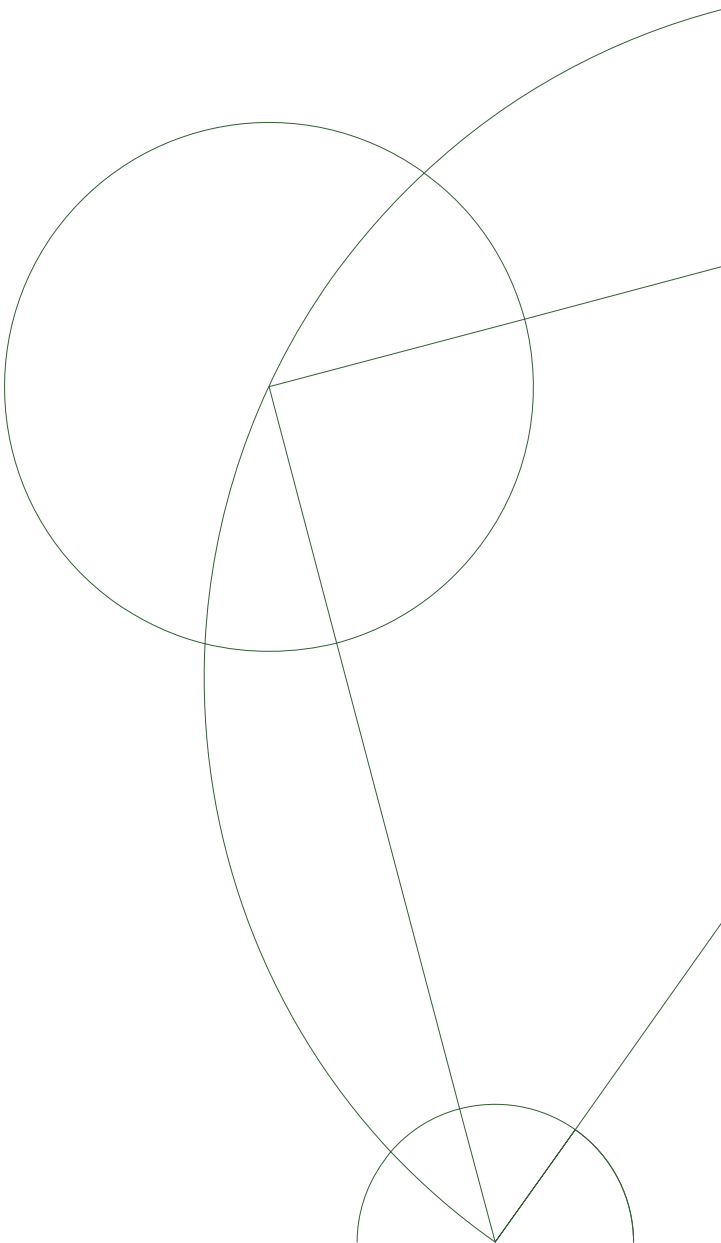
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Jian Wu – xcb479@alumni.ku.dk

Deep Contact

Accelerating Rigid Simulation with Convolutional Networks



Supervisor: Kenny Erleben

August 6th 2018

Abstract

This is a master theis for Jian Wu(KUID:xcb479), from IT and Cognition program.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introdustion

1.1 Motivation

Multibody dynamics is the art of simulating the physical interactions of rigid bodies in a virtual world. Depending on the application, the importance of physical correctness may vary greatly. For engineering purposes, the level of physical correctness will take top priority, which is often at the expense of interactivity. On the other hand, when used in a computer game, the physical simulations only need to be plausible whereas interactivity is of utmost importance.

Using the terminology physical correctness is problematic for many rea- sons. Each step from real world observation to the final simulation consists of idealizations and discretizations. The mathematical models are idealized descriptions of empirical observations, the numerical model is a discretiza- tion of the mathematical model and so forth. In this context, the rigid body assumption is just another approximation. Still, there exist cases where the rigid body assumption is a fair approach, examples include assembly line simulation [FAS] and robot simulation [Gaz]. Instead of physical correctness, I shall use the terminology plausibility.

A multibody dynamics simulator is a complex system of many parts, the focus of this thesis is on the specific problem of determining contact forces. When two rigid bodies are in contact, the resulting interactions can be described as a set of contact forces or impulses acting on the two bodies. Contact forces ensure that my tea mug stays firmly on the table, rather than sinking into it or slid-

ing along it. When a brick is thrown at a wall,

Movies studies have been pushing facial and character animation to a level where machine learning can automize a large part of this work in a production pipeline. Research community is exploring machine learning for gait control too and quite successfully or for upscaling 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 have been done in terms of inverse simulations or pilings to control rigid bodies to perform a given artistic 'target'. These thechniques are more in spirit of inverse problems that maps initial conditions to a well defined outcome(number of bounces or which face up on a cude) or level of detail idea replacing interiors of piles with stracks of cylinders pf decreasing radius to make an overall apparent pile have a given angle of of repose.

Contact forces determine what happens to both brick and wall. The accuracy of the computed contact forces affects both plausibility and stability of the final simulation. For this reason, good contact force determination methods are wanted in physical simulation, whether it is used in engineering systems or computer games.

1.2 Goals

Taking a master's degree is a specialization process, a fine tuning of skills. Early on, I had an idea that my specialization would be computer vision and deep learning. Deep learning methods have got great successes in mnay fields, like computer vision, natual language processing. However, they s

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.

1.3 Overview and Outline

1.3.1 Overview

This master project aims to apply deep leaning in rigid simulation, specificly improving the speed of computing contact forces. Basicly, now researchers mainly use iterative solver to compute contact forces for each time step. Our idea is to use deep learning to give a available initial value which is close to final solution. This project can be devided to two parts. The first part is to find a method to generate assessible data for deep learning then test it whether it is available. The second part is to train one learning model and apply it to check its performance.

The paper mainly covers topics from computer simulation and deep learning. Try to improve the simulation of rigid dynamics. I gave an oueline as following,

- **Chapter 1**, is the introduction to the whole paper, including motuvation, learning goals, work process 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 import part to analize the main interpolation method used for the project, Smoothed Particle Hydrodynamics(SPH). At the end of the chapter, some experiments are taken to analize whether SPH is good to this case.

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

- **Chapter 5**, does describe implementation details specificly, from data generation to model design and training. Analysis about SPH kernel determine, grid size and smoothing length choice, model training process and comparation between learning model and built-in algorithm in simulation software can be found in this chapter.

- **Chapter 6**, is the end of this paper. In the chapter, some conclusions will be made based on experiments implementation. Finally, I will make a future work, which can be a improvement for current project.

Chapter 2

Contact Models

In an attempt to apply deep learning in a rigid dynamic simulation process, the first step is to understand the details associated with rigid dynamic simulation. 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 numercial solution for dynamic system equation.

