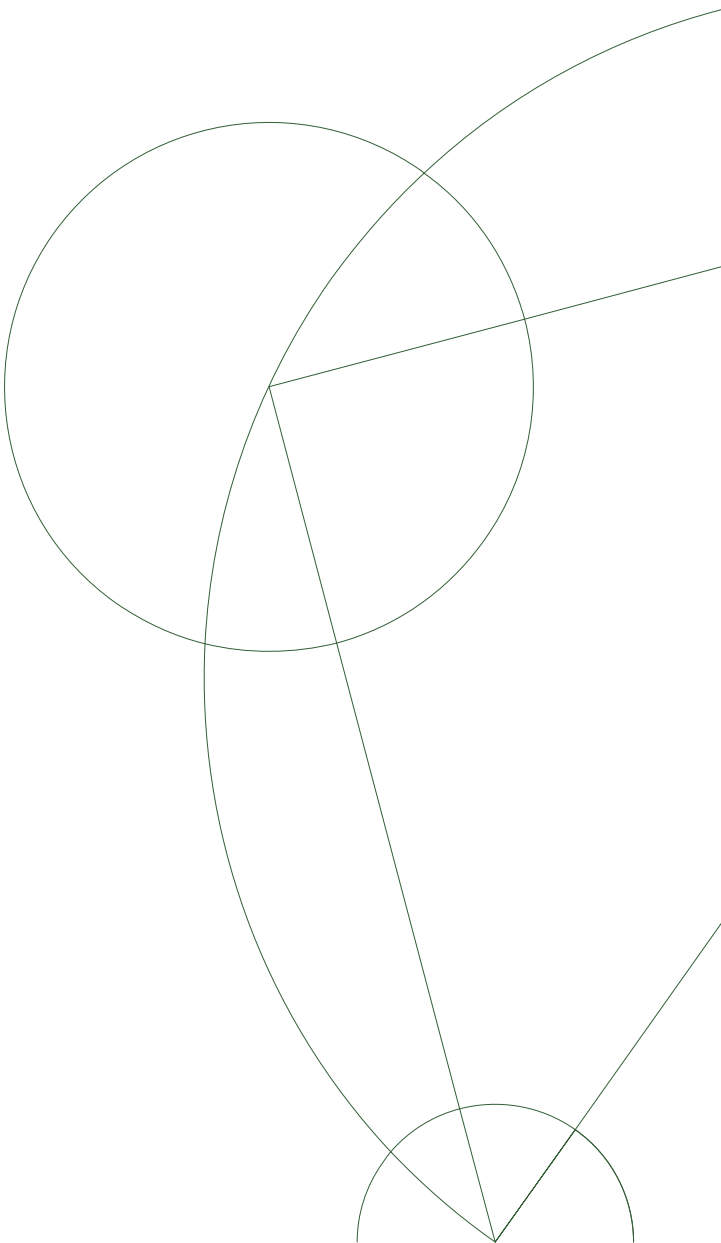
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Jian Wu – xcb479@alumni.ku.dk

Deep Contact

Accelerating Rigid Simulation with Convolutional Networks



Supervisor: Kenny Erleben

August 6th 2018

Abstract

The subject of this thesis is to apply deep learning method in rigid dynamic simulation, specifically for predicting good contact forces. The central concept of this thesis is to convert the simulation state information to accessible data for deep learning model. The basic idea is to use smoothed-particle hydrodynamics(SPH) based method to transform all essential data to a grid image, then predicting a grid image for contact forces. The values of contact forces generated from contact grid image by interpolation would be used as warm starting for iterative contact solver.

The results show that smoothed-particle hydrodynamics(SPH) based method is a good way to convert the simulation state to a multiple- channels image without losing lots of essential information. Finally, the trained CNN model can actually provide great warm starting values for the contact solver in each step. But because it has to take the longer time on computing interpolation values, the CNN based method is not still efficient. The implementation of CNN experiments also shows the potential of applying deep learning in contact solution.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introdustion

1.1 Motivation

Deep learning has been widely used in many academic and industrial fields. Due to no need for too much data-processing and generally high performance, deep learning has become increasingly popular to replace some traditional methods. Although researchers have developed effective deep learning methods to push computer vision to a new high level, deep learning for computer simulation is still not far away and clear. Rigid and liquid simulation problems are always the most interesting and important issues in the computer simulation. Researchers are exploring the possibility of using deep learning methods to improve current simulation solutions. A paper recently published announces that deep learning has been successfully applied to the upgrade of liquid simulation[1].

However, for rigid body problems, it is not quite clear how to approach the technicalities in applying deep learning. Some work has been done in terms of inverse simulations or pilings to control rigid bodies to perform a given artistic 'target'. These techniques are more in the spirit of inverse problems that maps initial conditions to a well-defined outcome(number of bounces or which face up on a cube) or level of detail idea replacing interiors of piles with stacks of

cylinders of decreasing radius to make an overall apparent pile have a given angle of repose.

In this project, I would try to find a method which can apply deep learning on rigid simulation pipeline.

1.2 Overview

The centerpiece of this project is to covert a rigid body simulation into imaging data and apply to learn on these images. Basically, researchers mainly use iterative solver to compute contact forces for each time step. The idea shown in the thesis is not to completely replace contact solving by deep learning but rather take a hybrid approach of using the deep learning results as initial starting iterates for the contact forces solver. In other words, our hope is to use deep learning to provide usable warm starting that is close to the final solution. Totally, this project is consist of two parts. The first part is to find a method to generate accessible data for deep learning and test it whether it is a good strategy. Once experiments prove that it works well for the rigid dynamic simulation. The second part is to train one learning model based on the training dataset and apply the model to check its performance in the simulation pipeline. Ideally, the learning model can accelerate the iterative solution process.

The paper is mainly composed of 6 chapters

- **Chapter 1**, is an introduction to the entire paper, including motivation, learning objectives, workflow, and outline.
- **Chapter 2**, mainly gives a brief description for contact models, inlcuding how to applied classic *Newton-Euler* equation with constraints.
- **Chapter 3**, is another important part to analyze the main interpolation method used for the project, Smoothed Particle Hydrodynamics(SPH). At the end of the chapter, some experiments are taken to analyze whether SPH is good to this case.

- **Chapter 4**, gives some description of one of the current hot techniques, deep learning, including a brief history of deep learning, some basic mathematics concepts, and optimization methods for the training process.

- **Chapter 5**, does describe implementation details specifically, from data generation to model design and training. Also, some analyses are made based on the experiment, including what values of Smoothed Particle Hydrodynamics(SPH) settings(kernel, grid size, and smoothing length) should be given, how to set the model training parameters, and comparison between learning model and built-in algorithm in simulation software.

- **Chapter 6**, is the end of this paper. According to the experimental implementation, some conclusions are drawn. Finally, a plan is developed for the future work, which can be an improvement over the current project.

- **Appendix A** Sample code for smooth kernel used in smoothed particle hydrodynamics

- **Appendix B** Sample code for Convolutional Neural Network architecture designed in this thesis. And some parts of deep learning setting(loss function, learning rate scheduler)

1.3 Goals

Taking a master's degree is a specialization process, a fine-tuning of skills. Doing a master thesis is more like a process of learning. During the master thesis project, you should only focus on one topic and put all your effort into it. Before doing this project, I spent more time on computer vision and deep leaning, without any experience in the computer simulation. Reading many papers during this project helps me get some basic knowledge in contact model simulation and

go deeper into deep learning. More importantly, I learned how to start and do one research when you are facing new topics

1.3.1 Learning Goals

In this work, I would design some experiments on assessing the performance of the training data generation and deep learning model. Six months ago, I set up a list of learning goals in the projecr description. Now, I enclose it here.

1. Describe the contact force problem among rigid objects by building *Newton-Euler* equations.
2. Analyze possible kernels which can work for simulator and compare the performances of different kernels on mapping the state of the simulator onto a grid.
3. Analyze and compare the performance of different grid-sizes on the chosen kernel.
4. Design one convolution neural network to transfer momentum images into contact force images.
5. Design one experiment to determine the accuracy of several force solutions.
6. Describe the questions and issues during the learning process and reflect on how to make learning model work better.
7. Design one experiment about training both normal forces and friction forces as one map.
8. Design one experiment about training normal forces and friction forces as two maps.
9. Compare the two results from two experiments.

1.4 Work Contribution

Since this is a student master project, Lukas, Lucian and I are engaged in the same topic. We finished the initial work with cooperation. All our work would be base on a python game engine, *pybox2d*. The initial work includes,

- *pybox2d* updating, which is to upgrade code of *pybox2d* so that it can output the data that we need.
- Grid-particle transformation, includes smoothed particle hydrodynamics(SPH) implementation, interpolation query building.

After that, we did individual experiments to set parameters for generating data and designed our learning model separately. All public and personal code can be reviewed on Github Repository[1]. The specific code can be also reviewed in Appendix. The final trained model can be obtained and tested on my personal Dropbox[2].

[1] `https://github.com/JaggerWu/Deep-Contact`

[2] `https://www.dropbox.com/s/jrwzqib6ghrq59i/model.h5?dl=0`

Chapter 2

Contact Models

In an attempt to apply deep learning in a contact dynamic system, the first step is to understand how the contact dynamic system is modeled and how system model is solved by numerical method. Therefore, this chapter focuses on rigid body simulations to help readers understand how computers simulate rigid dynamics based on traditional *Newton-Euler* equations, including how *Newton-Euler* describes rigid dynamics, how to set constraint equations and numerical solution for dynamic system equation.

2.1 Overview

In rigid body simulation, contact force is used to prevent the rigid bodies from penetrating each other. The accuracy of the calculated contact force has great influence on the fidelity of simulation. In order to achieve physical rationality, frictional forces are necessary. The formulae of friction contact force problems include the modeling of normal force constraint and friction force constraint. Because that frictional contact modeling is more complicated, I will give a simple and general solution for normal contact forces. The frictional contact model will be mentioned briefly, rather than giving details. Besides, the topic of the paper is about applying deep learning to contact simulation, I just want to do the simulation as simple as possible.

So rigid bodies' shape would not be considered here. All the bodies would be a circle, which is similar to particles.

Before exporting the contact model, it is important to note that the simulation system is constrained by velocity-based location updates [2]. Since collision detection is another big academic topic [3] and this is a thesis on contact problems, the following models are given under the assumption that contact determination has already been performed.

2.2 Rigid dynamics Simulation

2.2.1 Classical mechanics

Simulation of the motion of a system of rigid bodies is based on a famous system of differential equations, the *Newton–Euler equations*, which can be derived from Newton's laws and other basic concepts from classical mechanics:

1. Newton's first law: The velocity of a body remains unchanged unless acted upon by a force.

2. Newton's second law: The time rate of change of momentum of r a body is equal to the applied force.

3. Newton's third law: For every force, there is an equal and opposite force.

Before presenting the *Newton–Euler* equations, we need to introduce a number of concepts from classical mechanics. I will start with one simple simulation with only position vector $\boldsymbol{q}(t)$ and velocity vector $\boldsymbol{v}(t)$. Then, we will introduce some concepts by adding rotation to pure simulation, rotational velocity $\boldsymbol{\omega}(t)$, and moment $\boldsymbol{\tau}$ (also known as a torque).

2.2.2 Simulation Basics

Firstly, we can start with a simple simulation with only position and velocity. Simulating the motion of a rigid body is almost the same as simulating the motion of a particle, so I will start with particle simulation. For particle simulation, we let function $\boldsymbol{q}(t)$ describe the particle's location in world space at time t. Then we use the change of $\boldsymbol{q}(t)$ to denote the velocity of the particle at time t.

$$\boldsymbol{v}(t) = \dot{\boldsymbol{q}}(t) \tag{2.1}$$

So, the state of a particle at a time t is the particle's position and velocity. We generalize this concept by defining a state vector $\mathbf{Y}(t)$ for a system: for a single particle,

$$\mathbf{Y}(t) = \begin{bmatrix} \boldsymbol{q}_1(t) \\ \boldsymbol{v}_1(t) \end{bmatrix} \tag{2.2}$$

For a system with n particles, we enlarge $\mathbf{Y}(t)$ to be

$$\mathbf{Y}(t) = \begin{bmatrix} \boldsymbol{q}_1(t) \\ \boldsymbol{v}_1(t) \\ ... \\ \boldsymbol{q}_n(t) \\ \boldsymbol{v}_n(t) \end{bmatrix} \tag{2.3}$$

However, to simulate the motion of particles actually, we need to know one more thing – the forces. $\boldsymbol{f}(t)$ is defined as the force acting on the particle. If the mass of the particle is m, then the changes of $\mathbf{Y}(t)$ will be given by

$$\dot{\mathbf{Y}}(t) = \frac{\mathrm{d}}{\mathrm{d}(t)}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}(t)} \begin{bmatrix} \boldsymbol{q}(t) \\ \boldsymbol{v}(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}(t) \\ \boldsymbol{f}(t)/m \end{bmatrix} \tag{2.4}$$

2.2.3 Rigid Body Concepts

Unlike a real particle, a rigid body occupies a volume of space and has a particular shape(including circle rigid objects). Rigid bodies

are more complicated, besides translating them, we can rotate them as well. To locate a rigid body, we use $\boldsymbol{q}(t)$ to denote their translation and a rotation matrix $\boldsymbol{R}(t)$ to describe their rotation.

