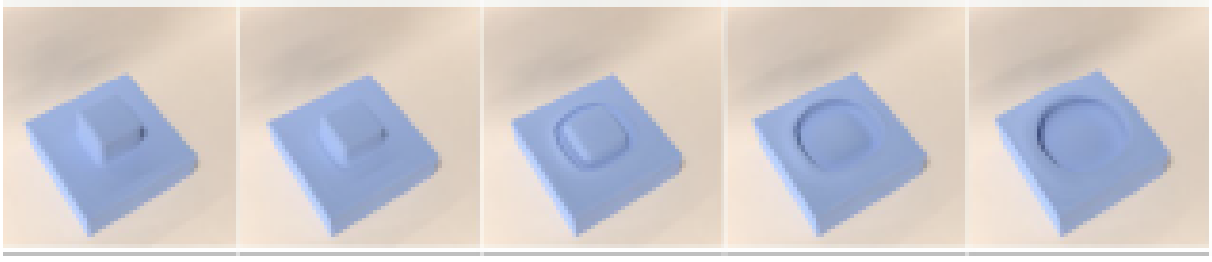

Figure 8.29: Liquid Column Simulation

Table 8.7: Model Parameters for Autoencoder

Parameter	Quantity
Convolutional Layers	6
Latent Space size	1024
Optimizer	Adam
Learning Rate	0.001
Decaying Rate	0.005
Epochs	37

8.4.1 Results

The model to predict the total pressure was trained for 37 epochs with Given loss functions in 8.22 the training and validation loss were as follows:

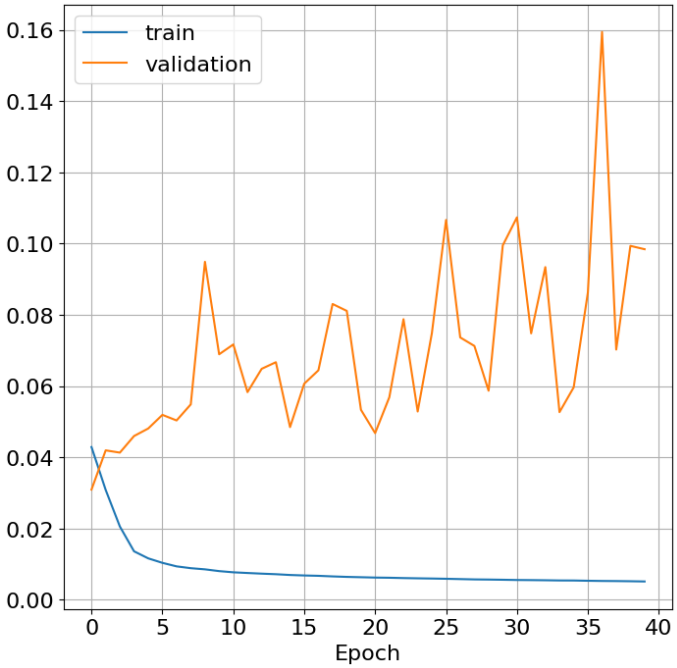

Figure 8.30: Training and Validation loss for AE

As one can observe from figure 8.30 , the training loss seems to converge after first epoch the validation loss starts to increase which shows a high bias towards training data. hence any results that were generated were futile

Future Works

- **Accelerating complex fluid simulations:** Now that we have established a base framework for inculcating neural networks in the Fluid solvers, we can increase the complexity of the fluid simulation by adding different slopes into the equations or complex terrains with obstacles the domain.
- **Extending the framework to Pressure field:** As demonstrated in Chapter 8: similar architecture for accelerating the solvers on pressure fields can also be used compressed features for pressure field can also be used.

Bibliography

[1] D. Dutykh, C. Acary-Robert, and D. Bresch, "Mathematical modeling of powder-snow avalanche flows," *Studies in Applied Mathematics*, vol. 127, p. 38–66, Feb 2011.

[2] Y. Xie, E. Franz, M. Chu, and N. Thuerey, "tempogan: A temporally coherent, volumetric GAN for super-resolution fluid flow," *CoRR*, vol. abs/1801.09710, 2018.

[3] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating eulerian fluid simulation with convolutional networks," *CoRR*, vol. abs/1607.03597, 2016.

[4] B. Kim, V. C. Azevedo, N. Thuerey, T. Kim, M. H. Gross, and B. Solenthaler, "Deep fluids: A generative network for parameterized fluid simulations," *CoRR*, vol. abs/1806.02071, 2018.

[5] P. A. Cundall and O. D. L. Strack, "A discrete numerical model for granular assemblies," *Géotechnique*, vol. 29, no. 1, pp. 47–65, 1979.

[6] K. Yamane, M. Nakagawa, S. A. Altobelli, T. Tanaka, and Y. Tsuji, "Steady particulate flows in a horizontal rotating cylinder," *Physics of Fluids*, vol. 10, no. 6, pp. 1419–1427, 1998.

[7] A. C. Fischer-Cripps, "The hertzian contact surface."

[8] D. Cline, D. A. Cardon, P. K. Egbert, and B. Young, "Fluid flow for the rest of us : Tutorial of the marker and cell method in computer graphics," 2005.

[9] F. H. Harlow and J. E. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *The physics of fluids*, vol. 8, no. 12, pp. 2182–2189, 1965.

[10] N. Thuerey and T. Pfaff, "MantaFlow," 2018. *http://mantaflow.com*.

[11] K. Abdolmaleki, K. Thiagarajan, and M. Morris-Thomas, "Simulation of the dam break problem and impact flows using a navier-stokes solver," *15th Australian Fluid Mechanics Conference*, vol. 13, 01 2004.

[12] J. Brackbill, D. Kothe, and H. Ruppel, "Flip: A low-dissipation, particle-in-cell method for fluid flow," *Computer Physics Communications*, vol. 48, pp. 25–38, 01 1988.

[13] S. Wiewel, M. Becher, and N. Thuerey, “Latent-space physics: Towards learning the temporal evolution of fluid flow,” *CoRR*, vol. abs/1802.10123, 2018.