2.1 Overview

In rigid body simulation, contact force is used to prevent rigid body from penetrating each other. The accuracy of calculated contact force has great influence on the fidelity of simulation. In order to achieve physical rationality, frictional forces are essential. The formulae of friction contact force problems include the modeling of normal force constraint and friction force constraint. In the future, friction contact force problem will be simply called contact force problem. 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 acedamic 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 r 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. We 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 rotatiopn 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 partcle 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 particle, a rigid body occupies a volume of space and has a particular shape. Rigid bodies are more complicated, beside 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 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 is mainly consist by its position and orientation (describing spatial information), and its linear and angualr 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:

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

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

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

2.2.5 Twist/Wrench

We will now introduce vectors called twists, which describe velocities, and wrenches, which 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

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

If we define $\boldsymbol{M}$ and $\mathbf{F}$ as Equation 2.15 and 2.16

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & m\mathbf{1}_{d\times d} \end{bmatrix} \tag{2.15}$$

$$\mathbf{F} = \begin{bmatrix} \boldsymbol{\tau} - \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} \\ \boldsymbol{f} \end{bmatrix} \tag{2.16}$$

So, we can rewrite Newton-Euler equation as,

$$\boldsymbol{M}\dot{\mathbf{v}} = \mathbf{F} \tag{2.17}$$

2.3 VELOCITY-BASED COMPLEMENTARITY FORMULATION

After get some basic physical and mathematical knowledge about dynamic system. Then, I will describe the equation of motion for system with contact forces. When objects collide, their interactions are modeled using contact forces. The contact forces consist of both normal forces and frictional forces, and are subject to a set of constraints. Without specifying the individual constraints, I define the set of constraint functions,

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

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

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

The constraints are added to Equation 2.17 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 Modeling contact

The frictional contact force problem can be stated as a linear complementarity problem (LCP) [19]. However, a slightly different formulation is used in interactive physical simulations, we will derive

this formulation. Without loss of generality, we will only consider a single contact point. The focus of this paper is on the contact force model, so the time stepping scheme and matrix layouts are based on the velocity-based formulation in [5]. Then we can rewirte the Newton–Euler equations based on Equation 2.20,

$$\boldsymbol{M}\dot{\mathbf{v}} = \boldsymbol{J}_n^T\lambda_n + \boldsymbol{J}_t^T\lambda_t + \mathbf{F}_{ext} \tag{2.21}$$

then we can get,

$$\dot{\mathbf{v}} = \boldsymbol{M}^{-1}\mathbf{F}_{ext} + \boldsymbol{M}^{-1}\boldsymbol{J}_n^T\lambda_n + \boldsymbol{M}^{-1}\boldsymbol{J}_t^T\lambda_t \tag{2.22}$$

Let the Lagrange multipliers $\boldsymbol{\lambda} = [\lambda_n\ \lambda_t]^T$ and contact Jacobian $\boldsymbol{J} = [\boldsymbol{J}_n, \boldsymbol{J}_t]$, then we can rewrite Equation 2.22 as,

$$\dot{\mathbf{v}} = \boldsymbol{M}^{-1}\boldsymbol{J}^T\boldsymbol{\lambda} + \boldsymbol{M}^{-1}\mathbf{F}_{ext} \tag{2.23}$$

The laws of physics must be combined into what we term an 'instantaneous-time' model, which describes the continuous-time motions of the rigid bodies. Following this, we discretize this model over time to obtain a 'discrete-time' model, which is a sequence of time-stepping subproblems. The subproblems are formulated and numerically solved at every time step to simulate the system.
It is well known that when friction is added to the acceleration-level dynamics equations (Equation 2.22).Anitescu and Potra [4] present a time-step method that combines the acceleration-level LCP with an intergration step for the velocities, arriving at a method having method having velocities and impulses as unknowns, rather tham acceleration and forces. the methud is guaranteed to have a solution, regardless of the number of contacts.
To discretize the system 2.22, the acceleration can be approximated by [4] as:

$$\dot{\mathbf{v}} \approx \frac{(\mathbf{v}_{t+1} - \mathbf{v}_t)}{\Delta t} \tag{2.24}$$

$\mathbf{v}_t$ 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,

$$\mathbf{v}^{t+1} = \mathbf{v}^t + \boldsymbol{M}^{-1}\boldsymbol{J}^T\Delta t\boldsymbol{\lambda} + \Delta t\boldsymbol{M}^{-1}\mathbf{F}_{ext} \tag{2.25}$$

Then we can define,

$$\begin{aligned}\mathbf{w} &= \boldsymbol{J}\mathbf{v} \\ &= \underbrace{\boldsymbol{J}\boldsymbol{M}^{-1}\boldsymbol{J}^T\Delta t}_{\boldsymbol{A}}\boldsymbol{\lambda} + \underbrace{\boldsymbol{J}(\Delta t\boldsymbol{M}^{-1}\mathbf{F}_{ext} + \mathbf{v}^t)}_{b} \\ &= \boldsymbol{A}\boldsymbol{\lambda} + \boldsymbol{b}\end{aligned} \tag{2.26}$$

2.4 The Numercial Solution Method

nce discrete-time models have been obtained, numerical methods must be applied to compute solutions. We start with how to integrate the motion of free moving rigid bodies such as bodies in ballistic motion without any collisions or contact. Subsequently, in Sections 4.1- 4.4 we cover numerical methods for computing solutions of the discrete LCP contact model from Section 3.1 and approaches for simulating articulated bodies. The methods are presented in a general setting; hence, A and b are arbitrary as defined in (23)

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

Most open-source software for interactive real-time rigid body simulation uses the Projected Gauss–Seidel (PGS) method for computing contact forces has been widely used by many open-source software and computer physics engine, 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.

Data: $N, \boldsymbol{\lambda}, \boldsymbol{A}, \boldsymbol{b}$
Result: Compute the values of $\boldsymbol{\lambda}$
for $k = 1$ ***To*** N **do**
  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
end

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

[1] `http://box2d.org/`
[2] `https://pybullet.org/wordpress/`

Chapter 3

Partcle-grid-particle

3.1 Overview

Deep learning methods have been widely applied in computer vision, like segmentation, objects recognition.Training data is always important to the whole learning process. Normally, researchers prefer using original image as the input images for training. However, 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 state data, like m, $\boldsymbol{v}$, $\boldsymbol{q}$, etc. In their new paper, they give different colors to diffent moving bodies, using a squence of frames as training data to make DNN get state information($\boldsymbol{v}, m, \boldsymbol{q}$) [5]. Although visual interact networks works good to predict physics motion. However, it does not work well for many bodies. In our view, we do not need CNN replace the contact solver entirely. We only hope deep neural network can accerlate the speed of simulation. In other words, we hope the learning model can give a reasonable values which are close to the correct ones. Afterward, the iteractive contact solver will converage rapidly.

Overall, the advanages of using grid-based method is,

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

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

2. Grid map image can restore assessible data(mass, linear velocity, angular velocity) for deep learning neural networks, while the visualization image of simulation can only shown the position of rigids.

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 Smotthed 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 convolutional neural network and intrdoced in Chapter 4), then one image(the number of channel is 2) will be got, which can be called *label image*.

3. Interpolated values generated by *label image* will be used as starting iterate values for contact force solver. In our hypothesis, the given strating values will speed up reaching convergence.

3.2 Grid-Based method

Traditional rigid motion simulation mainly use particle-based method. However, if we want to replace traditional contact solver with deep learning model, it is hard for cnn model to recognize the original image and do learning. Grid-based methos is a good to transfer original image to a grid-cells and then use

3.3 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) was invented to simulate nonaxisymmetric phenoma in astrophysis initially [6]. 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.

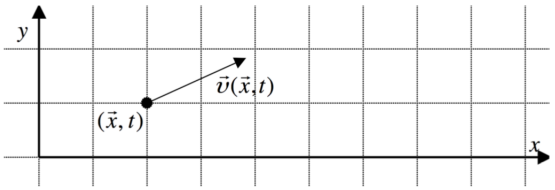


Figure 3.1: Grid description, *retrieved from MIT*(2011)

3.3.1 Fundamentals

At 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[7]. For ant field $A(\mathbf{r})$, a smoothed interpolated version $A_I(\mathbf{r})$ can be defined by a kernel $W(\mathbf{r}, h)$,

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

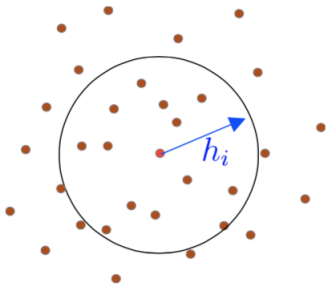


Figure 3.2: Visilaztion of SPH

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

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

Normally, we want the kenel to be Non-negative and rotational invariant.