2.2.4 Rigid Body Equations of Motions

Whereas linear momentum $\boldsymbol{P}(t)$ is related to linear velocity with a scalar (the mass), angular momentum is related to the angular velocity with a matrix $\boldsymbol{I}$, called the angular inertia matrix. The reason for this is that objects generally have different angular inertias around different axes of rotation. Angular momentum is defined as $\boldsymbol{L}$. The linear momentum and angular momentum are defined in Equation 2.5.

$$m\dot{\boldsymbol{v}}(t) = \boldsymbol{f}(t) \tag{2.5a}$$

$$\boldsymbol{P}(t) = m\boldsymbol{v}(t) \tag{2.5b}$$

$$\boldsymbol{L}(t) = \boldsymbol{I}(t)\boldsymbol{\omega}(t) \tag{2.5c}$$

The total torque $\boldsymbol{\tau}$ applied to the body is equal to the rate of change of the angular momentum, as defined in 2.11:

$$\boldsymbol{\tau} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{L} = \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{I}\boldsymbol{\omega}) \tag{2.6}$$

Then we can covert all concepts we need to define stare $\mathbf{Y}$ for a rigid body.

$$\mathbf{Y}(t) = \begin{bmatrix} \boldsymbol{q}(t) \\ \boldsymbol{R}(t) \\ \boldsymbol{P}(t) \\ \boldsymbol{L}(t) \end{bmatrix} \tag{2.7}$$

Like what is expressed in $\mathbf{Y}(t)$, the state of a rigid body mainly consists by its position and orientation (describing spatial information), and its linear and angular momentum(describe velocity information). Since mass m and bodyspace inertia tensor $\boldsymbol{I}_{body}$ are

constants, we can the auxiliary quantities $\boldsymbol{I}(t)$, $\boldsymbol{\omega}(t)$ at any given time.

$$\boldsymbol{v}(t) = \frac{\boldsymbol{P}(t)}{m} \tag{2.8a}$$

$$\boldsymbol{I}(t) = \boldsymbol{R}(t)\boldsymbol{I}_{body}\boldsymbol{R}(t)^T \tag{2.8b}$$

$$\boldsymbol{\omega}(t) = \boldsymbol{I}(t)^{-1}\boldsymbol{L}(t) \tag{2.8c}$$

Then, the derivative $\dot{\mathbf{Y}}(t)$ is

$$\dot{\mathbf{Y}}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} \boldsymbol{q}(t) \\ \boldsymbol{R}(t) \\ m\boldsymbol{v}(t) \\ \boldsymbol{L}(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}(t) \\ \boldsymbol{\omega}(t) \times \boldsymbol{R}(t) \\ \boldsymbol{f}(t) \\ \boldsymbol{\tau}(t) \end{bmatrix} \tag{2.9}$$

Then, we can evaluate Equation 2.10 as follows:

$$\begin{aligned} \boldsymbol{\tau} &= \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{I}\boldsymbol{\omega}) \\ &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + \dot{\boldsymbol{I}}\boldsymbol{\omega} \\ &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T)\boldsymbol{\omega} \\ &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + (\dot{\boldsymbol{R}}\boldsymbol{I}_{body}\boldsymbol{R}^T + \boldsymbol{R}\boldsymbol{I}_{body}\dot{\boldsymbol{R}}^T)\boldsymbol{\omega} \\ &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + ([\boldsymbol{\omega}]\boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T + \boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T\hat{\omega})\boldsymbol{\omega} \\ &= \boldsymbol{I}\dot{\boldsymbol{\omega}} + [\boldsymbol{\omega}]\boldsymbol{I}\boldsymbol{\omega} - \boldsymbol{I}[\boldsymbol{\omega}]\boldsymbol{\omega} \end{aligned} \tag{2.10}$$

Since $\boldsymbol{\omega} \times \boldsymbol{\omega}$ is zero, the final term can be cancels out. This relationship left is known as :

$$\boldsymbol{\tau} = \boldsymbol{I}\dot{\boldsymbol{\omega}} + [\boldsymbol{\omega}]\boldsymbol{I}\boldsymbol{\omega} \tag{2.11}$$

2.2.5 Twist/Wrench

Twist is introduced to describe linear and angular velocity, and **Wrench**, is to describe forces and explain how these objects transform from one coordinate frame to another one.

Twist

A twist is a vector that expresses rigid motion or velocity. In Section 2.2.4, we saw how to parameterize the velocity of a rigid body as a linear velocity vector and an angular velocity vector. The coordinates of a twist are given as a 4-vector in 2-D simulation, which we can check in 2.12

$$\mathbf{v} = \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} \tag{2.12}$$

. The defination can be found in 2.12, containing a linear velocity vector $\boldsymbol{v}$ and an angular velocity $\boldsymbol{\omega}$. According to

Wrench

A wrench is a vector that expresses force and torque acting on a body. A wrench can be defined by

$$\mathbf{f} = \begin{bmatrix} \boldsymbol{\tau} \\ \boldsymbol{f} \end{bmatrix} \tag{2.13}$$

A wrench contains an angular component $\boldsymbol{\tau}$ and a linear component f, which are applied at the origin of the coordinate frame they are specified in.

2.2.6 Newton-Euler Equation

Newton-Euler equations for a rigid body can now be written in terms of the body's acceleration twist $\mathbf{v}$ mentioned in 2.12 and the wrench $\mathbf{f}$ mentioned in 2.13 acting on the body. We can simply write the Newton and Euler equations,

$$\begin{bmatrix} \boldsymbol{\tau} - \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} \\ \boldsymbol{f} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & m\mathbf{1}_{d\times d} \end{bmatrix} \dot{\mathbf{v}} \tag{2.14}$$

d stands for the number pf dimensions, like $d = 2$ in 2-D and 3 in 3-D.

Then, we can rewrite *Newton-Euler* equation as,

$$\boldsymbol{M}\dot{\mathbf{v}} = \mathbf{F} \tag{2.15}$$

where $\boldsymbol{M}$ and $\mathbf{h}$ are defined in Equation 2.16 and 2.17.

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & m\mathbf{1}_{d\times d} \end{bmatrix} \tag{2.16}$$

$$\mathbf{F} = \begin{bmatrix} \boldsymbol{\tau} - \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} \\ \boldsymbol{f} \end{bmatrix} \tag{2.17}$$

2.3 Velocity-based conatct model

After getting some basic physical and mathematical knowledge about a dynamic system. Then, the next step is to describe the equation of motion for the system with contact forces.

When the collision happens, we can use contact forces to model the interactions between two colliding objects. The contact forces consist of both normal forces and frictional forces, which are subject to a set of constraints. The constraints are defined in Equation 2.18.

$$\boldsymbol{c}(\boldsymbol{q}) = [c_1(\boldsymbol{q}), c_2(\boldsymbol{q}), ..., c_n(\boldsymbol{q})] \tag{2.18}$$

the Jacobian, $\boldsymbol{J}_c$ of Equation 2.18 is,

$$\boldsymbol{J}_c = \begin{bmatrix} \frac{\partial c_1}{\partial q_1} & \frac{\partial c_2}{\partial q_1} & \cdots & \frac{\partial c_n}{\partial q_1} \\ \frac{\partial c_1}{\partial q_2} & \frac{\partial c_2}{\partial q_2} & \cdots & \frac{\partial c_n}{\partial q_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial c_1}{\partial q_n} & \frac{\partial c_2}{\partial q_n} & \cdots & \frac{\partial c_n}{\partial q_n} \end{bmatrix} \tag{2.19}$$

The constraints are added to Equation 2.15 by applying the Lagrange multiplier method such that,

$$\boldsymbol{M}\dot{\mathbf{v}} = \boldsymbol{J}_c^T\boldsymbol{\lambda} + \mathbf{F}_{ext} \tag{2.20}$$

where $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, ..., \lambda_n]^T$ are the Lagrange multipliers.

2.3.1 Linear Complementarity Problem

The classical method to solve contacts forces is to derive the contact problems as a linear complementarity problem formulation. I will give the definition of Linear Complementarity Problem below,

(LCP): Given an unknown vector $\mathbf{x} \in \mathbb{R}^m$, a known fixed matrix $\boldsymbol{A} \in \mathbb{R}^{m \times m}$, and a known fixed vector $\boldsymbol{b} \in \mathbb{R}^m$, determine $\mathbf{x}$ such that,

$$f(\mathbf{x}) = \mathbf{x}^T(\boldsymbol{A} \cdot \mathbf{x} + \boldsymbol{b}) = 0 \tag{2.21}$$

subject to the constraints,

$$\boldsymbol{A} \cdot \mathbf{x} + \boldsymbol{b} \geq \mathbf{0} \tag{2.22a}$$

$$\mathbf{x} \geq \mathbf{0} \tag{2.22b}$$

For LCPs, we adopt the shorthand notation, LCP($\boldsymbol{A}, \boldsymbol{b}$).

2.3.2 Modeling contact

The contact force problem can be stated as a linear complementarity problem (LCP)[4]. I will derive this formulation. The focus of this chapter is on the contact force model, so the time stepping scheme and matrix layouts are based on the velocity-based formulation. If I just consider normal contact forces, the *Newton–Euler* equations based on Equation 2.20 can be rewritten as,

$$\boldsymbol{M}\dot{\mathbf{v}} = \boldsymbol{J}_n^T \boldsymbol{\lambda}_n + \mathbf{F}_{ext} \tag{2.23}$$

then we can get,

$$\dot{\mathbf{v}} = \boldsymbol{M}^{-1}\mathbf{F}_{ext} + \boldsymbol{M}^{-1}\boldsymbol{J}_n^T \boldsymbol{\lambda}_n \tag{2.24}$$

The laws of physics must be incorporated into what we call "instantaneous time", which describes the continuous motion of the rigid bodies. After that, we discretize the model over time to obtain a 'discrete-time' model, which is a series of time step sub-problems. Sub-problems are formulated and numerically solved at each time step to simulate the system.

To discretize the system 2.24, the acceleration can be approximated by [5] as:

$$\dot{\mathbf{v}} \approx \frac{(\mathbf{v}_{t+1} - \mathbf{v}_t)}{\Delta t} \tag{2.25}$$

$\mathbf{v}_t$ and $\mathbf{v}_{t+v}$ are the velocities at the beginning of the current time step, and the next time step, Δt is the time step. Then we can get,

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \boldsymbol{M}^{-1}\boldsymbol{J}_c^T \Delta t \boldsymbol{\lambda}_c + \Delta t \boldsymbol{M}^{-1}\mathbf{F}_{ext} \tag{2.26}$$

Then we can define,

$$\begin{aligned} \mathbf{w} &= \boldsymbol{J}_n \mathbf{v}^{t+1} \\ &= \underbrace{\boldsymbol{J}_n \boldsymbol{M}^{-1} \boldsymbol{J}_n^T \Delta t}_{\boldsymbol{A}} \boldsymbol{\lambda}_n + \underbrace{\boldsymbol{J}_n (\Delta t \boldsymbol{M}^{-1} \mathbf{F}_{ext} + \mathbf{v}^t)}_{\boldsymbol{b}} \\ &= \boldsymbol{A}\boldsymbol{\lambda}_n + \boldsymbol{b} \end{aligned} \tag{2.27}$$

Also, for a general $\mathbf{w}$

$$\mathbf{w} = \boldsymbol{J}\mathbf{v} = \frac{\mathrm{d}\boldsymbol{c}}{\mathrm{d}\boldsymbol{q}} \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \dot{\boldsymbol{c}} \tag{2.28}$$

Based on physics law, for each contact $\lambda = 0$ or $\dot{c} = 0$, then,

$$\boldsymbol{\lambda}^T \dot{\boldsymbol{c}} = 0$$

So Equation **??** can be considered as one LCP. So finally, contact model is to find solution for,

$$\boldsymbol{\lambda} = LCP(\boldsymbol{A}, \boldsymbol{b})$$

Modeling Friction

In order to make the virtual world more real, the friction is essential. However, when friction is imported to the model, the system will be more comlecated due to Coulomb's friction law. We can rewrite Equation 2.24,

$$\dot{\mathbf{v}} = \boldsymbol{M}^{-1}\mathbf{F}_{ext} + \boldsymbol{M}^{-1}\boldsymbol{J}_n^T\boldsymbol{\lambda}_n + \boldsymbol{M}^{-1}\boldsymbol{J}_t^T\boldsymbol{\lambda}_t \tag{2.29}$$

And the another constraints, $-\mu\lambda_n \leq \lambda_t \leq \mu\lambda_n$ for each contact, will be added to the system. Since the focus of paper is on find a solution by deep learning, I did not explore too much on frictinal contact model. I recommend Sarah and Kenny's paper [4], [6]to know more details.

2.4 The Numerical Solution Method

After obtaining discrete-time models, numerical methods must be applied to compute solutions. Normally, an iterative solution will be used to solve LCP. Projected Gauss-Seidel Solver is one of the most classic methods. One introduction will be given below,

2.4.1 Projected Gauss-Seidel(PGS) solver for contact forces

One of the most classic methods is projected gauss-seidel(PGS). Most open-source software for interactive real-time rigid body simulation uses the Projected Gauss-Seidel (PGS) method for computing contact forces, like *Box2D*[1] for 2-*D* and *Bullet*[2] for 3-*D*. PGS is computationally very efficient with an iteration cost of $O(n)$, using a careful memory layout of sparse matrices allows for a memory footprint of $O(n)$. In addition to being computationally and memory-wise efficient, PGS is very robust and can deal gracefully with ill-conditioned problems (due to many redundant constraints) or ill-posed problems (due to badly defined constraints). For these reasons, PGS is well suited for interactive applications like computer games. I introduced PGS in Algorithm 1. Generally, $\boldsymbol{\lambda}_{init} = \mathbf{0}$, and some experiment will be done to explore its convergence rate in section 3.4.2. Finally, PGS algorithm will return you the final solution $\boldsymbol{\lambda}$ and a list of convergence rate $\boldsymbol{\theta}$. Normally, the convergence rate can indicate the performance of iterative solver. In order to represent the overall performance of model or solver, I use average convergence rate in each iteration. You can see the analysis in Section 3.4.2 to get more details.

As a conclusion, the iterative solver will be used to solve the LCP problem. Our hope is to find values which are close to the

[1] `http://box2d.org/`

[2] `https://pybullet.org/wordpress/`

Data: $N, \boldsymbol{\lambda}_{init}, \boldsymbol{A}, \boldsymbol{b}$
Result: Compute the values of $\boldsymbol{\lambda}$, the convergence rate $\boldsymbol{\theta}$
for $k = 1$ ***To*** N **do**
  if $k = 1$ **then**
    $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda}_{init}$
  end
  $\boldsymbol{\lambda}_{old} \leftarrow \boldsymbol{\lambda}$
  for *all* i **do**
    $\boldsymbol{r}_i \leftarrow \boldsymbol{A}_{i*}\boldsymbol{\lambda} + \boldsymbol{b}_i$;
    $\boldsymbol{\lambda}_i \leftarrow \max(0, \boldsymbol{\lambda}_i - \frac{\boldsymbol{r}_i}{\boldsymbol{A}_{ii}})$;
  end
  $\boldsymbol{\theta}_k \leftarrow \max(|\boldsymbol{\lambda} - \boldsymbol{\lambda}_{old}|)$
end

Algorithm 1: $pgs(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{\lambda})$

final solution through deep learning. Then the values will be used as warm starting so that iterative solve can converge rapidly.

Chapter 3

Partcle-grid-particle

3.1 Overview

Deep learning methods have been widely applied in computer vision, like segmentation, objects recognition. And training data is always crutial to the whole learning process. Normally, researchers prefer using the original image as the input images for training. However, it is still not clear that what kind of data can be used for predictiting the motion of onjects. Researchers in DeepMind[1] found that when deep neural networks are applied in physics motion prediction, the most difficult thing is to recognize different objects and get time-state data, like m, $\boldsymbol{v}$, $\boldsymbol{q}$, etc. In their new paper, they provide different colors to different moving bodies and use a sequence of frames as training data to make deep neural networks get state information($\boldsymbol{v}, m, \boldsymbol{q}$) [7]. Although visual interact networks work well to predict physics motion, it would be confused when it is facing many bodies dynamic system. In our view, we do not need deep learning model(e.g. convolutional neural networks) to replace the contact solver completely. We only hope deep neural network can accelerate the simulation. The learning model can give reasonable values which are close to the final solutions. Afterward, the iterative contact solver will use the values as starting, and ideally coverage

[1] `https://deepmind.com/`

rapidly.

After discussion, we decided to transform each time-state to a grid map by Smoothed Particle Hydrodynamics. Then we will use the grid maps as the training data for deep learning model. Overall, the advantages of using the grid-based method are,

1. Grid map can describe the mass distribution so that neural networks can understand the distribution of objects. In other words, it is possible for deep neural networks to recognize the objects in the simulation.