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

$$W(\|\mathbf{r}-\mathbf{r}'\|,h) \geq 0 \tag{3.5}$$

For numerical work, we can use midpoint rule,

$$A_I(\mathbf{x}) \approx A_S(\mathbf{x}) = \sum_i A(\mathbf{x}_i)W(\|\mathbf{x}_i-\mathbf{x}\|,h)\Delta V_i \tag{3.6}$$

Since $V_i = m_i/\rho_i$

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

The default, gradient and Laplacian of A are:

$$\begin{aligned}\nabla A_S(\mathbf{x}) &= \sum_i \frac{m_i}{\rho_i}A(\mathbf{x}_i)\nabla W(\|\mathbf{x}_i-\mathbf{x}\|,h)\\ \nabla^2 A_S(\mathbf{x}) &= \sum_i \frac{m_i}{\rho_i}A(\mathbf{x}_i)\nabla^2 W(\|\mathbf{x}_i-\mathbf{x}\|,h)\end{aligned} \tag{3.8}$$

3.3.2 Kernels

Smoothing kernels functions are one of the most important points in SPH. Stability, accurancy and speed of the whole method depends on these fuctions. Different kernels are being used for different purposes. One possibilyty for W is a Gaussian. However, most current SPH implementations are based on kernels with finite support. We mainly introduce gaussian, poly6 and spicy kernel here. And compare the different kernels and their property.

Poly6

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

$$W_{poly6}(\mathbf{r},h) = \frac{315}{64\pi h^9}\begin{cases}(h^2-\|\mathbf{r}\|^2)^3 & 0 \leq \|\mathbf{r}\| \leq h\\ 0 & \text{Otherwise}\end{cases} \tag{3.9}$$

Then, the gradient of this kernel function can be

$$\nabla W_{poly6}(\mathbf{r},h) = -\frac{945}{32\pi h^9}\begin{cases}\mathbf{r}(h^2-\|\mathbf{r}\|^2)^2 & 0 \leq \|\mathbf{r}\| \leq h\\ 0 & \text{Otherwise}\end{cases} \tag{3.10}$$

The laplacian of this kenel can be expressed by,

$$\nabla^2 W_{poly6}(\mathbf{r}, h) = -\frac{945}{16\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)(3h^2 - 7\|\mathbf{r}\|^2) & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.11}$$

As Müller stated[8], if the kernel is 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 kerbek approaches zero to the center.', which we can see in Figure 3.4. Another kernel, spiky kernel, is proposed by Desbrum and Gascuel[9] to solve this problem.

Spiky

The kernel proposed by Desbrum and Gascuel[9]

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\mathbf{r}\|)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.12}$$

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

$$\nabla W_{spiky}(\mathbf{r}, h) = -\frac{45\mathbf{r}}{\pi h^6 \|\mathbf{r}\|} \begin{cases} (h - \|\mathbf{r}\|)^2 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.13}$$

The laplacian of spiky can be expressed by,

$$\nabla^2 W_{spiky}(\mathbf{r}, h) = \frac{90}{\pi h^6} \begin{cases} h - \|\mathbf{r}\| & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \text{Otherwise} \end{cases} \tag{3.14}$$

3.3.3 Grid size and smoothing length

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

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

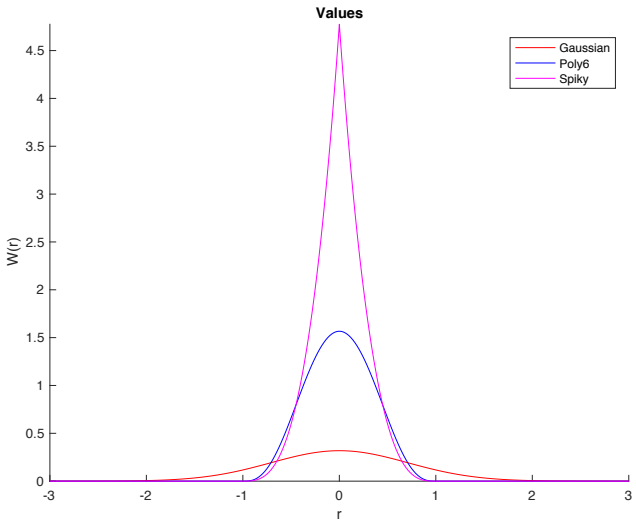


Figure 3.3: Comparation of different kernels, we set smoothing length $h = 1$ here.

3.3.4 Neignbor Search

Neighbor search is one of the most crucial procedures in SPH method considersing all interpolation equations, $A(\mathbf{r})$, needs the neighbor list for every particle (refer to equation 3.8). A naive neighbor searching approach would end up with a complexity of $o(n^2)$. The complexity is not good enough since it is impossoble to reach any interactive speed when the particle count increses. With an efficient nearest neighbor searching(NNS) algorithm, it is possible to have a significant performance increase since it is the most time consuming procedure in SPH computation. In order to decrease the complexity, we choose to use $k-d$ tree data structure to store the particla spatial information and then do the nearest neighbor searching.

Hierarchical Tree

Using an adaptive hierarchy tree search is proposed by Paiva[10] to find particle neighbors. Since the simulation takes place in two dimensions, $k-d$ tree data structure was used in this approach.

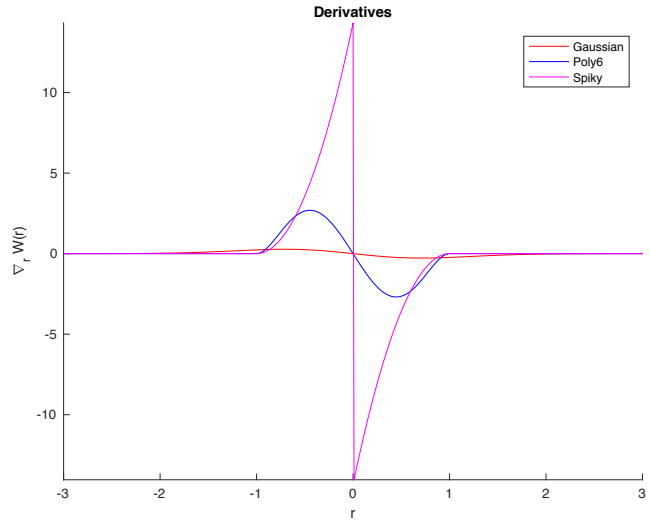


Figure 3.4: Comparation of gradient of different kernels, we set $h = 1$ here.

An octree structure has been adapted from Macey (b) Octree Demo. The hierarchy tree is formed with a pre-defined height. The simulation box is divided recursively into eight pieces, nodes. The nodes at height = 1 are called leaves. Each node has a surrounding box for particle query which is used to check if the particle is inside the node. Each parent node, contains the elements that are divided through its children. The particle is being checked at each level of the tree if it's intersecting the node. If it does, descend one level down in that node. After reaching to leaves, bottom level, particle has been checked if its within the search distance, h. If the particle is in the range, add it to the neighbor list. Neighbor searching is done when the whole tree has been traversed. The search distance set to be smoothing length in this implementation.

The complexity of this tree search method is $O(nlog(n))$, n being the number of particles. The performance of this algorithm is worse than Spatial Hashing method. In addition, the results obtained using this NNS algorithm wasn't accurate and stable for this implementation. Therefore as mentioned above, Spatial Hashing method was preferred.

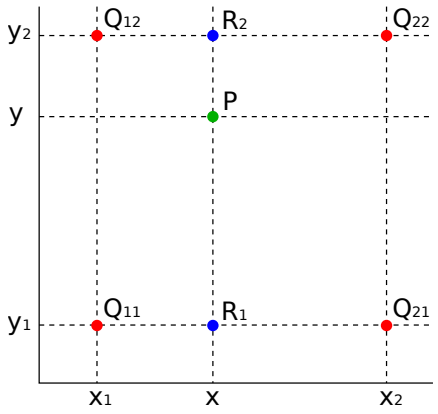


Figure 3.5: The figure shows visiualization of bilinear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate.

3.4 Grid to particle

SPH will be used for us to transform current state of dynamic system to grid images. After getting 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. Once the contact force image is obtained, we will 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 $t + \Delta t$.

3.4.1 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 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

$$\begin{aligned} f(x, y_1) &\approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \\ f(x, y_2) &\approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \end{aligned} \tag{3.15}$$

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

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{3.16}$$

Combine $f(x, y_1)$ and $f(x, y_2)$ defined in equation 3.15, we can get,

$$\begin{aligned} f(x, y) \approx & \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) \\ & + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right) \\ = & \frac{1}{(x_2 - x_1)(y_2 - y_1)} (\\ & f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) \\ & + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1)) \\ = & \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \\ & \cdot \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix} \end{aligned} \tag{3.17}$$

3.5 Experiment and Conclution

Our hope is that strating 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

hyposis is that 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 methos in this section. More details will be analized and described in Chapter 5.

3.5.1 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.

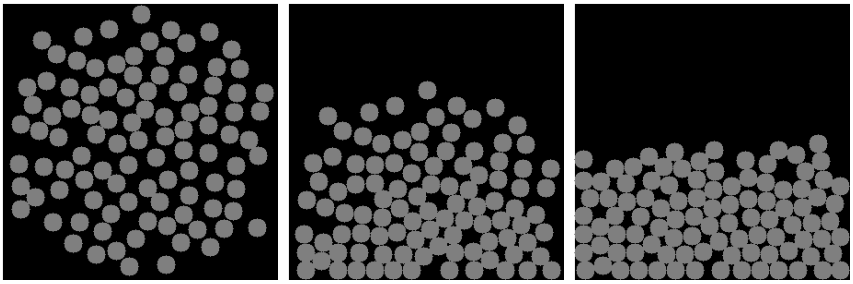
- **Non-Model** In each step of the simulation, before the contact solver starts iteration to make resolution for current dynamic system equation, the initial λ_f abd λ_t of every contact will be given value 0.
- **Builtin-Model** In each step of the simuation, before the contact solver starts iterations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

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[2]. 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

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

in eacg step will be set as fixed, 3000. Then I will use the average covergence 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

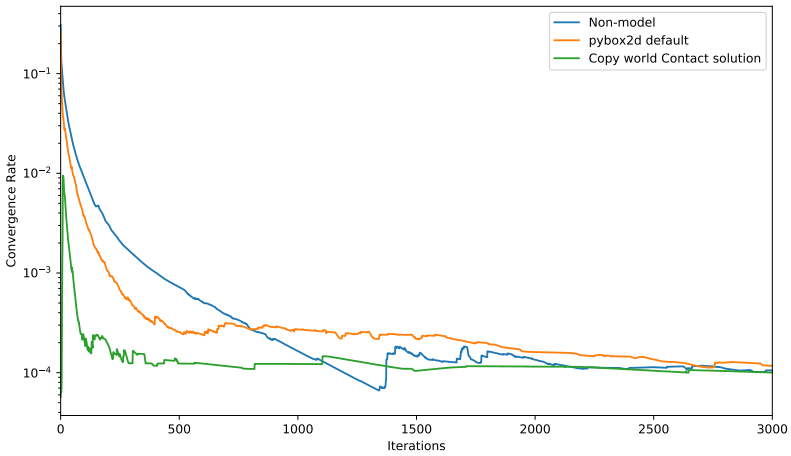


Figure 3.7: Average covergence rate for different models(not including **SPH-based model**).

The results are described in Figure 3.7. Obviously, **Copy-Model** performs the best, coverages the most rapidly. Once all the initial values for iterative solver are zero, it has to take long time to reach converagence, which is indicated by **Non-Model**. The **Builtin-Model** get converagence much fater than **Non-Model**, but still slower than **Copy-Model**. Then one hypothesis about sph based method can be,

- The performance of **SPH-Model** will converge similar to **Builin-Model**, even better.

3.5.2 SPH-based method test

The **SPH-Model** is defined as follow, as well as the experiment algorithm in 2,

- **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 will be generated by given contact point position (x_j, y_j) $j \in \mathcal{C}$ and will be used as initial values for iterative contact solver in *pybox2D*.

Data: Given a set of bodies $\mathcal{B}$ and the state in time t, as well as some information of contacts between these bodies $\mathcal{C}$.
Result: Get the contact forces $\boldsymbol{\lambda}_j = [\lambda_{nj}, \lambda_{tj}]$ $j \in \mathcal{C}$ at time t.
while *Simulation Runing* **do**

1. read current state,

$$\boldsymbol{\lambda}_j = [\lambda_{nj}, \lambda_{tj}] \quad j \in \mathcal{C}$$

2. map the solved contact to a gird image, $\mathbf{x} = (x_i, y_i)$ is the spatial position of one node in in grid image.

$$G_{\boldsymbol{\lambda}}(\mathbf{x}) \equiv \sum_{j \in \mathcal{C}} W(\mathbf{x}, \boldsymbol{q}_j)\boldsymbol{\lambda}_j, \quad \boldsymbol{\lambda} = [\lambda_n, \lambda_t]$$

3. Once the contact force image is obtained, the conatct position $\boldsymbol{q}_j = (x_j, y_j)$ $j \in \mathcal{C}$ will be used to interpolation image values based on Equation 3.17.

$$\boldsymbol{\lambda}_j \approx G_{\boldsymbol{\lambda}}(\boldsymbol{q}_j) \quad j \in \mathcal{C}$$

The interpolated values $\boldsymbol{\lambda}$ will be used as restarting iterated for *pybox2d* conatc solver.

4. Update t

$$t = t + \Delta t$$

end

Algorithm 2: Experiment algorithm for test **SPH-Model**

As the hypothesis we expect, **SPH-Model** converages even faster than **builtin-Model**. This can demonstrade the **sph-based** method

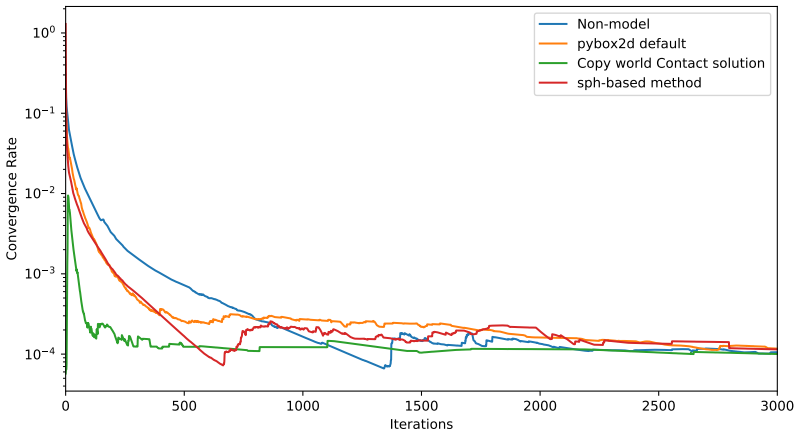


Figure 3.8: Covergence rate for different models(including **SPH-based model**).

is a good strategy, which can make itervative contact solver coverage quickly.

3.5.3 Conlusion

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 acheived, the interpolated values will help the iterative solver to get convergence faster. So, the next step is to train a suitable model based on training data. The overall algorithm is concluded in Algorithm 3.

Data: Given a set of bodies $\mathcal{B}$ and the state in time t, as well as some information of contacts between these bodies $\mathcal{C}$.