2. Grid map image can restore assessable data(mass, linear velocity, angular velocity) for deep learning neural networks, while the visualization image of simulation can only describe the position of rigid. This is helpful deep neural networks to find the relationship between state and contact forces.

The basic method for generating training data which is more accessible to learning is that we will map a discrete element method(DEM) into a continuum setting use techniques from smooth particle hydrodynamics. Given a set of bodies $\mathcal{B}$ and a set of contacts between these bodies $\mathcal{C}$. The work process is like,

1. Based on Smoothed Particle Hydrodynamics(SPH), map current state(m, v_x, v_y, ω, n_x) to a image(the number of channel is 5.), which is called *feature image*

2. The *feature image* will be used as input to a model(created by a convolutional neural network and introduced in Chapter 4), then one image(the number of channels is 2) will be got, which can be called *label image*.

3. For all contacts positions, interpolated values will be generated based on *label image*. Then, the values will be used as starting iterate values for contact force solver. In our hypothesis, the given starting values will speed up the solver to reach convergence.

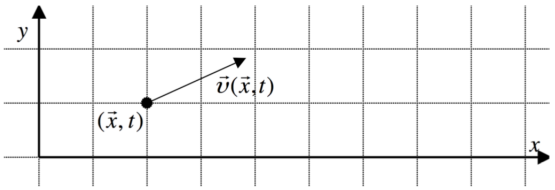


Figure 3.1: Grid description, *retrieved from MIT*(2011)

3.2 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) was invented to simulate nonaxisymmetric phenomena in astrophysics initially [8]. The principal idea of SPH is to treat hydrodynamics in a completely mesh-free fashion, in terms of a set of sampling particles. It turns out that the particle presentation of SPH has excellent conservation properties, energy, linear momentum, angular momentum, mass, and velocity.

3.2.1 Fundamentals

The heart of SPH is a kernel interpolation method which allows any function to be expressed in terms of its values at a set of disordered points - the particles[9]. For ant field $A(\mathbf{r})$, a smoothed interpolated version $A_I(\mathbf{r})$ can be defined by a kernel $W(\mathbf{r}, h)$,

$$A_I(\mathbf{r}) = \int A(\mathbf{r}')W(\|\mathbf{r} - \mathbf{r}'\|, h)\,\mathrm{d}\mathbf{r}' \tag{3.1}$$

where the integration is over the entire space, and W is an interpolating kernel with

$$\int W(\|\mathbf{r} - \mathbf{r}'\|, h)\,\mathrm{d}\mathbf{r}' = 1 \tag{3.2}$$

and

$$\lim_{h\to 0} W(\|\mathbf{r} - \mathbf{r}'\|, h)\,\mathrm{d}\mathbf{r}' = \delta(\|\mathbf{r} - \mathbf{r}'\|) \tag{3.3}$$

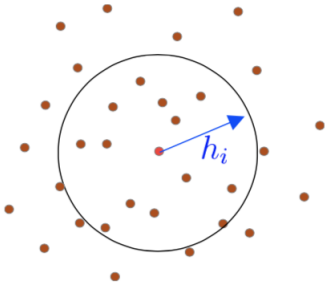


Figure 3.2: Visilaztion of SPH

Normally, we want the kernel to be non-negative and rotational invariant, which can be written mathematically

$$W(\|\mathbf{x}_i - \mathrm{x}_j\|, h) = W(\|\mathrm{x}_j - \mathrm{x}_i\|, h) \tag{3.4a}$$

$$W(\|\mathbf{r} - \mathbf{r}'\|, h) \geq 0 \tag{3.4b}$$

For numerical work, we can use the midpoint rule,

$$A_I(\mathbf{x}) \approx A_S(\mathbf{x}) = \sum_i A(\mathbf{x}_i) W(\|\mathbf{x}_i - \mathbf{x}\|, h) \Delta V_i \tag{3.5}$$

Since $V_i = m_i/\rho_i$

$$A_S(\mathbf{x}) = \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) W(\|\mathbf{x}_i - \mathbf{x}\|, h) \tag{3.6}$$

The default, gradient and Laplacian of A are:

$$\nabla A_S(\mathbf{x}) = \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) \nabla W(\|\mathbf{x}_i - \mathbf{x}\|, h) \tag{3.7a}$$

$$\nabla^2 A_S(\mathbf{x}) = \sum_i \frac{m_i}{\rho_i} A(\mathbf{x}_i) \nabla^2 W(\|\mathbf{x}_i - \mathbf{x}\|, h) \tag{3.7b}$$

3.2.2 Smoothing Kernels

Smoothing kernels function W is one of the most important points in SPH. Stability, , and speed of the whole method depend on these

functions. For different purposes, researchers will normally determine different kernels. One possibility for W is a Gaussian. However, most current SPH implementations are based on kernels with finite support. So, I mainly introduce **poly6** and **spiky** kernels here. In the chapter 5, I will compare the different kernels and analyze which one is better.

Poly6

The kernel is also known as the 6th degree polynomial kernel.

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.8}$$

Then, the gradient of this kernel function can be

$$\nabla W_{poly6}(\mathbf{r}, h) = -\frac{945}{32\pi h^9} \begin{cases} \mathbf{r}(h^2 - \|\mathbf{r}\|^2)^2 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.9}$$

The Laplacian of this kernel can be expressed by,

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = -\frac{945}{16\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)(3h^2 - 7\|\mathbf{r}\|^2) & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.10}$$

SPH is normally used in the liquid simulation. Müller suggested in [10], this kernel can be used for the computation of pressure forces. Particles tend to build cluster under high pressure because 'as particles get very close to each other, the repulsive force vanishes because the gradient of the kernel approaches zero to the center.'. We can see that in Figure 3.4. Another similar kernel, spiky kernel, is proposed by Desbrum and Gascuel[11] to solve this problem.

Spiky

The kernel proposed by Desbrum and Gascuel[11]

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\mathbf{r}\|)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.11}$$

Then, the gradient of the spiky kernel can be described by,

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45\mathbf{r}}{\pi h^6 \|\mathbf{r}\|} \begin{cases} (h - \|\mathbf{r}\|)^2 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.12}$$

The laplacian of spiky can be expressed by,

$$\nabla^2 W_{spiky}(\mathbf{r}, h) = \frac{90}{\pi h^6} \begin{cases} h - \|\mathbf{r}\| & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.13}$$

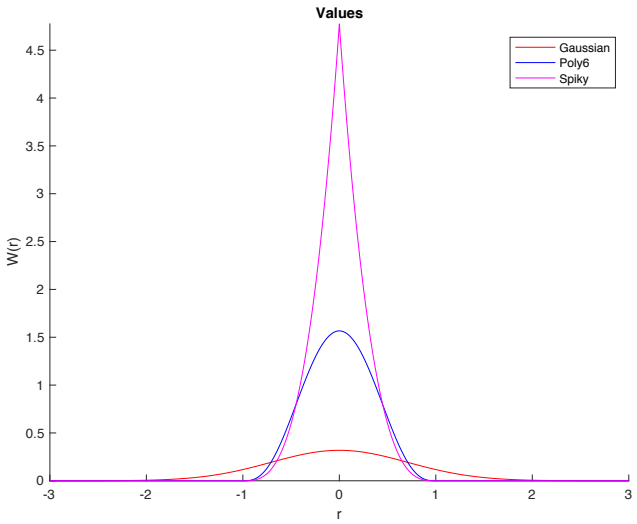


Figure 3.3: Comparation of different kernels, we set smoothing length $h = 1$ here.

3.2.3 Grid size and smoothing length

The grid should be also fine enough to capture the variation in our simulation. In this case, it is reasonable to have a grid fine enough such that no two contact points are mapped into the same cell.

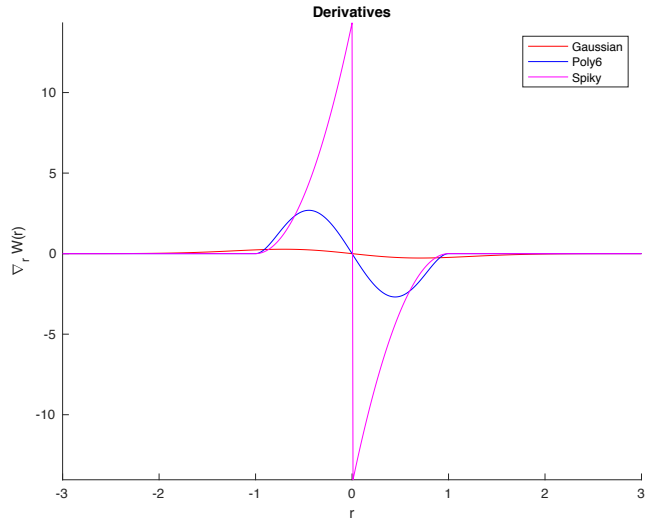


Figure 3.4: Comparation of gradient of different kernels, we set $h = 1$ here.

Smoothing length, h, is another one of the most important parameters, which affects the whole SPH method by changing the kernel value results and neighbor searching results. Too small or too big values might cause lose essential information in the simulation.

3.2.4 Neighbor Search

Neighbor search is one of the most crucial procedures in SPH method considering all interpolation equations, $A(\mathbf{r})$, needs the neighbor list for every particle (refer to equation 3.7). A naive neighbor searching approach would end up with a complexity of $O(n^2)$. The complexity is not good enough since it is impossible to reach any interactive speed when the particle count increases. It is possible to have a significant performance increase with using an efficient nearest neighbor searching(NNS), since NNS is the most time-consuming process in SPH computation. In order to decrease the complexity, we choose to use $k - d$ tree data structure to store the particle spatial infor-

mation and then do the nearest neighbor searching. $k-d$ tree is a data structure proposed by Bentley and Jon [12], and we will use the $k-d$ algorithm built in *Scipy*[2]

3.2.5 Summary

Then, I can conclude the whole process for map current simulation state to grid images,

Data: Given a set of bodies $\mathcal{B}$ and the state in time t, as well as all the spacial positions of grid nodes $\mathbf{x}$

Result: the body grid image $\boldsymbol{G}_{\mathcal{B}}$

for *all* i *and* j **do**

1. Find the nearest neihbors $\mathcal{B}_{near} \subseteq \mathcal{B}$ around $\mathbf{x}_{ij}$
2. read the current state,

$$m_k, \boldsymbol{v}_k, \boldsymbol{q}_k, \omega_k \quad k \in \mathcal{B}_{near}$$

3. Define state vector

$$\boldsymbol{S}_k \leftarrow [m_k, v_{kx}, v_{ky}, \omega_k]$$

4. Compute grid values

$$\boldsymbol{G}_{\mathcal{B}}(i,j) \leftarrow \sum_{k \in \mathcal{B}_{near}} W(\mathbf{x}_{ij}, \boldsymbol{q}_k) \boldsymbol{S}_k$$

end

Algorithm 2: Mapping bodies into a grid image. It can be called by $Bodies2grid(\mathbf{x}, \mathcal{B})$

[2] `https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html`

Data: Given a set of contacts $\mathcal{C}$ between a set of bodies $\mathcal{B}$ and the state in time t, as well as all the spacial positions of grid nodes $\mathbf{x}$

Result: the contact grid image $\boldsymbol{G}_\lambda$

for *all* i *and* j **do**

1. Find the nearest neihbors $\mathcal{C}_{near} \subseteq \mathcal{C}$ around $\mathbf{x}_{ij}$
2. read the current contact forces values and its position.

$$\boldsymbol{q}_k, \boldsymbol{\lambda}_k \quad k \in \mathcal{C}_{near} \quad \boldsymbol{\lambda} = [\lambda_n, \lambda_t]$$

3. Compute grid values

$$\boldsymbol{G}_\lambda(i,j) \leftarrow \sum_{k \in \mathcal{C}_{near}} W(\mathbf{x}_{ij}, \boldsymbol{q}_k)\boldsymbol{\lambda}_k$$

end

Algorithm 3: Mapping contacts into a grid image. It can be called by $Contacts2grid(\mathbf{x}, \mathcal{C})$

3.3 Grid to particle

SPH will be used for us to transform current state of dynamic system to grid images. After obtaining the grid image for simulation state in time t, we will use the grid cells as input and renew the contact grid image based on trained model. After getting the contact force grid image, the next step is to use contact position to interpolate image values. The interpolated values will be stored in the contact points and used as starting iterates for contact force solver. Then we can update states of all rigid bodies in time $t + \Delta t$.

3.3.1 Bilinear interpolation

We applied bilinear interpolation in our case, since we did mainly research on a rectilinear $2 - D$ grid. The key idea is to perform linear interpolation first in one direction, and then again in the other direction. Although each step is linear in the sampled values and

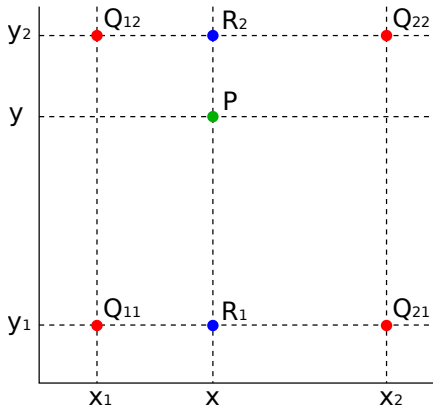


Figure 3.5: The figure shows the visualization of bilinear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate.

in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

As shown in Figure 3.5, We have known $Q_{a,b} = (x_a, y_b)$ and $a \in \{1, 2\} \quad b \in \{1, 2\}$. Then, we can firstly do linear interpolation in the x-direction. This yields

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \tag{3.14a}$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \tag{3.14b}$$

After getting the two values in $x-$direction $f(x, y_1)$ and $f(x, y_2)$, we can combine these values to do interpolation in $y-$ direction.

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{3.15}$$

Combine $f(x, y_1)$ and $f(x, y_2)$ defined in equation 3.14, we can get,

$$
\begin{aligned}
f(x,y) \approx & \frac{y_2-y}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})\right) \\
& + \frac{y-y_1}{y_2-y_1}\left(\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})\right) \\
= & \frac{1}{(x_2-x_1)(y_2-y_1)}(\\
& f(Q_{11})(x_2-x)(y_2-y)+f(Q_{21})(x-x_1)(y_2-y) \\
& + f(Q_{12})(x_2-x)(y-y_1)+f(Q_{22})(x-x_1)(y-y_1)) \\
= & \frac{1}{(x_2-x_1)(y_2-y_1)}\begin{bmatrix} x_2-x & x-x_1 \end{bmatrix} \\
& \cdot \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix}
\end{aligned}
\tag{3.16}
$$

Then, the bilinear interpolation algorithm can be summarized,

Data: A grid image $\boldsymbol{G}$, given a position (x, y), as well as all the spacial positions of grid nodes $\mathbf{x}$

Result: The interpolated value $\boldsymbol{p}$

1. Find the cell $(\mathbf{x}_{i,j}, \mathbf{x}_{i+1,j}, \mathbf{x}_{i+1,j+1}, \mathbf{x}_{i,j+1})$covering (x, y)
2. $$(x_1, y_1) \leftarrow \mathbf{x}_{i,j} \quad \text{and} \quad (x_2, y_2) \leftarrow \mathbf{x}_{i+1,j+1}$$
3. $$\boldsymbol{p} \leftarrow \frac{1}{(x_2-x_1)(y_2-y_1)}\begin{bmatrix} x_2-x & x-x_1 \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{G}_{i,j} & \boldsymbol{G}_{i+1,j} \\ \boldsymbol{G}_{i,j+1} & \boldsymbol{G}_{i+1,j+1} \end{bmatrix} \begin{bmatrix} y_2-y \\ y-y_1 \end{bmatrix}$$

Algorithm 4: Caculation interpolated value in $\boldsymbol{G}$. It can be called by *interpolate*$(\boldsymbol{G}, \mathbf{x}, (x, y))$

3.4 Experiment and Conclution

Our hope is that starting iterates will be close to "solution" of the contact problem, which can indicate the contact force solvers will coverage very rapidly or maybe not even need to iterate. In order to test whether the SPH method can be applied in our case, the contact force solution is mapped to image and interpolated values are generated and used to restart the contact force solver. Our hypothesis is that the iterative solver quickly recovers an iteration close to the original solution before mapping to force grid image. I will just compare the performance of SPH-based method with other methods in this section. More details will be analyzed and described in Chapter 5.