Result: Get the contact forces $\boldsymbol{\lambda}_j = [\lambda_{nj}, \lambda_{tj}] \;\; j \in \mathcal{C}$ at time t.

while *Simulation Runing* **do**

1. read current state, $\mathbf{x} = (x_i, y_i)$ is the spatial position of one node in in grid image.

$$m_i, \boldsymbol{v}_i, \boldsymbol{q}_i, \omega_i \;\; i \in \mathcal{B}$$

$$\boldsymbol{n}_j \;\; j \in \mathcal{C}$$

2. map the current state to a gird image,

$$G_m(\mathbf{x}) \equiv \sum_{i \in \mathcal{B}} W(\mathbf{x}, \boldsymbol{q}_i) m_i$$

$$G_{\boldsymbol{v}}(\mathbf{x}) \equiv \sum_{i \in \mathcal{B}} W(\mathbf{x}, \boldsymbol{q}_i) \boldsymbol{v}_i, \quad \boldsymbol{v} = (v_x, v_y)$$

$$G_{\boldsymbol{n}}(\mathbf{x}) \equiv \sum_{i \in \mathcal{C}} W(\mathbf{x}, \boldsymbol{q}_i) \boldsymbol{n}_i, \quad \boldsymbol{n} = (n_x, n_y)$$

$$G(\mathbf{x}) = [G_m(\mathbf{x}), G_{\boldsymbol{v}}(\mathbf{x}), G_{\boldsymbol{n}}(\mathbf{x})]$$

3. Use $G(\mathbf{x})$ as the input image to the learning model, the output will be resize to a imag, the output image will be called G_{output},

$$G_{output}(\mathbf{x}) = [G_{\lambda_n}(\mathbf{x}), G_{\lambda_t}(\mathbf{x})]$$

4. Once the contact force image is obtained, the conatct position $\boldsymbol{q}_j = (x_j, y_j) \;\; j \in \mathcal{C}$ will be used to interpolation image values based on Equation 3.17.

$$\boldsymbol{\lambda}_j \approx G_{output}(\boldsymbol{q}_j) \;\; j \in \mathcal{C}$$

The interpolated values $\boldsymbol{\lambda}$ will be used as starting iterated for *pybox2d* conatc solver.

5. Update t

$$t = t + \Delta t$$

end

Algorithm 3: Introducrion to the deep contact model solver in this thesis.

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

Deep learning is a branch of machine learning. It is an algorithm that attempts to use high-level abstraction of data using multiple processing layers consisting of complex structures or multiple nonlinear transforms. Deep learning is an algorithm based on the representation learning of data in machine learning. Observations (e.g., an image) can be represented in a variety of ways, such as a vector of each pixel intensity value, or more abstractly represented as a series of edges, regions of a particular shape, and the like. It is easier to learn tasks from instances using some specific representation methods (for example, face recognition or facial expression recognition). The advantage of deep learning is to replace the manual acquisition feature with unsupervised or semi-supervised feature learning and

hierarchical feature extraction efficient algorithms [11].

Early works on Deep Learning, or rather on Cybernetics, as it used to be called back then, have been made in 1940-1960s, and describe biologically inspired models such as the Perceptron, Adaline, or Multi Mayer Perceptron [11], [12]. Then, a second wave called Connectionism came in the 1960s-1980s with the invention of back-propagation [13]. This algorithm persists to the present day and is currently the algorithm of choice to optimize Deep Neural Networks. A notable contribution is the Convolutional Neural Networks (CNNs) designed, at this time, to recognize relatively simple visual patterns, such as handwritten characters [14]. Finally, the modern era of Deep Learning has started in 2006 with the creation of more complex architectures [15]–[17]. Since a breakthrough in speech and natural language processing in 2011, and also in image classification during the scientific competition ILSVRC in 2012, Deep Learning has conquered many Machine Learning communities, such as Reddit, and won challenges beyond their conventional applications area [1].

Especially during the last four years, Deep Learning has made a tremendous impact in computer vision reaching previously unattainable performance on many tasks such as image classification, objects detection, image segmentation or image captioning [6]. This progress have been made possible by the increase in computational resources, thanks to frameworks such as TensorFlow[2], modern GPUs implementations such as Cudnn[3], the increase in available annotated data, and the community-based involvement to open source codes and to share models. These facts allowed for a much larger audience to acquire the expertise needed to train modern convolutional networks. Thus, larger and deeper architectures are trained on bigger datasets to achieve better accuracy each year. Also, already trained models have shown astonishing results when transfered on smaller datasets and evaluated on different visual tasks.

With deep learning becoming more and more popular in many fields of researching, some classical methods can be replaced by

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

[2] `https://www.tensorflow.org/`

[3] `https://developer.nvidia.com/cudnn`

deep learning. Many successful application of deep learning can be found in computer vision part, like image segmentation[18],objection recognition[19].

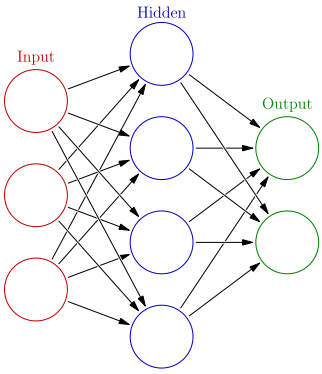


Figure 4.1: visulization of one simple 3-layers neural networks, including input layer, hidden layer and output layer, *retrieved from Wikipedia*

4.2 Convolutional Neural Networks

An entire convolutional neural network consists of an input and output layer, as well as multiple hidden layers. Normally, the hidden layers consist of convolutional layer, pooling layers, fully conneted layers and normalization layers.

Description of the process as a convolution in neural networks is by convention. Mathematically it is a cross-correlation rather than a convolution. This only has significance for the indices in the matrix, and thus which weights are placed at which index.

4.2.1 Convolutions

It turns out that there is a very efficient way of pulling this off, and it makes advantage of the structure of the information encoded within an image – it is assumed that pixels that are spatially closer together will "cooperate" on forming a particular feature of interest much more than ones on opposite corners of the image. Also, if a particular (smaller) feature is found to be of great importance when defining an image's label, it will be equally important if this

feature was found anywhere within the image, regardless of location.

Enter the convolution operator. Given a two-dimensional image, I, and a small matrix, $\boldsymbol{K}$ of size $h \times w$, (known as a convolution kernel), which we assume encodes a way of extracting an interesting image feature, we compute the convolved image, $\boldsymbol{I} * \boldsymbol{K}$, by overlaying the kernel on top of the image in all possible ways, and recording the sum of elementwise products between the image and the kernel:

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

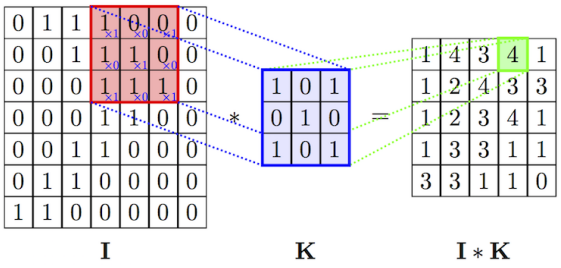


Figure 4.2: One simple example of convolution.

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 includes 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 an 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. Animations showing different kind of convolution can be viewed on line[4].

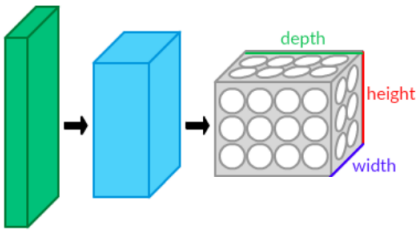


Figure 4.3: Visualization of convolutions network, *retrieved from Wikipedia*

4.2.3 Activation Layer

The Activation functions are an extremely important feature of the artificial neural networks. They basically decide whether a neuron

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

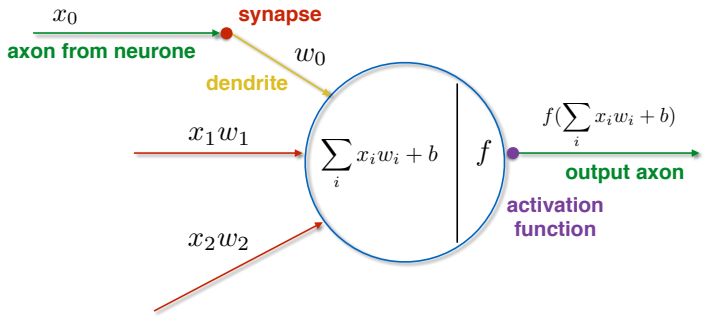


Figure 4.4: Mathematical model for describing activation function

should be activated or not. Whether the information that the neuron is receiving is relevant for the given information or should it be ignored. 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.

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

The activation function is the non linear transformation that we do over the input signal. This transformed output is then sen to the next layer of neurons as input. The most widely used activation function in networks today is Rectified Linear Unit(ReLU). More details will be talked below.

Rectified Linear Unit

The ReLU has the following mathematical form,

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

The ReLU has become very popular in the last few years, because it was found to greatly accelerate the convergence of stochastic gradient descent compared to the $sigmoid/tanh$ functions due to its linear non-saturating form (e.g. a factor of 6 in [20]). In fact, it does not suffer from the vanishing or exploding gradient. An other advantage is that it involves cheap operations compared to the expensive exponentials. However, the ReLU removes all the negative informations and thus appears not suited for all datasets and architectures.

4.2.4 Pooling Layer

Pooling layers are also known as downsampling layers. A commonly used pooling is maxpooling(figure 4.5). The downsampled output is produced by taking the maxium input value within the kernel, resulting in output 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 was often used historically but has recently fallen out of favor compared to max pooling, which works better in practice[21].

There are two important arguments for implementinf pooling layers,

1. Decreasing the number of weights.
2. Decreasing the chance of overfitting the training data.

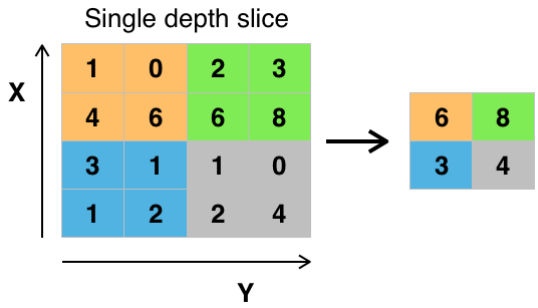


Figure 4.5: Maxpooling with a (2×2) kernel and stride $s = 2$. Maxpooling layers reduce spatial dimension of the input [22]

4.2.5 Fully Connected Layer

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

4.2.6 Batch Normalization

This layer quickly became very popular mostly because it helps to converge faster[23]. It adds a normalization step (shifting inputs to zero-mean and unit variance) to make the inputs of each trainable layers comparable across features. By doing this it ensures a high learning rate while keeping the network learning.

Also it allows activations functions such as TanH and Sigmoid to not get stuck in the 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 be 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[24].

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

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, in such a great extend 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

The first main approach to overcome overfitting is the classical weight decay, which adds a term to the cost function to penalize the parameters in each dimension, preventing the network from exactly modeling the training data and therefore help generalize to new examples:

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

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

Data augmentation

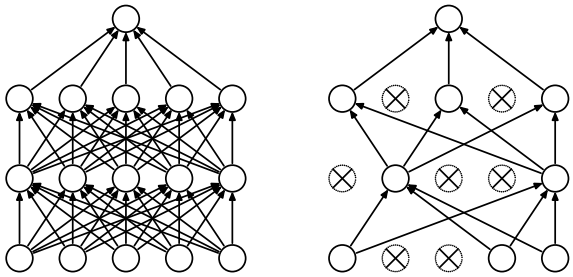
It is a method of boosting the size of the training set so that the model cannot memorize all of it. This can take several forms depending of the dataset. For instance, if the objects are supposed to be invariant to rotation such as galaxies or planktons, it is well suited to apply different kind of rotations to the original images.

Dropout

Dropout layers[25] 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 this data. The difference in performance between training data and validation data will decrease. 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

It consists in stopping the training before the model begins to overfit the training set. In practice, it is used a lot during the training 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 minize 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 determins the step size to get to the local or global minumum.

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

Mini-batch Stochatic 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

It is the kind of optimization used in parallel computing environments. Different computers train the same architecture with almost the same parameters values. It allows more exploration of the parameters space, which can lead to improved performance [26].

4.3.4 Learning Rate Scheduling in Gradient Optimization

There are serval variants of SGD available. Determining the approciate learning rate, or step size, often is a complex problem. Applying too high learning rate cause suboptimal performance, too low learning rate caused slow covergence. 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. Below an overview of some gradient descent optimizztion algoritms is given.

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 mathesmatical details are shown 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 the Nesterov Accelerated Gradient method computes an approximation of the next position of the parameters. 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 [27] determines an adaptive learning rate for each parameter. Besides decaying average of past squared gradients v_n, Adam keeps an exponentially decaying average of past gradients m_n. Vectors v_n and m_n are estimates of the mean and the uncentered variance of the gradients respectively which 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[28] is an efficient method for computing gradients required to perform gradient-based optimization of the weights in neural networks [19]. The specific combination of weights which minimize the loss function (or error function) is the solution of the optimization problem. The method requires the computation of the gradient of the error function at each iteration, therefore the loss function should be both continue and differentiable at 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 the 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 forward pass, the inputs and desired outputs are presented. A pre-defined loss function L is used to minimize the difference between 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.

Therefore the core of the backpropagation problem is to compute the gradient of the loss function with respect to the network weights. 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 nerual networks.

Backpropagation Example for a Multi-Layer Network

I describe the details of backpropagation algorithm for one simple example in Figure 4.7. The cost function L is given below, el is the error between the true outpt 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:

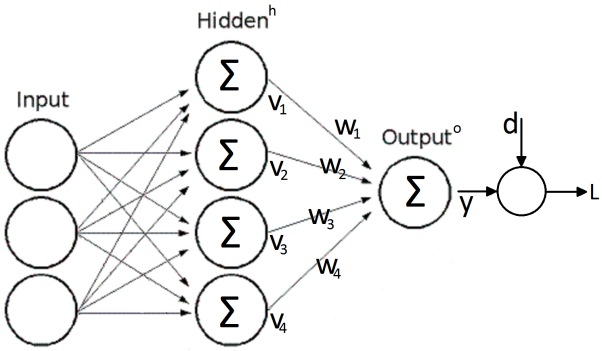


Figure 4.7: The example for showing the details of backpropagation

$$
\begin{aligned}
L &= \frac{1}{2}\sum_l (e_l)^2 \\
e_l &= d_l - y_l \\
y_l &= \sum_j w_j^o v_j
\end{aligned}
\tag{4.11}
$$

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,

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

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 by:

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

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 caculate the Jacbian,

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

Which yields the update rule for the hidden layers:

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

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

Implementation Details

This chapter mainly talk about

1. Do the simulation based one computer physics library. Totally, 100 different rigid motion simulation should be finished. For every simulation, fixed steps should be recorded.

2. Restore information of each state of per simulation by using **XML** formats, including positions, velocities, contact forces, etc.

3. Read **XML** file, and then generate some grid images for training.

4. Do the deep learning based on training dataset which is created by last step.

5. Apply the trained model to initialize the values of contact forces(λ). Then compare deep learning method and classical methods.

5.1 Rigid Motion Simulation

pybox2d[1] is chosen as the main physics engine to implemente computer simulation experiments. **pybox2d** is a 2D physics library for your games and simple simulations. It's based on the Box2D library, written in *C++*. It supports several shape types (circle, polygon, thin line segments), and quite a few joint types (revolute, prismatic, wheel, etc.).