3.4.1 pybox2d simulation

In order to test whether **SPH-based** method works for this case. All experiments will be done based on the basis physical engine, *pybox2d*. Before testing the performance of **SPH-based** method, **Non-model**, **Builtin-model** and **Copy-model** are defined to see the influence of different initial values for iterative contact solver.

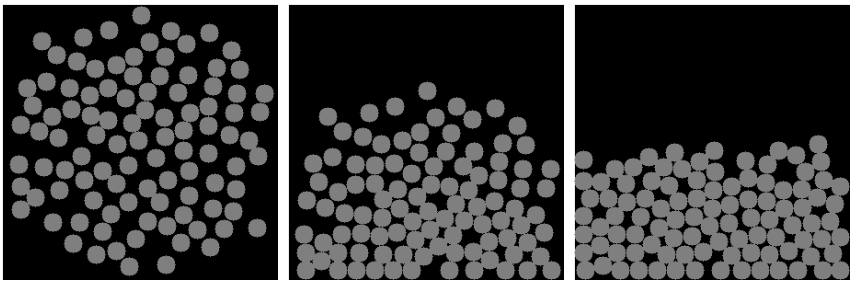
- **Non-Model** In each step of the simulation, before the contact solver starts iteration to make a resolution for current dynamic system equation, the initial λ_f and λ_t of every contact will be given value 0.
- **Builtin-Model** In each step of the simulation, before the contact solver starts iterations to make a resolution for current dynamic state equation, the initial value of λ_f and λ_t will be determined by the built-in algorithm. In other words, this model is the default solution built in *pybox2d*.
- **Copy-Model** In each step of the simulation before the contact solver starts iteration to make a resolution for current dynamic

system equation, the initial value of λ_f and λ_t will be the actual solution after exact iterations solver.

After defining these models, each model will be applied in the same rigid dynamic simulation process. The setting is given below,

- **World Setting**, the world box size is 30×30, and there are 100 circle rigids($r = 1$, all circle rigid bodies in the same size.) inside the box. Initially, the rigid circles will be located following gaussian distribution[3]. Then, all rigid circles will fall down by gravity. The visualization of simulation is shown in Figure 3.6.

- **Simulation Setting**, there will be totally 600-steps simulation. For each step, $\Delta t = 0.01s$, and the number of iteration in each step will be set as fixed, 3000. Then I will use the average convergence rate to show how fast the model coverages.



(a) Time Step=0 (b) Time Step=200 (c) Time Step=400

Figure 3.6: Visualization for experiment simulation

The results are described in Figure 3.7. Then, we can get some basic conclusions,

- **Copy-Model** performs the best, nearly no iteration to get convergence, since the correct solution would be given to the solver for start iterating.

[3] `https://en.wikipedia.org/wiki/Normal_distribution`

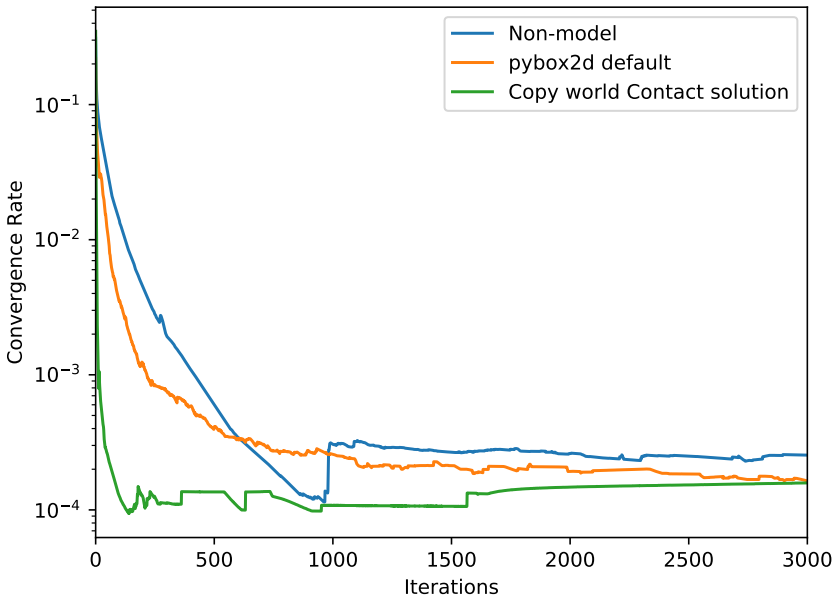


Figure 3.7: Average convergence rate for different models(not including **SPH-based model**).

- If all the starting values for iterative solver are zeros, it has to take a long time to reach convergence, which is indicated by **Non-Model**.
- The **Builtin-Model** performs similarly with **Non-Model**, far slower than **Copy-Model**.

Then, we can make a hypothesis,

- **Hypothesis** Ideally, if SPH-based method can work well for this project, the contact solver would coverage faster than **Builtin-Model** and **None-Model**, but still slower than **Copy-Model**, which give the correct solution to solver directly.

3.4.2 SPH-based method test

The **SPH-Model** is defined as follow, as well as the experiment algorithm in Algorithm 5,

- **SPH-Model** In each step of simulation, one grid map $G(\boldsymbol{\lambda})$ $\boldsymbol{\lambda} = [\lambda_n, \lambda_t]$ will be created based on given rigid bodies $\mathcal{B}$ and contacts $\mathcal{C}$ between the bodies. Then, interpolated values would be generated by given contact point position (x_j, y_j) $j \in \mathcal{C}$ and would be used as initial values for iterative contact solver in *pybox2D*.

Data: contacts $\mathcal{C}_t$ $t \in T$ between these bodies. Contacts $\mathcal{C}_t$ are from a copy world, which will run the simulation of time t in advance.

Result: Get the contact forces $\boldsymbol{\lambda}_t$ and make simulation keep running.

for *all* t *in* T **do**

1. Read $\mathcal{C}_t$ 2. map the solved contacts to a gird image,

$$\boldsymbol{G}_\lambda \leftarrow Contacts2grid(\mathbf{x}, \mathcal{C}_t)$$

3. Once the contact force image $\boldsymbol{G}_\lambda$ is obtained, then

for *all* $j \in \mathcal{C}$ **do**

$$\boldsymbol{\lambda}_j \leftarrow interpolate(\boldsymbol{G}_\lambda, \mathbf{x}, \boldsymbol{q}_j)$$

end

4. Using $\boldsymbol{\lambda}$ as a warm starting to **restart** the iterative solver,

$$\boldsymbol{\lambda} \leftarrow Solver(\boldsymbol{\lambda})$$

5. Updating the simulation world and $t \leftarrow t + \Delta t$

end

Algorithm 5: Algorithm describeing how **SPH-Model** works for dynamic simulation.

Finally, the plot about the As the hypothesis I supposed, **SPH-Model** converges faster than **Non-Model**. This can demonstrate the **SPH-based** method is a good strategy, which can make iterative contact solver converge quickly.

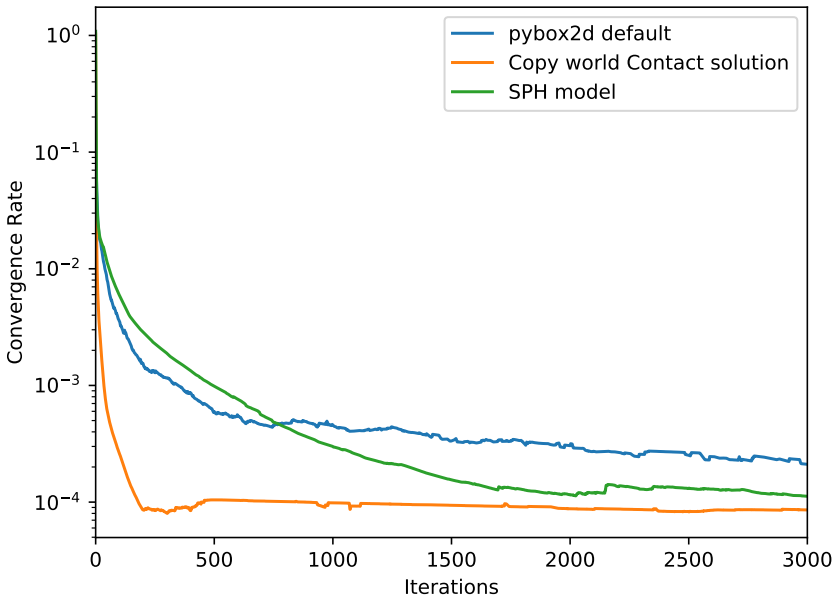


Figure 3.8: Convergence rate for different models(including **SPH-based model**).

3.4.3 Conclusion

Based the result in Figure 3.7 and 3.8, it can be concluded that **SPH-based** strategy is available for the case in this thesis. Once the correct contact forces grid is achieved, the interpolated values will help the iterative solver to get convergence faster. So, the next step is to train an available model based on training data. The overall algorithm is concluded in Algorithm 7.

Chapter 4

Deep Learning

This chapter consists of four sections.

- **Section** 4.1, a brief introduction to deep learning neural networks, including its history, development and current application.
- **Section** 4.2, decribing details of about Convolutional Neural Networks, including different types of layers and their functionas.
- **Section** 4.3, talking about the techqiues used in deep learning training process, which make training get convagence faster and more accurate.

4.1 Introduction

As a subdiscipline of machine learning, deep learning attempts to perform data abstraction at a high level via multiple processing layers. It is an algorithm based on the data representation learning of machine learning. Observations (e.g., an image) can appear in various forms, such as a vector of specific pixel intensity, or more abstractly, a series of edges, regions of a certain shape, etc. Learning tasks can be completed more easily by applying some special

representation methods (for example, facial recognition or facial expression recognition). The merit of deep learning lies in transforming manual acquisition into unsupervised or semi-supervised learning along with its efficient algorithms of hierarchical feature extraction [13].

Previous works on deep learning (or Cybernetics as called at that time) were first published during the 1940s to 1960s, which discusses biologically inspired models like Perceptron, Adaline, or Multi Mayer Perceptron [13], [14]. Then, from the 1960s to 1980s backpropagation was invented and the second wave named Connectionism followed up cite rumelhart1986learningsn. This algorithm has been active up to now and is currently utilized for optimization of Deep Neural Networks. Convolutional Neural Networks (CNNs) is among the most outstanding contributions, which is designed to identify visual patterns of relative simplicity, such as handwritten characters [15]. Finally, the appearance of more complex architectures in 2006 marks the arrival of deep learning's modern era[16]–[18]. Breakthroughs in the processing of speech and natural language (2011) as well as image classification at ILSVRC[1] the scientific competition (2012) have made deep learning a conqueror in many Machine Learning communities like Reddit and winner of challenges beyond conventional applications[2].

Especially, deep learning has enormously impacted the field of computer vision by achieving unprecedented performance on tasks of image classification, objects detection, image segmentation, image captioning and so on in the past four years [8]. Such huge progress is primarily built upon the enlargement of computational resources such as frameworks (e.g.s TensorFlow[3]), modern GPU implementations (e.g. Cudnn[4]), increasing amount of available annotated data, and participation in open source codes and share models based on

[1] http://www.image-net.org/challenges/LSVRC/

[2] http://blog.kaggle.com/2014/04/18/winning-the-galaxy-challenge-with-convnets

[3] https://www.tensorflow.org/

[4] https://developer.nvidia.com/cudnn

community. Thanks to these advances, more audience is accessible to the expertise that is necessary for the training of modern convolutional networks. Higher accuracy can be achieved through training of larger and deeper architectures on greater datasets every year, while phenomenal outcomes have been exhibited by the already trained ones when they are transferred to smaller datasets or evaluated with varied visual tasks.

The increasing popularity of deep learning in various research fields enables it the replacement of some conventional approachesg. The computer vision section gives many successful applications, such as image segmentation[19],objection recognition[20].

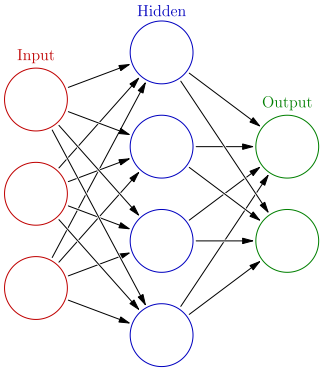


Figure 4.1: visulization of one simple 3-layers neural networks, including input layer, hidden layer and output layer, *retrieved from Wikipedia*

4.2 Convolutional Neural Networks

An input layer, an output layer, and multiple hidden layers make up the complete convolutional neural network. Normally, the hidden layers include convolutional layer, pooling layers, fully connected layers, and normalization layers.

The process in neural networks is described classically as a convolution, while it is a cross-correlation rather than a convolution in

terms of math. Only indices in the matrices are meaningful, which are thus assigned with weights.

4.2.1 Convolutions

An effective solution to such issue by utilizing the structure of image-encoded information is to assume that compared with the pixels at the opposite corners of the image, the spatially closer ones can construct some certain feature of interest cooperatively. Further, an important (smaller) feature in image label definition tends to possess equal importance wherever it is within the image.

A convolution operator is entered. For a given two-dimensional image, I, and a small matrix, $\boldsymbol{K}$ with size $h \times w$, (known as a convolution kernel), which is presumptively encoded with an extraction way of an interesting image feature, the convolved image, $\boldsymbol{I} * \boldsymbol{K}$, is computed by overlaying the kernel on image top in every possible way and recording the sum of elementwise products between the image and the kernel:

$$(\boldsymbol{I} * \boldsymbol{K})_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} \boldsymbol{K}_{ij} \cdot \boldsymbol{I}_{x+i-1,y+j-1} \tag{4.1}$$

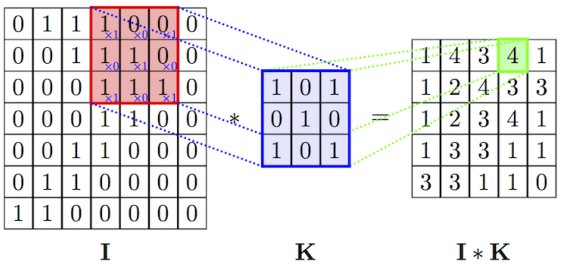


Figure 4.2: One simple example of convolution.