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

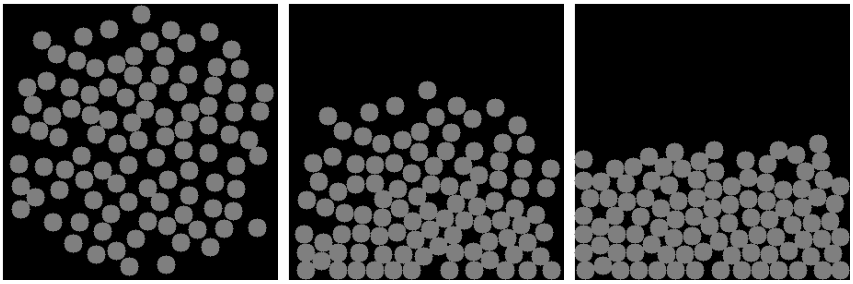
5.1.1 Simulation Configuration

- **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 eacg step will be set as fixed, 3000. Then I will use the average covergence rate to show how fast the model coverages.

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.



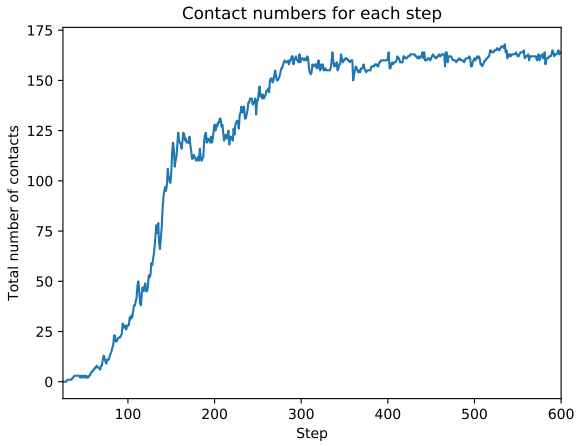
(a) Time Step=0 (b) Time Step=200 (c) Time Step=400

Figure 5.1: Visualization for experiment simulation

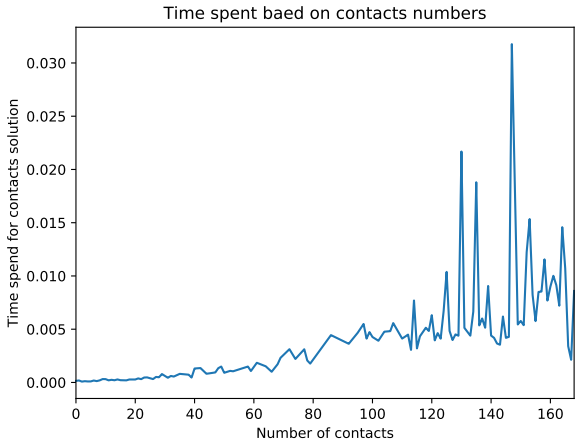
5.1.3 SPH parameters

In Section 3.5.2, it has been tested that **SPH** is a good strategy to generate grid images representing a discrere snapshot of the dynamics. Then, the grid size and smoothing length h are import to our data generation. The grid size should be reasonable so that there is

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



(a) The number of contacts in each step.



(b) Time spend for contact solution.

Figure 5.2: Visualization for experiment simulation

no two contact points are mapped into the same cell, since if there are more than two contact values in one cell, it will be hard for CNN to recognize, which will decrease the accuracy of prediction. This was mentioned in section 3.3.3.

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.

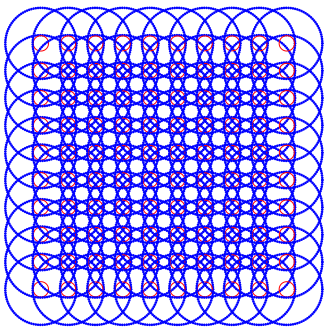


Figure 5.3: Example visiualization for Smoothed Particle Hydrodynamics. The small red circles stand for the grid nodes, and the blue circles stand for kernel size.

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

$$d_x = d_y = d$$

- 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, one contact point can only be mapped to one cell, so,

$$h \leq d$$

- 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 5.4.

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

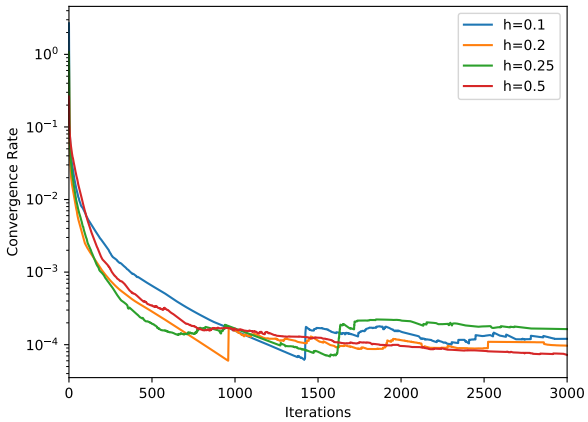


Figure 5.4: The grid size d is set 0.25. $h = 0.1, 0.2, 0.25, 0.5$ is tested respectively. This figure shows different coveragence rate based on different h value.

step, data generation, I will make $h = d$ always. The next step is to explore what d value will be good. $h = d$ 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 5.6. From this figure, it is obviously, when $d = 0.5$, iterative solver converages the most rapidly. Based on

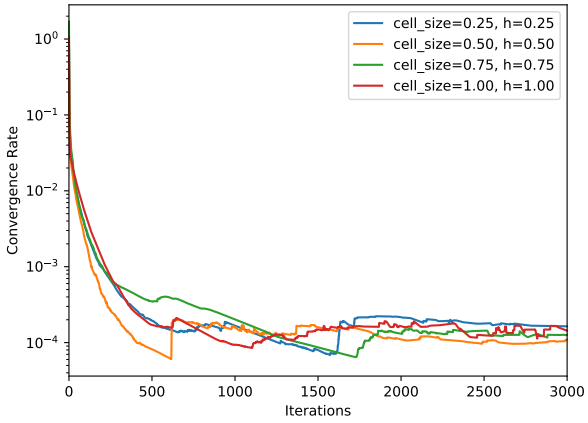


Figure 5.5: Coveragence rate for different d value.

these experiments, the grid cell size and smoothing length can be

determinde,

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

Kernel Choosing

The choice of kernel also needs to be addressed. In the paper, two kernels, **Poly6** and **Spiky**, have been introduced in section 3.3.2. These two kernels are seperately used based on Algorithm 2. Then, the plot about coveragence rate of different kernels is obtained. From the plot, it is clear that kernel **Poly6** perform better. As a con-

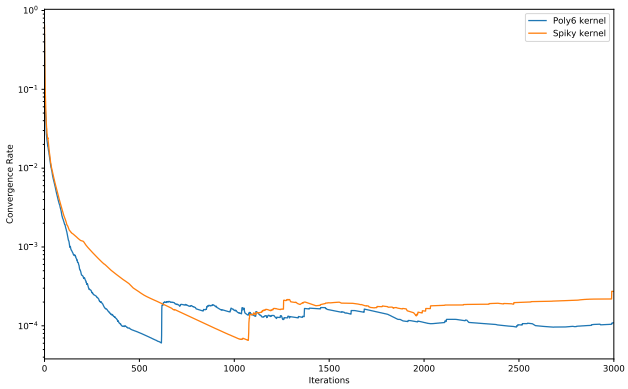


Figure 5.6: Coveragence rate for kernel **Poly6** anf **Spiky**

clusion, 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 3.5.2, following Algorithm 2. 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.

5.2 Data Generation

To create data that are more accessible to learning, we will map a discrete element method into a continuum settig use techniques from smooth particle hydrodynamics(SPH). In order to get enough

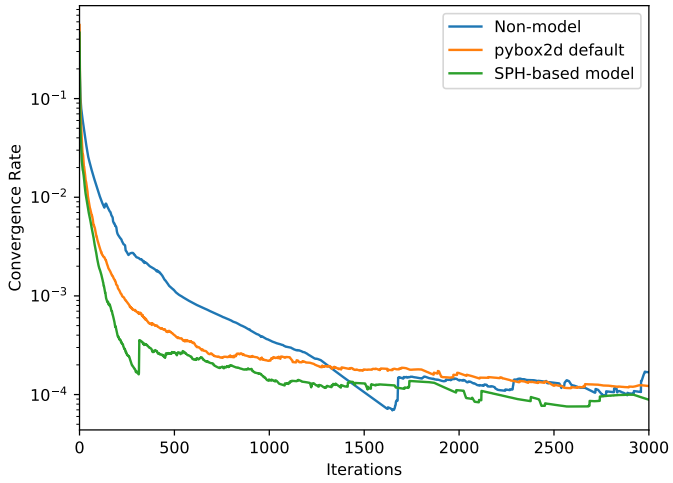


Figure 5.7: Coveragence rate for models(different initial values for $\boldsymbol{\lambda}$).

training data, 100 simulation with different initial configuration have been done.

5.2.1 XML Restoration

In each simulation, 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 is consist of postion, impulse(including in normal direction and tangent direcrion), master body and slave body. Two examples are given in the following.

This is nn example for 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 3.3.2.
- **Kernel Settings**, cell size $d_x = d_y = 0.5$, smoothing length $h = 0.5$.

5.2.3 XML to grid

After each state has been stored in *XML* file, then

5.3 CNN Training

5.3.1 CNN Architecture

The neural network was designed using Keras[29]. Keras is a neural networks Application Programming Interface (API) written in Python, it runs on top of either TensorFlow. With some inspiration from AlexNet[30], the networks shows five constracting 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 $64,498,866$ weights for an input size of $(41 \times 41 \times 5)$. All convolutional layers have kernels of size $(3 \times 3 \times d)$, and are followed by ReLU activation function. Figure 5.8 shows visualization of model architecture, and Table 5.1 shows the specific information about parameters when input images go through the CNN.

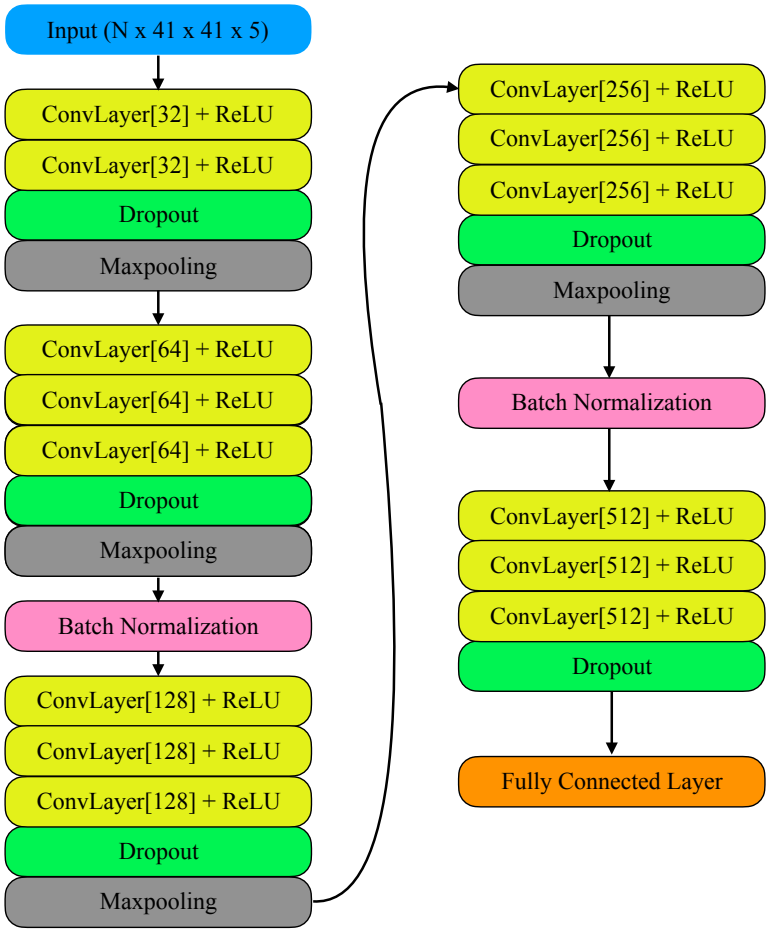


Figure 5.8: Architecture of CNN model

Layer	Output Shape
Input	$n_b \times 61 \times 61 \times 5$
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 $61 \times 61 \times 5$.

5.3.2 Traing Configuration

Loss Function

Firstly, we define a filter funtion,

$$g(x) = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases} \tag{5.1}$$

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

$$L = \frac{1}{N}\sum_{i}^{N} g(\hat{y}_i)(y_i - \hat{y}_i)^2 \tag{5.2}$$

5.3.3 Training Details

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

Hyperparameter	Setting
Activation function	ReLU
Weight initilization	He normal[31]
Weight regularizer	L2 [15]
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

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 4.

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

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

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 4: Learning Rate Scheduling

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 3 and compare it with other solutions.

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 fall down due to gravity. Figure 5.9 shows the details. As what is expeted, CNN model performs similar with **SPH-Model**(mentioned in section 3.5.2). Although the model is not as good as **SPH-model**, its coverages rate is obviously faster than buit-in ones.

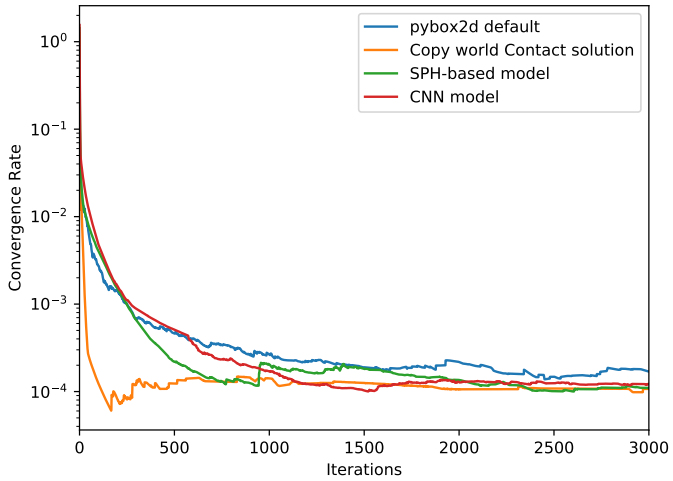

pybox2d default
Copy world Contact solution
SPH-based model
CNN model
Convergence Rate
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
500
1000
1500
2000
2500
3000
Iterations

Figure 5.9

Chapter 6

Conclusion

6.1 Future work

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares, "Particle-based non-newtonian fluid animation for melting objects", in *Computer Graphics and Image Processing, 2006. SIBGRAPI'06. 19th Brazilian Symposium on*, IEEE, 2006, pp. 78–85.

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

[12] 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.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. [Online]. Available: `http://dx.doi.org/10.1038/323533a0`.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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`.

[28] 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.

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

[30] 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`.

[31] 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.

Contact Model Simulation

```
function [lambda, theta] = pgs(A, b, lambda)

N = length(b);
K = 100;
theta = zeros(K,1);
gamma = 1.0;
r = gamma ./ diag(A);  % Initial r-factors

k = 1;
while k <= K

    lambda_old = lambda;

    for i=1:N
        res = A(i,:)*lambda + b(i);
        lambda(i) = max(0, lambda(i) - r(i) * res );
    end

    delta = lambda - lambda_old;
    theta(k) = max(abs(delta(i)));

    k = k +1;
end

end
```

Appendix B.

Smooth

Appendix C.

Deep Learning

```
    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'))

self.model.compile(
    loss=self.loss_func,
```

```
            optimizer=self.optimizer,
            metrics=self.metrics,
        )
```