4.2.2 Convolutional layers

Normal RGB images are represented by matrices containing color information in the form of Red-Gray-Blue color codes. An image

therefore has size $h \times w \times d$, where d is the number of channel of image, in normal RGB images, $d = 3$. However, in the case of this thesis, channels consist of $[m, v_x, v_y, \omega, n_x]$, therefore in its case, $d = 5$. Convolutional layers are essential layers in CNNs, producing feature maps from input images or lower level feature maps.

Convolutional layers include a kernel (or filter). Let K be a kernel with x rows, y columns and depth d. Then the kernel with size $(K_x \times K_y \times d)$ works on a receptive field $(K_x \times K_Y)$ on the image. The kernel height and width are smaller than the input image height and width. The kernel slides over (convolves with) the image, producing a feature map (Figure 4.2). Convolution is the sum of the element-wise multiplication of the kernel and the original image. Note that the depth d of the kernel is equal to the depth of its input. Therefore, it varies within the network. Usually, the depth of an image is the number of color channels, the three RGB channels. In this case, $d = 5$.

The kernel stride is a free parameter in convolutional layers which has to be defined before training. The stride is the number of pixels by which the kernel shifts at a time. A drawback of using convolutional layers is that it decreases the output map size. A larger stride will result in a smaller sized output. Equations 4.2 show the relationship between output size O and input size of an image I after convolution with stride s and kernel K. Furthermore, the feature map size decreases as the number of convolutional layers increases. Row output size O_x and column output size O_y of convolutional layers are determined as follows:

$$\begin{cases} O_x = \frac{I_x - K_x}{s} + 1 \\ O_y = \frac{I_y - K_Y}{s} + 1 \end{cases} \tag{4.2}$$

As an example, an image of size $(32 \times 32 \times 3)$, a kernel of size $(3 \times 3 \times 3)$ and a stride $s = 1$ result in an activation map of size $(30 \times 30 \times 1)$. Using additional n kernels, the activation map becomes $(30 \times 30 \times n)$. So, additional kernels will increase the depth of the

convolutional layer output. Online resources can be referred to for more animation examples with different types of convolution[5].

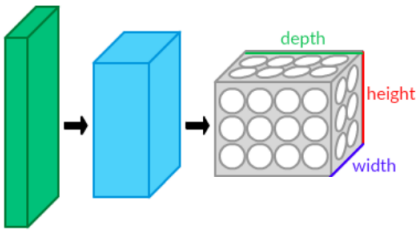


Figure 4.3: Visualization of convolutions network, *retrieved from Wikipedia*

4.2.3 Activation Layer

Essentially, whether a neuron should be activated is decided by the activation functions. As a feature for the artificial neural networks of prime importance, they are in charge of distinguishing the relevant signals received by neuron from the irrelevant ones. Normally, we can express the general function as Equation 4.3. And one mathematical model is shown in Figure 4.4 to describe how activation function is.

$$y = f_{Activation}(\sum_{i}(w_i \cdot x_i) + bias) \tag{4.3}$$

By transforming the input signal, the nonlinear activation function can be obtained, which, in turn, is subsequently sent to the next layer of neurons as an input. Rectified Linear Unit (ReLU) is considered as the activation function that enjoys the most pervasive application nowadays. More details are presented below.

Rectified Linear Unit

ReLU commonly appears in the mathematical form as follows,

$$y = max(0, x) \tag{4.4}$$

[5] `https://github.com/vdumoulin/conv_arithmetic`

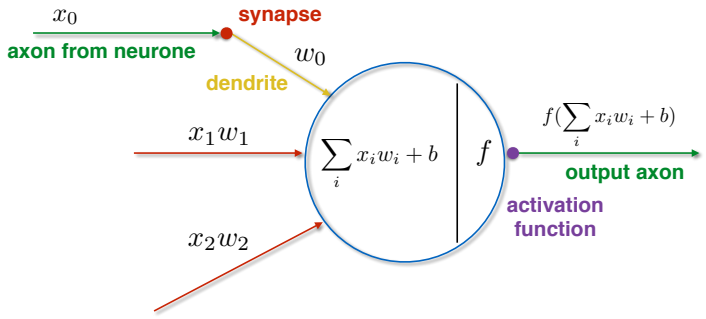


Figure 4.4: Mathematical model for describing activation function

Thanks to its linear non-saturating form (e.g. a factor of 6 in [21]), a great accelerating effect on converging the stochastic gradient descent were discovered for ReLU when compared with the *sigmoid*/*tanh* functions. Since then, it has become a very favorable approach. Vanishing or exploding gradient can hardly affect it actually. Further, it is much more economical than the conventionals exponentials. Nonetheless, ReLU cannot cater for all datasets and architectures, for it may get rid of every piece of negative information.

4.2.4 Pooling Layer

Pooling layers are also known as downsampling layers. A commonly used pooling is max pooling (figure 4.5). The downsampled output is produced by taking the maximum input value within the kernel, resulting in the output with a decreased size. There are several other methods which are commonly used in neural networks, such as average pooling and L2-norm pooling. Average pooling used to be a common method, while it is quitting the historical arena these days due to the more practical mmaxpooling[22].

There are two important arguments for implementing pooling layers,

1. Decreasing the number of weights.

2. Decreasing the chance of overfitting the training data.

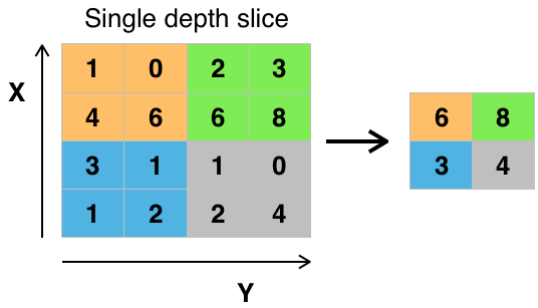


Figure 4.5: Maxpooling with a (2×2) kernel and stride $s = 2$. Maxpooling layers reduce spatial dimension of the input [23]

4.2.5 Fully Connected Layer

Following the steps of convolutional and maxpooling layers, fully connected layer realizes the high-level reasoning in neural networks eventually. .As what a regular neural network shows, all activations in the above-mentioned layers are connected by the neurons at a fully connected layer. A matrix multiplication and the follow-up bias offset will then complete all the calculations of these activations.

4.2.6 Batch Normalization

Because of its assistance in faster convergence, this layer shoots to popularity rapidlyr[24]. By adding a normalization step of shifting inputs to zero-mean and unit variance, it achieves the comparability across features of inputs in every trainable layer. Consequently, a high learning rate is guaranteed during the network learning process.

Besides, activation functions like TanH and Sigmoid are protected by it from getting stuck in saturation mode (e.g. gradient equal to 0).

4.3 Training Method

4.3.1 Loss Function

The value of the loss function L represents the difference between the training image after it has propagated through the network and desired annotated output image.

Two assumptions are made about this loss function.

1. It is able to be defined as the average over the loss functions for individual training images, as the training often is carried out in batches.

2. It should be able to be defined as a function of the network outputs.

Below a brief overview is given of some widely used loss functions, where $f_\theta(x_i)$ are the neuron outputs and y_i are the desired outputs.

Quadratic Cost Function

The Mean Squared Error (MSE) cost function is one of the simplest cost functions. Normally it will be used in estimation problems[25].

$$L = \frac{1}{N}\sum_{n=1}^{N}(f_\theta(x_i) - y_i)^2 \tag{4.5}$$

4.3.2 Overfitting

Overfitting is a problem that arises in neural network training. When a model is overfitted to the training data, it loses its capability of generalization. The model has learned the training data, including noise, to such a great extent that it has failed to capture underlying general information. CNNs have a large number of weights to be trained, therefore overfitting can occur due to training too few training examples.

Regularization L2

As the first primary route to averting overfitting, the classical method of weight decay increased one more term to the cost function as a penalization for parameters in each dimension, so that the network is prevented from exact modeling of the training data and generalization of new examples is thus ensured.

$$Error(x, y) = Loss(x, y) + \sum_{i} \theta_i \tag{4.6}$$

where θ is with a vector containing all parameters in the network.

Data augmentation

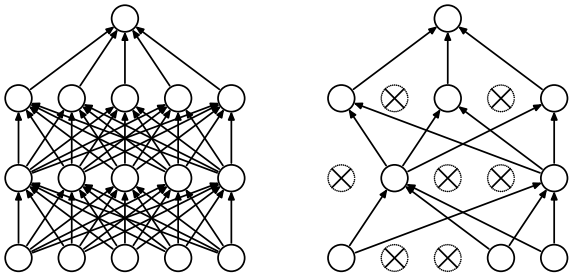
The size of the training set can be enlarged by data augmentation in order to prohibit memorizing of the whole set by the model. A few types may be involved depending on the dataset. For instance, objects supposed to be constantly rotating, such as galaxies or planktons, will fit different kinds of rotations to the original images.

Dropout

Dropout layers[26] are a tool to prevent overfitting (Figure 4.6). In dropout, nodes and its connections are randomly dropped from the network. Dropout constrains the network adaptation to the training set, consequently it prevents that the weights are not too much fitted for this data. Training data will thus have reduced performance difference with validation date. Dropout layers are used during training only, not during validation or testing. Nowadays, dropout method has been the main method to prevent overfitting.

Early Stopping

Early Stopping is another normal way to prevent overfitting. It makes the training stop before it is to be overfitted by the model, which is practically useful during the training process of neural networks.



(a) Standard Neural Net (b) After applying dropout.

Figure 4.6: A nerual network structure before and after applying dropout

4.3.3 Stochastic Gradient Descent Variants

In both Gradient Descent (GD) and Stochastic Gradient Descent (SGD) parameters are updated according to an update rule to minimize a loss function in an iterative manner. Computing the exact gradient using GD in large datasets is expensive (GD is deterministic), as this method runs through all training samples to perform a single update for one iteration step. In Stochastic Gradient Descent (or on-line Gradient Descent) an approximation of the true gradient is computed. This is done by using only one or a subset of training samples for a parameter update. When using a subset of training samples, this method is called mini-batch SGD.

SGD is a method to minimize the loss function $L(\theta)$ parametrized by θ. This is achieved by updating θ in the negative gradient direction of the loss function $\nabla_\theta L(\theta)$ with respect to the parameters, in order to decrease the loss function value. The learning rate η determines the step size to get to the local or global minimum.

$$\theta_{n+1} = \theta_n - \eta \nabla_{\theta_n} L(f_{\theta_n}(x_i), y_i) \tag{4.7}$$

Mini-batch Stochastic Gradient Descent

This method performs an update for every mini-batch of n training samples. Mini-batch SGD reduces the variance of the parameter

updates. Larger mini-batches reduce the variance of SGD updates by taking the average of the gradients in the mini batch. This allows taking bigger step sizes. In the limit, if each batch contains one training sample, it is the same as regular SGD.

Distributed SGD

In parallel computing environments, disturbed SGD is commonly applied for optimization. For a specific architecture, basically no difference in parameter values exists when training with different computers. The parameter space can thus be further explored, and improved performance will be obtained resultantly [27].

4.3.4 Learning Rate Scheduling in Gradient Optimization

There are serval variants of SGD available. Determining the appropriate learning rate, or step size, often is a complex problem. Applying too high learning rate cause suboptimal performance, while too low learning rate led to slow convergence. Learning rate scheduling is used as an extension of the SGD algorithm to improve performance. In learning rate scheduling, the learning rate is a decreasing function of the iteration number. Therefore, the first iterations have larger learning rate and consequently cause bigger parameter changes. Later iterations have similar learning rates, responsible for fine-tuning. I gave an overview of some gradient descent optimization algorithms,

Momentum

Momentum method is a method to speed up the SGD in the relevant direction. A fraction γ of the previous update is added to the current update. The mathematical details are defined in Equation 4.8.

$$\begin{aligned} v_n &= \gamma v_{n-1} + \eta \nabla_\theta L(\theta) \\ \theta &= \theta - v_n \end{aligned} \tag{4.8}$$

Nesterov Accelerated Gradient

The Momentum method does not take into the direction it is going in, while parameters' next position can be approximated via the Nesterov Accelerated Gradient method. The update rule is given in Equation 4.9.

$$\begin{aligned} v_n &= \gamma v_{n-1} + \eta \nabla_\theta L(\theta - \gamma v_{n-1}) \\ \theta &= \theta - v_n \end{aligned} \tag{4.9}$$

Adam

The Adaptive Moment Estimation (Adam) optimizer [28] determines an adaptive learning rate for each parameter. Apart from the decaying average for past squared gradients v_n, an exponentially decaying average is also maintained by Adam for the past gradients m_n. Estimates for the mean and the uncentered variance of the gradients are Vectors v_n and m_n, respectively, and both of them are biased towards zero. Bias-corrected estimates $\hat{v}_t$ and $\hat{m}_t$ are computed for the update rule in Equation **??**.

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{\hat{v}_n} + \epsilon} \cdot \hat{m}_n \tag{4.10}$$

4.3.5 Backpropagation

The CNN requires to adjust and update its kernel parameters, or weights, for the given training data. Backpropagation[29] is an efficient method for computing gradients required to perform gradient-based optimization of the weights in neural networks [19]. Minimizing the loss function (or error function), the weights are combined together on purpose for solving the problem of optimization. Herein, the gradient of error functions needs computing during each iteration, which requires continuity and differentiability of the loss functions for all iteration steps.

The initial weights of an untrained CNN are randomly chosen. Consequently before training, the neural network cannot make meaningful predictions for network input, as there is no relation between an image and its labeled output yet. By exposing the network to a training data set, comprising images and their labeled outputs with correct classes, the weights are adjusted. Training is the adaptation of the weights in such way that the difference between desired output and network output is minimized, which means that the network is trained to find the right features required for classification. There are two computational phases in a neural network, the forward pass and the backward pass in which the weights are adapted.

Forward pass

An image is fed into a network. The first network layer outputs an activation map. Then, this activation map is the input to the first hidden layer, which computes another activation map. Using the values of this activation map as inputs to the second hidden layer, again another activation map is computed. Carrying out this process for every layer will eventually yield the network output.

Backward pass

In this phase, the weights are updated by backpropagation. One epoch of backpropagation consists of multiple parts, usually multiple epochs are carried out for a training image:

1. **Loss Function** In the forward pass, the inputs and desired outputs are presented. A pre-defined loss function L is used to minimize the disparity when comparing the input and desired output. The goal is to adjust the weights so that the loss function value decreases, this is achieved by calculating the derivative with respect to the weights of the loss function.

2. **Backward pass** During the backward pass, the weights that have contributed the most to the loss are chosen in order to adjust them so that the total loss decreases.

3. **Weight update** In the final part all weights are updated in the negative direction of the loss function gradient.

Hence, gradient computation for loss functions in terms the network weights is actually the core of backpropagation problem. Computing the partial derivative $\frac{\partial L}{\partial \omega}$ is essential (carried out in the backward pass) to minimize the loss function value. Stochastic Gradient Descent (SGD) is the most common way to optimize neural networks.

Backpropagation Example for a Multi-Layer Network

I describe the details of the backpropagation algorithm for one simple example in Figure 4.7. The cost function L is given below, el is the error between the true output d_l and network output y_l. The network output y_l is computed in the forward pass and depends on outputs of the previous layer v_j and the output layer weights w_j^o. Some mathesmatical equations we can get:

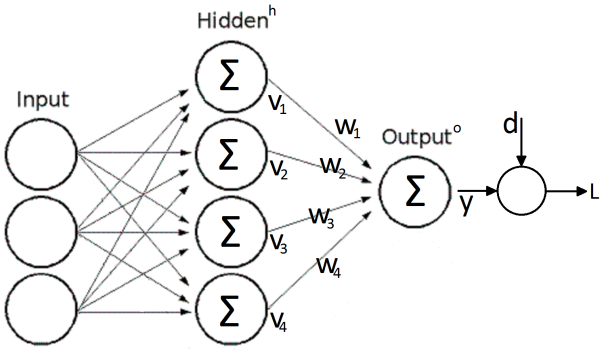


Figure 4.7: The example for showing the details of backpropagation

$$\begin{aligned} L &= (e_l)^2 \\ e_l &= d_l - y_l \\ y_l &= \sum_j w_j^o v_j \end{aligned} \tag{4.11}$$

The Jacobian is given by:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial e_l} \cdot \frac{\partial e_l}{\partial y_l} \cdot \frac{\partial y_l}{\partial w_j} \tag{4.12}$$

Then combining Equation 4.11 and 4.15, we can caculate that,

$$\frac{\partial L}{\partial w_j^o} = -2 v_j e_l \tag{4.13}$$

Using SGD updating rules **Momentum** mentioned in section 4.3.3 Equation 4.8, the output weights are updated using:

$$w_j^o(n+1) = w_j^o(n) + \alpha(n) v_j e_j \tag{4.14}$$

$\alpha(n)$ stands for the learning rate in nth-iteration, you can see more details in section 4.3.3.

After having updated the output weights, the weights in the hidden layers can be updated. As it is a backward pass, first gradients of the output layers are computed, then the gradients of the hidden layers. The Jacobian is given as follows:

$$\frac{\partial L}{\partial w_{ij}^h} = \frac{\partial L}{\partial e_l} \cdot \frac{\partial e_l}{\partial y_l} \cdot \frac{\partial y_l}{\partial v_j} \cdot \frac{\partial v_j}{\partial w_{ij}^h} \tag{4.15}$$

Based on the networks shown in Figure 4.7, we can define v_j based on $w_i j$ and input x_i

$$v_j = \sum_{i=1}^{N_{input}} x_i \cdot w_{ij}^h \tag{4.16}$$

Then combining Equation 4.15 and 4.16, we can calculate the Jacbian,

$$\frac{\partial L}{\partial w_{ij}^h} = -2 e_l w_j^o x_i \tag{4.17}$$

Which yields the update rule for the hidden layers:

$$w_{ij}^h(n+1) = w_{ij}^h(n) + 2\alpha(n) e_l w_j^o x_i \tag{4.18}$$

Finally the network is tested using a test dataset, this dataset contains data that differ from the ones in the training dataset. By increasing the amount of training data, the more training iterations are carried out, the better the weights are tuned.

Chapter 5

Results and Analysis

After introducing the basic knowledge which is available and used in the project, I will start the experiments and analyze the results. The main experiments would be,

1. Do the simulation based one computer physics library(*pybox2d*). Totally, 100 different rigid motion simulation would be carried out. For every simulation, I would record the fixed number of simulation steps, 600.

2. **XML** formulation would be used to store every step of all simulation. The data of each state would involve positions, velocities, contact forces, etc.

3. Once obtaining the **XML** files. I would transform the **XML** files into a *pybox2d* world, and then carry out some experiments to determine different grid size and smoothing length of SPH.

4. After choosing the proper parameter of the SPH kernel(types, grid cell size, and smoothing length), I would transform these **XML** files to grid images, which are used as training dataset.

5. Design a deep learning model and do the training on the training dataset.

6. Apply the trained model to predict the starting iterate values of contact forces($\boldsymbol{\lambda}$). Then I would compare the learning model with other classical methods.

5.1 Rigid Motion Simulation

pybox2d[1] is chosen as the main physics engine to implement computer simulation experiments. *pybox2d* is a 2D physics library for your games and simple simulations. It's based on the Box2D library, which is written in *C++*. It supports several shape types (circle, polygon, thin line segments), and quite a few joint types (revolute, prismatic, wheel, etc.). In my experiment, I would first try to apply deep learning in the simple simulations. So I would mainly simulation with circle rigid objects sharing the same radius.

5.1.1 Simulation Configuration

Simulation configs are listed below,

- **World Setting**, the world box size is 30×30, and there are 100 circle rigids($r = 1$, all circle rigid bodies in the same size.) inside the box. Initially, the rigid circles will be located following gaussian distribution[2]. Then, all rigid circles will fall down by gravity. The visualization of simulation is shown in Figure 5.1.

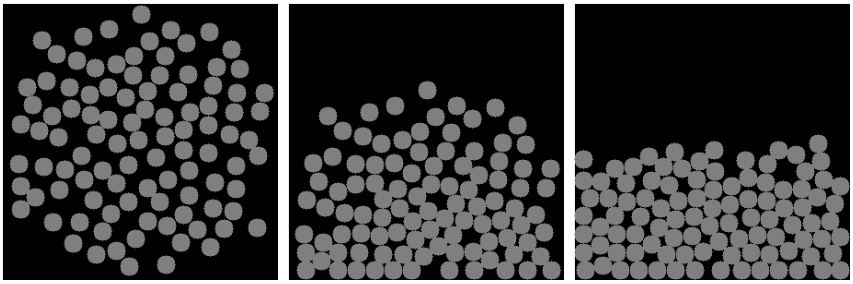
- **Simulation Setting**, there will be totally 600-steps simulation. For each step, $\Delta t = 0.01s$, and the number of iteration in each step will be set as fixed, 3000. Then I will use the average convergence rate to describe the performances of models.

[1] `https://github.com/pybox2d/pybox2d`

[2] `https://en.wikipedia.org/wiki/Normal_distribution`

5.1.2 Simulation Details

Before generating data, one dynamic simulation was run to check how *pybox2d* works and some figures have been obtained. Figure 5.2a describe the relationship between time step and contacts number, and Figure 5.2b gives the relationship between time spend and contacts number.

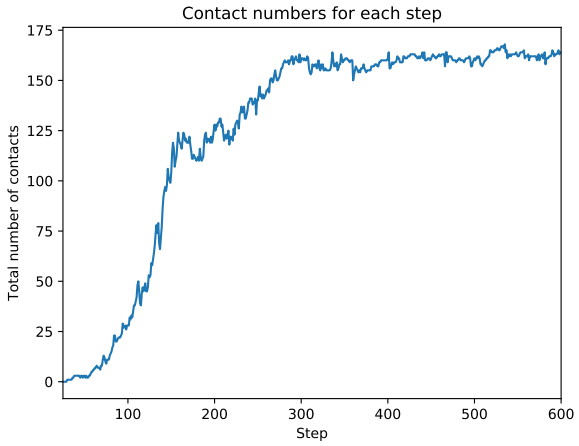


(a) Time Step=0 (b) Time Step=200 (c) Time Step=400

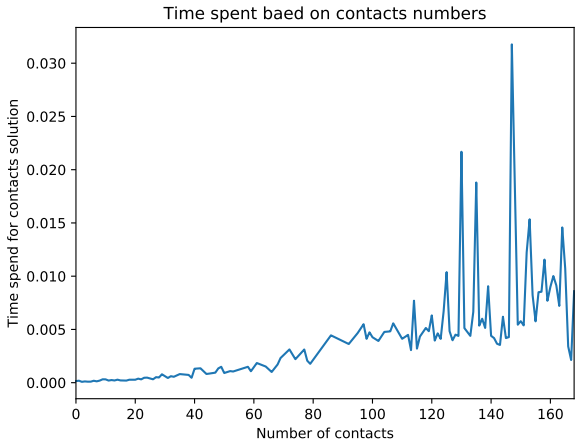
Figure 5.1: Visualization for experiment simulation

5.1.3 SPH parameters

In Section 3.4.2, it has been tested that **SPH** is a good strategy to generate grid images representing a discrete snapshot of the dynamics. Then, the grid size and smoothing length h are imported into our data generation. The grid size should be reasonable so that there is no two contact points are mapped into the same cell, since if there are more than two contact values in one cell, the rid node might store mixture information state from two objects. which would be hard for CNN to find the relationship within one objects, which will decrease the accuracy of prediction. This was mentioned in section 3.2.3.



(a) The number of contacts in each step.



(b) Time spent for contact solutions.

Figure 5.2: Visualization for experiment simulation

Grid Size and Kernel Length

I define $\boldsymbol{d} = (d_x, d_y)$ as grid cell size and h as smoothing length. I conclude some rules for determining grid cell size and smoothing length.

- Since the objects are circles, the ideal cell size should like,

$$d_x = d_y = d$$

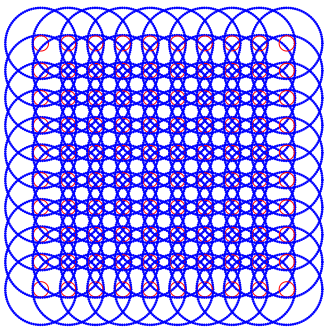


Figure 5.3: Example visiualization for Smoothed Particle Hydrodynamics. The small red circles stand for the grid nodes, and the blue circles stand for kernel size.

- Since there can not be two contact points are mapped into the same cell, d must be less than the distance of nearest two contact points. It can be defined.

$$d \leq r = 1$$

- Similarly, if one contact point can only be mapped to one cell, the smooth length h should be less than the minimum distance between two contact points d,

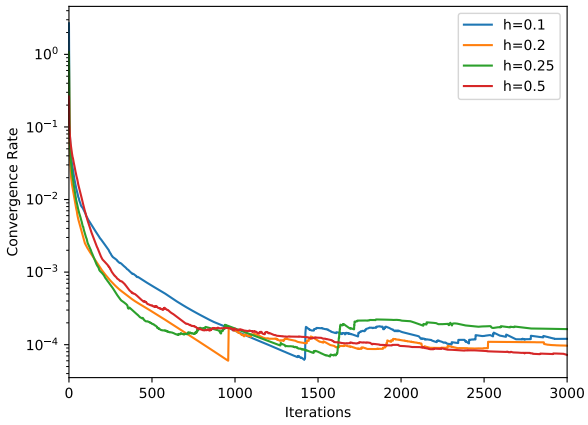
$$h \leq d$$

- For a given d, whatever the contact position $\boldsymbol{q} = (q_x, q_y)$ is, its information can be restore in nearby nodes. So it can be,

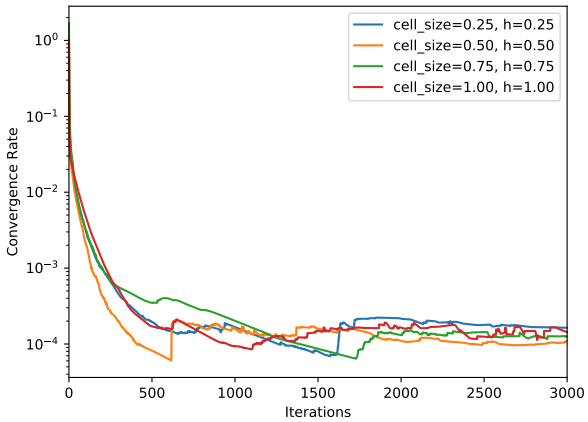
$$h \geq \frac{\sqrt{2}}{2} d \approx 0.71d$$

One experiment has been designed to test whether the rules can be applied in this case. Still using the simulation config introduced in section 5.1.1. The cell size is fixed, $d = 0.25$. Different smoothing length is appiled. The results are shown in Figure 5.4a.

As what is shown in Figure 5.4a, when $h = 0.4d$ or $h = 2d$, the solver coverages more slowly and unstable. Overall, for the next



(a) The grid size d is set 0.25. $h = 0.1, 0.2, 0.25, 0.5$ is tested respectively. This figure shows different coveragence rate based on different h value.



(b) Coveragence rate for different d value.

Figure 5.4: Experiments from kernel **Poly6**

step, data generation, I will make $h = d$ always. The next step is to explore what d value will be good. Once $h = d$ has been determined, the next step is to choose the value of d. Another experiment is taken to check the convergence rate with using diffrent d values. The reults are showns in Figure 5.4b. From this figure, it is obviously, when $d = 0.5$, iterative solver converages the most

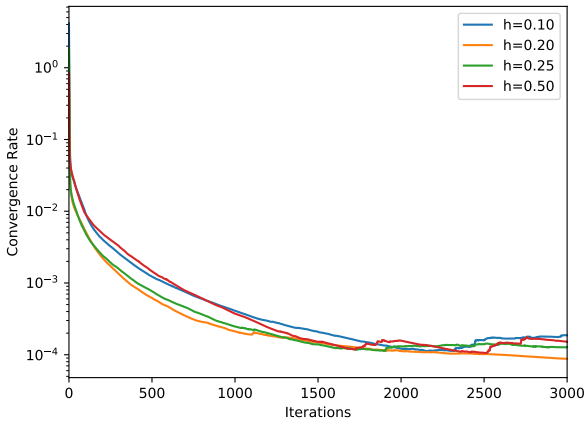
rapidly. **Importantly**, $d = 0.5$ and $h = 0.5$ is not always the best experimented with different numbers of rigid inside the world box. But, it generally performs better than other values. As a result, I will choose $h = d = 0.5$ as the parameters for SPH method.

$$d_x = d_y = 0.5 \text{ and } h = 0.5$$

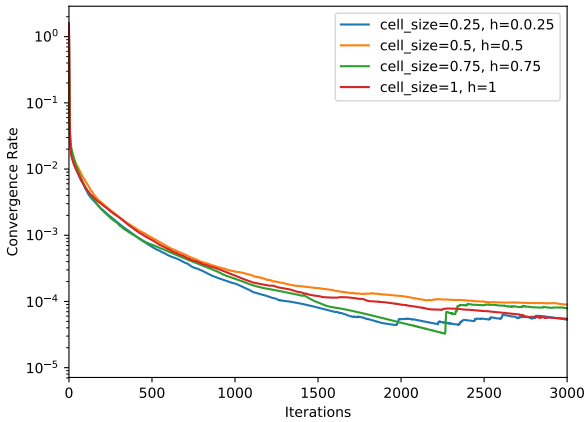
Kernel Choosing

The choice of kernel also needs to be addressed. In the paper, two kernels, **Poly6** and **Spiky**, have been introduced in section 3.2.2. In order to compare these two kernels and determine the best one, finding the best parameters for **Spiky** is essential. I did the same experiment with kernel **Spiky**. Find a good scale between h and d firstly and then choose d. The results are shown in Figure 5.5. The plot in Figure 5.5a reveals that $h = d$ is also a good choice for kernel **Spiky**. Although ds perform similarly, $d = 0.25$ performs more stable and a bit fater in convergence. These two kernels are separately used based on Algorithm 5. In order to compare these two kernels carefully, the fixed iteration number was extended to 5000. Then, the plot about the convergence rate of different kernels is obtained in Figure 5.6. From the plot, it is clear that kernel **Poly6** perform better. As a conclusion, SPH with **Poly6** kernel($\boldsymbol{d} = (0.5, 0.5)$, smoothing length $h = 0.5$), will be used for particle-grid transformation.

After determining grid size and smoothing length, I will compare this specific **SPH-Model** with other models mentionde in section 3.4.2, following Algorithm 5. The result is shown in Figure 5.7. Unlike the result shown in Figure 3.8. Compared with other models, **SPH-Model** gets convergence much faster. So, $h = d = 0.5$ is a good choice for this project and the future steps.



(a) The grid size d is set 0.25. $h = 0.1, 0.2, 0.25, 0.5$ is tested respectively. This figure shows different coveragence rate based on different h value.



(b) Coveragence rate for different d value.

Figure 5.5: Experiments for kernel **Spiky**.

5.2 Data Generation

To create data that are more accessible to learning, I will map a discrete element method into a continuum setting use techniques from smooth particle hydrodynamics(SPH). In order to get enough available training data and remove useless information, I used,

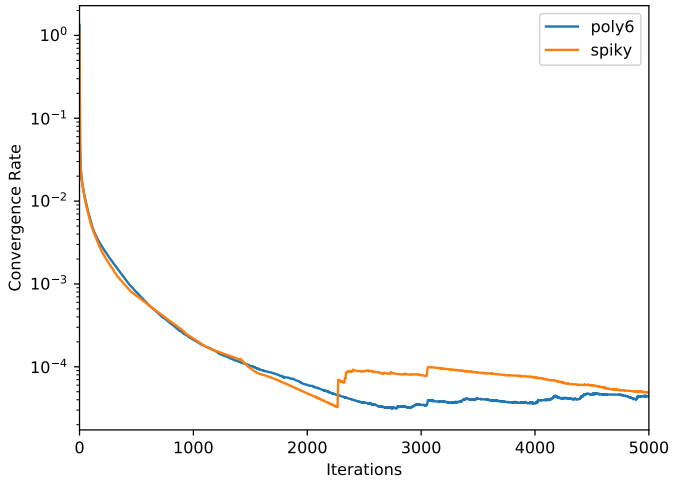


Figure 5.6: Coveragence rate for kernel **Poly6** anf **Spiky**. $h_{\text{poly6}} = d_{\text{poly6}} = 0.5$, while $h_{\text{spiky}} = d_{\text{spiky}} = 0.25$

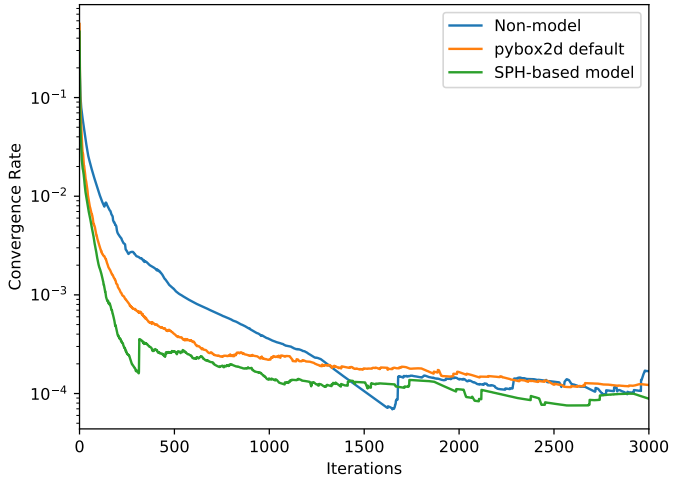


Figure 5.7: Coveragence rate for models(different initial values for $\boldsymbol{\lambda}$).

- Totally 150 simulations with different initial configuration. The number of circle objects is not fixed as well. However, if there are too few objects inside the box, contacts might not happen

until the simulation ends up. So the number of objects will be in the range of $[70, 110]$.

- If there is no contact in one state, the state means nothing to the learning. So the state files without any contacts will be removed.

5.2.1 XML Restoration

For each simulation, the state in every time step will be stored in one **XML** file. The structure of the body is consist of **mass**, **position**, **velocity**, **spin omega** and **inertia**. The structure of contact consists of **postion**, **impulse**(including in normal direction and tangent direcrion), **master** body and **slave** body. Two examples are given in the following.

This is an example of one body information is given below.

```
<body index="86" type="free">
    <mass value="3.14159274101"/>
    <position x="7.79289388657" y="2.62924313545"/>
    <velocity x="2.7878344059" y="-1.45545887947"/>
    <orientation theta="-0.115291565657"/>
    <inertia value="1.57079637051"/>
    <spin omega="-2.33787894249"/>
    <shape value="circle"/>
</body>
...
```

Another example *XML* code is for contatc force.

```
<contact index="1" master="2" master_shape="
    b2CircleShape(childCount=1, pos=b2Vec2(0,0), radius
    =1.2000000476837158, type=0,)" slave="97" slave_shape
    ="b2CircleShape(childCount=1, pos=b2Vec2(0,0), radius
    =1.2000000476837158, type=0, )">
    <position x="0.21963849663734436" y="
        13.875240325927734"/>
```

```
    <normal normal="b2Vec2(-1,2.9819e-05)"/>
    <impulse n="0.005236322991549969" t="
        -0.002184529323130846"/>
</contact>
...
```

5.2.2 SPH configuration

As it is talked in 5.1.3,

- **Kernel**, Poly6, which you can see in section 5.4.
- **Kernel Settings**, cell size $d_x = d_y = 0.5$, smoothing length $h = 0.5$.

5.2.3 XML to grid

After getting a set of **XML** file, the next step is to load state information from **XML** file, and then map them to grid images with SPH based method. Since for any contact point, there are two normal contacts, one with $[n_x, n_y]$ while the other with $[-n_x, -n_y]$. So the grid values for $\boldsymbol{n}$ would be always $\mathbf{0}$, and the same with $\boldsymbol{t}$. So finally, the channel number of each image will be 6, including $[m, v_x, v_y, \omega, \lambda_n, \lambda_t]$.

For the learning, I divide 8 channels into features(input) and label(output).

$$\text{Feature} = [m, v_x, v_y, \omega]$$

$$\text{Label} = [\lambda_n, \lambda_t]$$

5.3 CNN Training

5.3.1 CNN Architecture

The neural network was designed using Keras[30]. Keras is a neural networks Application Programming Interface (API) written in

Python, it runs on top of either TensorFlow. With some inspiration from AlexNet[31], the networks shows five constructing stacks of layers and each stack is consist of two or three convolutional layers in the same size. To avoid overfitting, each stack is followed by one dropout layer. One full-connected layer is set after the last convolutional layer. This input layer is followed by a batch normalization layer, normalizing images within a batch, which is discussed in Section 4.2.6. This architecture includes $51,599,092$ weights for an input size of $(61 \times 61 \times 4)$. All convolutional layers have kernels of size $(3 \times 3 \times d)$, and are followed by ReLU activation function. Figure 5.8 shows the visualization of model architecture, and Table 5.1 shows the specific information about parameters when input images go through the CNN.

- **Input Size**, the input will be $61 \times 61 \times 4$. Since the orginal world is 30×30 and grid size is $d = 0.5$, the generated grid would be 61×61. There would 4 channels
- **Output Size**, output size depends on the label size. The orginal label image would be $[\lambda_n, \lambda_t]$, so the label image size would be $61 \times 61 \times 2$, which should be flattened as the actual training label. The label size would be $61 \times 61 \times 2$
- **Weights Number**, the total weights number is $64,498,866$.

Then we can define *cnn* as the overall function of CNN model, where $\boldsymbol{G}_\lambda$ is the image of contact forces(label) and $\boldsymbol{G}_\mathcal{B}$ is the image of bodies state(feature).

$$\boldsymbol{G}_\lambda = cnn(\boldsymbol{G}_\mathcal{B}) \tag{5.1}$$

5.3.2 Traing Configuration

Loss Function

Firstly, we define a filter funtion,

$$g(x) = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases} \tag{5.2}$$

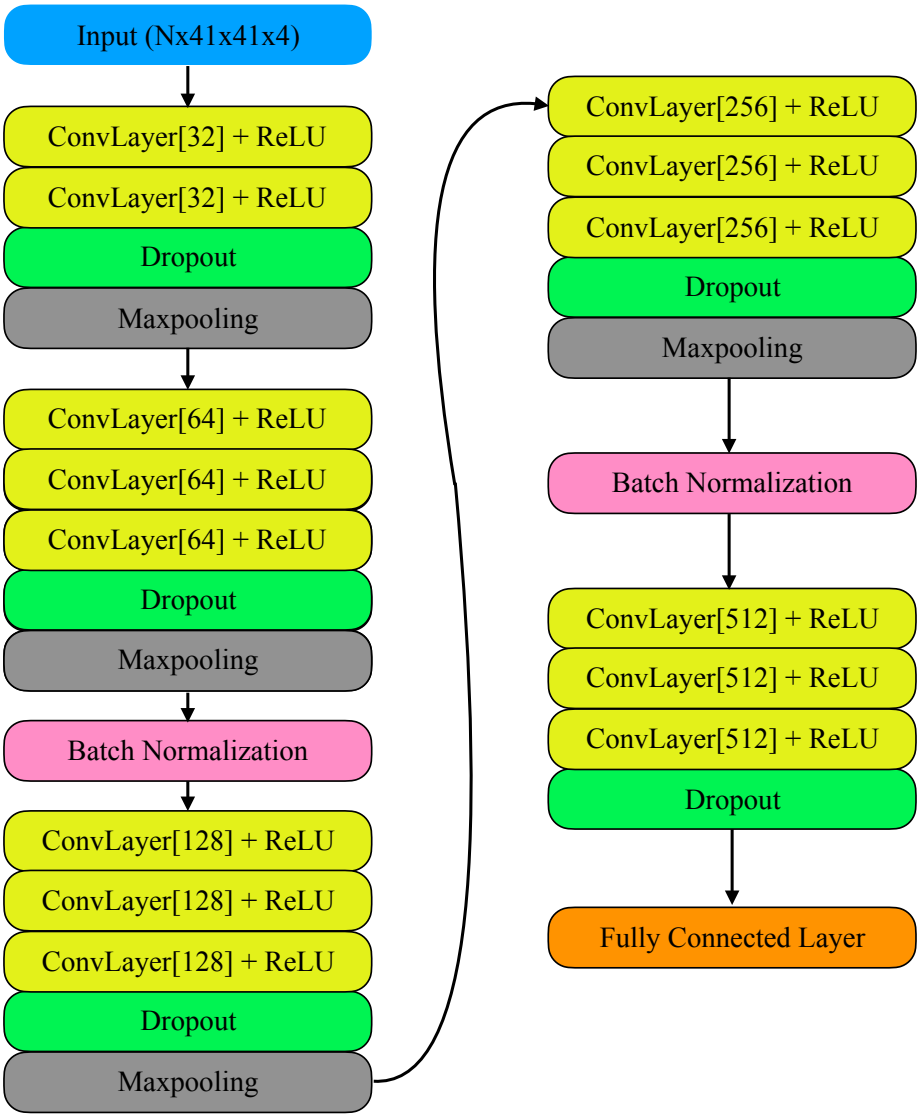


Figure 5.8: Architecture of CNN model

Then, we can update the loss function based on Euqation 4.5.

$$L = \frac{1}{N}\sum_{i}^{N} g(\hat{y}_i)(y_i - \hat{y}_i)^2 \tag{5.3}$$

Layer	Output Shape
Input	$n_b \times 61 \times 61 \times 4$
Convolution(32)	$n_b \times 61 \times 61 \times 32$
Convolution(32)	$n_b \times 61 \times 61 \times 32$
Dropout	$n_b \times 61 \times 61 \times 32$
MaxPooling	$n_b \times 31 \times 31 \times 32$
Convolution(64)	$n_b \times 31 \times 31 \times 64$
Convolution(64)	$n_b \times 31 \times 31 \times 64$
Convolution(64)	$n_b \times 31 \times 31 \times 64$
Dropout	$n_b \times 31 \times 31 \times 64$
MaxPooling	$n_b \times 16 \times 16 \times 64$
BatchNormalization	$n_b \times 16 \times 16 \times 64$
Convolution(128)	$n_b \times 16 \times 16 \times 128$
Convolution(128)	$n_b \times 16 \times 16 \times 128$
Convolution(128)	$n_b \times 16 \times 16 \times 128$
Dropout	$n_b \times 16 \times 16 \times 128$
MaxPooling	$n_b \times 8 \times 8 \times 128$
Convolution(256)	$n_b \times 8 \times 8 \times 256$
Convolution(256)	$n_b \times 8 \times 8 \times 256$
Convolution(256)	$n_b \times 8 \times 8 \times 256$
Dropout	$n_b \times 8 \times 8 \times 256$
MaxPooling	$n_b \times 4 \times 4 \times 256$
BatchNormalization	$n_b \times 4 \times 4 \times 512$
Convolution(512)	$n_b \times 4 \times 4 \times 512$
Convolution(512)	$n_b \times 4 \times 4 \times 512$
Convolution(512)	$n_b \times 4 \times 4 \times 512$
Dropout	$n_b \times 4 \times 4 \times 512$
Flatten	$n_b \times 8192$
Dense	$n_b \times 7442$

Table 5.1: Feature map (tensor) sizes through the network, the input has size $n_b \times 61 \times 61 \times 5$, with batch size n_b and patches of size $61 \times 61 \times 5$.

5.3.3 Training Details

The learning happened on GPU(*GeForce GTX 1080 Ti, 11 Gbps GDDR5X memory*)[3] held by Image Section, DIKU[4]. The whole learning takes nearly 24 hours. The model you can download in my personal dropbox[5]. I gave hyperparameters setting below,

Hyperparameter	Setting
Activation function	ReLU
Weight initilization	He normal[32]
Weight regularizer	L2 [16]
Convolution border mode	Same
Stride	2
Kernel size	$(3, 3)$
Dropout rate	0.1
Optimizer	SGD
Initial Learning Rate	$1 \times 5 \times 10^{-3}$
Batch Size	200
Epoch	1000
Validation Rate	0.2

Table 5.2: Hyperparameter settings.

Learning Rate

The learning rate will change with the number of epoch, as talked in section 4.3.4. I will give a specific value as the learning rate depending on the number of epoch. The learning rate will become smaller with increasing epoch. data. The overall algorithm is concluded in Algorithm 6.

[3] `https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/`

[4] `https://di.ku.dk/english/research/imagesection/`

[5] `https://www.dropbox.com/s/jrwzqib6ghrq59i/model.h5?dl=0`

Conclusion about Training process

- The size of the input image is $41 \times 41 \times 4$, and all pixels are important since all of them retore useful information. If we want to extract the features from 41×41, 3×3 kernel would be a good choice. But the disadvantages is that there would be a lot of training parameters, which might slow down the learning process.

- I do have nearly 18208 samples for training. However, it is hard for both the GPU and CPU to load all the data at one time. Dividing all training samples to a couple of batches would not only speed up the single training time and release memory, but also reduce the risk of overfitting.

- The learning rate will change with the number of the epoch, as analyzed in section 4.3.4. I will give a specific learning rate values depending on the number of the epoch. The learning rate will become smaller with With the epoch increase, I hope the value of learning rate can decrease. Since the high learning rate would cause suboptimal performance at the end of training. The overall algorithm is concluded in Algorithm 6.

- Validation data set is essential to test the accuracy of the model and whether it is overfitting. Before each epoch, validation would be randomly selected, 20% of total data.

5.4 Simualtion based on Trained Model

Once getting the trained model, the next step is to apply this model in simulation based on Algorithm 7 and compare it with other solutions. The details of process is described in Algorithm 7

I applied it in one test world built by the setting mentioned in section 5.1.1. The test world is consist one 30×30 box and $K(50 < K < 100)$ randomly distributing balls$(r = 1)$. The balls will

Data: *epoch*
Result: learning rate η
if $epoch < 100$ **then**
  $\eta = 5 \times 10{-}3$
end
if $100 < epoch < 300$ **then**
  $\eta = 2 \times 10^{-3}$
end
if $300 < epoch < 500$ **then**
  $\eta = 1 \times 10^{-3}$
end
if $epoch > 300$ **then**
  $\eta = 2 \times 10^{-4}$
end

Algorithm 6: Learning Rate Scheduling

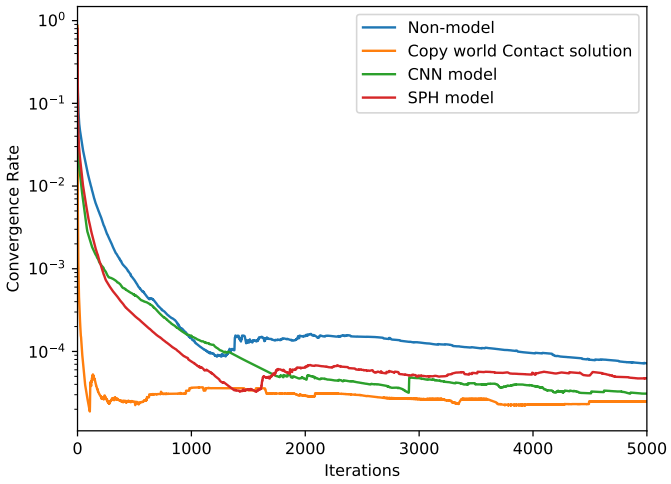


Figure 5.9: The final result. Add the final CNN solution to compare with other methods.

fall down due to gravity. Figure 5.9 shows the details. As what is expected, CNN model performs similar with **SPH-Model**(mentioned

in section 3.4.2). Although the model is not as good as **SPH-model**, its coverages rate is obviously faster than built-in ones.

From Figure 5.9, it can be indicated that the CNN model actually makes the iterative solver converge faster. But it is just a small improvement to buit-in warm starting. The CNN solution cannot reach a convergence as rapidly as **Copy-Model**. It is mainly the limitation of the SPH-based method.

In the next step, I want to review whether CNN model can speed up the simulation. I change the fixed iteration steps to a thold

Data: Given a set of bodies $\mathcal{B}_t \;\; t \in T$ in time t, a trained CNN model CNN, spatial positions of all grid nodes and one .

Result: Get the contact forces $\boldsymbol{\lambda}_t \;\; t \in T$ at time t.

for *all* t *in* T **do**

1. Read $\mathcal{B}_t$.
2. map the bodies state to a gird image,

$$\boldsymbol{G}_{\mathcal{B}} \leftarrow Bodies2grid(\mathbf{x}, \mathcal{B}_t)$$

3. Transform bodies imgae $\boldsymbol{G}_{\mathcal{B}}$ to a contact grid image $\boldsymbol{G}_{\lambda}$,

$$\boldsymbol{G}_{\lambda} \leftarrow cnn(\boldsymbol{G}_{\mathcal{B}})$$

4. Once the contact force image $\boldsymbol{G}_{\lambda}$ is obtained, then

for *all* $j \in \mathcal{C}$ **do**

$$\boldsymbol{\lambda}_j \leftarrow interpolate(\boldsymbol{G}_{\lambda}, \mathbf{x}, \boldsymbol{q}_j)$$

end

4. Using $\boldsymbol{\lambda}$ as a warm starting to the iterative solver to get the final solution,

$$\boldsymbol{\lambda} \leftarrow Solver(\boldsymbol{\lambda})$$

5. Updating the simulation world(velocity and position) by $\boldsymbol{\lambda}$ and $t = t + \Delta t$

end

Algorithm 7: Algorithm describing how CNN-Model works in the dynamic simulation.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Half one year ago, I had to formulate a set of goals for the work of this thesis. The goals were thus set before having gained the knowledge I have now. For this reason, the goals reached may appear slightly different from those set. I started out thinking that it would be better to train learning models for normal and friction forces respectively. After four-moonths considering, I have to face the problem that normal force prediction would use the same grid images with friction force. Besdies, the output size of these models would be the same as well. So, training two models equals to training one model with more convolutional layer and extended fully connected layer.

- **Describe the contact model by *Newton-Euler* equation**, this is described in Chapter 2. Normally, the contact model can be derived as a linear complementarity problem formulation(Equation 2.27), which can solved by number.
- **Analyze SPH kernel**, I introduced two smoothing kernel in this project, **Poly6** and **Spiky**. Intially, I tried to choose one kernel firstly, and then determine parameters(grid size and

smoothing length). However, I found it would be smart to find good grid size(d) and smoothing length(h) respectively, then compare the performance of two kernels with good parameters.

- **Analyze grid size and smoothing length**, the analysis about grid cell size and smoothing length can be found in Figure 5.4 and Figure 5.5.
- **CNN design**, the CNN architecture can be reviewed in Figure 5.8. Since there are no standard for CNN structure designing. I got the inspiration from AlexNet.
- **Comparision among different model solution**, the results describing the difference among **Copy-Model**, **SPH-Model**, **None-Model** and **CNN-Model**, were shown in Figure 3.7, Figure 3.8 and Figure 5.9.
- **Training both normal forces and friction forces**, depending on the feature of CNN, training models for normal and friction forces respectively, equals to training one model with more convolutional filters and layers, which can predict both of normal and friction forces.

6.2 Future Work

6.2.1 SPH-based method

- For the SPH-based method, it is still far away from perfect. It performs not much better than warm starting. So in the future, exploring deeper in SPH-based will help to improve the whole process. The probable attempt will including trying more new kernels, which could stay
- The interpolation used in this project is just one simple linear interpolation. I thought it would lose some essential information when it was interpolated back to particles. So exploring another interpolation method would be helpful to this project.

- If we want to use the CNN model to replace built-in warm starting, the biggest issue is simulation time. Although the CNN model can reach convergence more quickly than built-in warm starting method, it cannot replace the built-in method due to long-time spending. It takes a long time to do interpolation. So an efficient and accurate method or data structure for interpolation would speed up the whole process.

6.2.2 CNN architeture

Since the peformance of **CNN-Model** is close to **SPH-Model**, I think this CNN architecture is a good design and predicts the grid values well. However, in the future, more comlicated CNN structure should be tried on improving the accuracy of prediction.

6.2.3 More experiments

Since current experiments and attempts are all based on same-size circle objects, I hope more experiments can be carried out on circle objects in different size or different shapes(eg. trangle, rectangle).

References

[1] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating Eulerian Fluid Simulation With Convolutional Networks", *ArXiv e-prints*, Jul. 2016. arXiv: 1607.03597 [cs.CV].

[2] K. Erleben, "Velocity-based shock propagation for multibody dynamics animation", *ACM Trans. Graph.*, vol. 26, no. 2, Jun. 2007, ISSN: 0730-0301. DOI: 10.1145/1243980.1243986. [Online]. Available: http://doi.acm.org/10.1145/1243980.1243986.

[3] R. Boulic and A. Luciani, *Collision detection algorithm*, 2007.

[4] J. Bender, K. Erleben, and J. Trinkle, "Interactive simulation of rigid body dynamics in computer graphics", in *Computer Graphics Forum*, Wiley Online Library, vol. 33, 2014, pp. 246–270.

[5] M. Anitescu, "Modeling rigid multi body dynamics with contact and friction", PhD thesis, University of Iowa, 1997.

[6] M. Poulsen, S. Niebe, and K. Erleben, "Heuristic convergence rate improvements of the projected gauss–seidel method for frictional contact problems", 2010.

[7] N. Watters, A. Tacchetti, T. Weber, R. Pascanu, P. Battaglia, and D. Zoran, "Visual interaction networks", *CoRR*, vol. abs/1706.01433, 2017. arXiv: 1706.01433. [Online]. Available: http://arxiv.org/abs/1706.01433.

[8] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, and G. Wang, "Recent advances in convolutional neural networks", *CoRR*, vol. abs/1512.07108, 2015. arXiv: `1512.07108`. [Online]. Available: `http://arxiv.org/abs/1512.07108`.

[9] J. J. Monaghan, "Smoothed particle hydrodynamics", *Annual review of astronomy and astrophysics*, vol. 30, no. 1, pp. 543–574, 1992.

[10] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications", in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Eurographics Association, 2003, pp. 154–159.

[11] M. Desbrun and M.-P. Gascuel, "Smoothed particles: A new paradigm for animating highly deformable bodies", in *Computer Animation and Simulation'96*, Springer, 1996, pp. 61–76.

[12] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural networks*, vol. 61, pp. 85–117, 2015.

[14] F. Rosenblatt, "A probabilistic model for information storage and organization in the brain1", *Artificial Intelligence: Critical Concepts*, vol. 2, no. 6, pp. 386–408, 2000.

[15] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series", *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[16] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets", *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[17] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks", in *Advances in neural information processing systems*, 2007, pp. 153–160.

[18] F. J. Huang, Y.-L. Boureau, Y. LeCun, *et al.*, "Unsupervised learning of invariant feature hierarchies with applications to object recognition", in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, IEEE, 2007, pp. 1–8.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, p. 436, 2015.

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

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition", in *Artificial Neural Networks–ICANN 2010*, Springer, 2010, pp. 92–101.

[23] F.-F. Li and A. Karpathy, *Convolutional neural networks for visual recognition*, 2015.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *arXiv preprint arXiv:1502.03167*, 2015.

[25] Y.-l. Boureau, Y. L. Cun, *et al.*, "Sparse feature learning for deep belief networks", in *Advances in neural information processing systems*, 2008, pp. 1185–1192.

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[27] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd", in *Advances in Neural Information Processing Systems*, 2015, pp. 685–693.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *CoRR*, vol. abs/1412.6980, 2014. arXiv: `1412.6980`. [Online]. Available: `http://arxiv.org/abs/1412.6980`.

[29] P. J. Werbos, "Backpropagation through time: What it does and how to do it", *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[30] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2999134.2999257`.

[32] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning", in *International conference on machine learning*, 2013, pp. 1139–1147.

Appendices

Appendix A.

Smoothed Particle Hydrodynamics

A..1 Smoothing Kernels

```
def W_poly6_2D(r, h):
    '''
    :param r: A 2d matrix where each column is a vector
    :param h: Support radius
    :return:
    '''
    assert r.shape[0] ==2

    # We determine the length of the vectors and pick out those
                                        with length below h
    r_norm =np.linalg.norm(r, axis=0)
    W = np.zeros(r_norm.shape)
    W_i =np.where(r_norm <h)

    # We determine the weights
    c = 4 / (np.pi *np.power(h, 8))
    h2 =np.power(h, 2)
    W[W_i] =c * np.power(h2 -np.power(r_norm[W_i], 2), 3)

    # We normalize the weights so that they add up to 1
    W_sum =np.sum(W)
    W = W / W_sum
```

```
    return W

def spiky_2D(r, h):
    assert r.shape[0] ==2

    r_norm =np.linalg.norm(r, axis=0)
    W = np.zeros(r_norm.shape)
    W_i =np.where(r_norm <h)

    c = 15 / (np.pi *np.power(h, 6))
    h2 = np.power(h, 2)
    W[W_i] =c * np.power(h -np.abs(r_norm[W_i]), 3)

    # We normalize the weights so that they add up to 1
    W_sum =np.sum(W)
    W = W / W_sum

    return W
```

Appendix B.

Deep Learning

B..1 Loss Function

```
def loss_func(y_true, y_pred):
    # remove 0 in y_true
    flags =tf.to_float(K.not_equal(y_true, 0.0))
    return K.mean(K.square((y_pred -y_true) *flags), axis=-1)
```

B..2 CNN architecture

```
def build_model(self, input_shape, output_shape):
        self.model.add(
            Conv2D(
                32,
                (3, 3),
                padding='same',
                kernel_regularizer=keras.regularizers.l2(self.
                                                    weight_decay),
                kernel_initializer='he_normal',
                input_shape=input_shape))
        self.model.add(Activation('relu'))

        self.model.add(
            Conv2D(
                32,
```

```
        (3, 3),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal',
        input_shape=input_shape))
self.model.add(Activation('relu'))

self.model.add(MaxPooling2D(pool_size=(2, 2),
                            strides=(2, 2),
                            padding='same'))

self.model.add(
    Conv2D(
        64,
        (3, 3),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        64,
        (3, 3),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        64,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))
```

```
self.model.add(MaxPooling2D(pool_size=(2, 2),
                            strides=(2, 2),
                            padding='same'))

self.model.add(BatchNormalization())
self.model.add(
    Conv2D(
        128,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                                 weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        128,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                                 weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        128,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                                 weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(MaxPooling2D(pool_size=(2, 2),
                            strides=(2, 2),
                            padding='same'))

self.model.add(
```

```
    Conv2D(
        256,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        256,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(
    Conv2D(
        256,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
        kernel_initializer='he_normal'))
self.model.add(Activation('relu'))

self.model.add(MaxPooling2D(pool_size=(2, 2),
                            strides=(2, 2),
                            padding='same'))

self.model.add(BatchNormalization())
self.model.add(
    Conv2D(
        512,
        (2, 2),
        padding='same',
        kernel_regularizer=keras.regularizers.l2(self.
                                            weight_decay),
```

```
                kernel_initializer='he_normal'))
        self.model.add(Activation('relu'))
        self.model.add(
            Conv2D(
                512,
                (2, 2),
                padding='same',
                kernel_regularizer=keras.regularizers.l2(self.
                                                         weight_decay),
                kernel_initializer='he_normal'))
        self.model.add(Activation('relu'))
        self.model.add(
            Conv2D(
                512,
                (2, 2),
                padding='same',
                kernel_regularizer=keras.regularizers.l2(self.
                                                         weight_decay),
                kernel_initializer='he_normal'))
        self.model.add(Activation('relu'))

        self.model.add(Dropout(self.dropout))
        self.model.add(Flatten())

        output_size =output_shape[0]

        self.model.add(Dense(output_size))
        self.model.add(Activation('relu'))
```

B..3 Learning Rate Scheduler

```
def scheduler(epoch):
    if epoch <=100:
        return 0.05
    if epoch <=400:
        return 0.02
    return 0.01
```