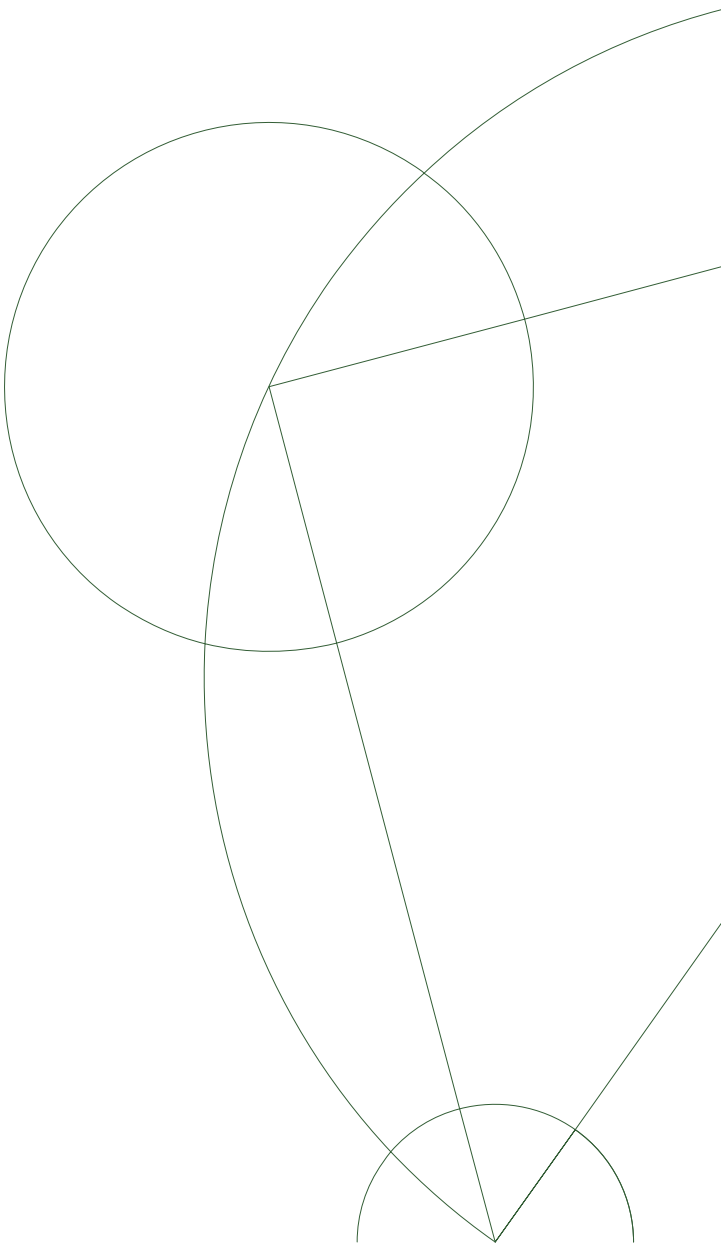
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

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 from

Contents

List of Figures

List of Tables

Chapter 1

Introdustion

1.1 Motivation

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.

1.2 Thesis Overview

This thesis explore the application of convolutional neural networks in Computer Simulation.

Chapter 2

Rigid Body Dynamics Simulation

This chapter mainly introduces rigid body simulation to help you understand how computer simulate rigid dynamics based on traditional newton-euler equations. For more details, some contact forces solvers are decribed in this chapter. Afterwards, we will use one of solver to run some simulation and get the image data for the next step, grids-transfer. All the discussion about rigid simulation and contacts solver are based on 2-D view.

2.1 Rigid dynamics Simulation

2.1.1 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 $x(t)$ and velocity vector $\boldsymbol{v}$. Then, we will introduce some concepts by adding rotatiopn to pur simulation, rotational velocity $\boldsymbol{\omega}$, and moment $\boldsymbol{\tau}$ (also known as a torque.

2.1.2 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 $\boldsymbol{p}(t)$ describe the particle's location in world space at time t. Then we use $\boldsymbol{v}(t) = \frac{\mathrm{d}}{\mathrm{d}(t)}\boldsymbol{p}(t)$ to denote the velocity of the particle at time t. 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,

$$\mathbf{Y}(t) = \begin{pmatrix} \boldsymbol{p}_1(t) \\ \boldsymbol{v}_1(t) \end{pmatrix} \tag{2.1}$$

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

$$\mathbf{Y}(t) = \begin{pmatrix} \boldsymbol{p}_1(t) \\ \boldsymbol{v}_1(t) \\ ... \\ \boldsymbol{p}_n(t) \\ \boldsymbol{v}_n(t) \end{pmatrix} \tag{2.2}$$

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

$$\frac{\mathrm{d}}{\mathrm{d}(t)}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}(t)}\begin{pmatrix} \boldsymbol{p}(t) \\ \boldsymbol{v}(t) \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}(t) \\ \boldsymbol{f}(t)/m \end{pmatrix} \tag{2.3}$$

2.1.3 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 $\boldsymbol{p}(t)$ to denote their translation and a rotation matrix $\boldsymbol{R}(t)$ to describe their rotation.

2.1.4 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 is defined as 2.5, and angular momentum is defined as 2.6

$$m\boldsymbol{v}\dot{(}t) = \boldsymbol{f}(t) \tag{2.4}$$

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

$$\boldsymbol{L}(t) = \boldsymbol{I}(t)\boldsymbol{\omega}(t) \tag{2.6}$$

The total torque $\boldsymbol{\tau}$ applied to the body is equal to the rate of change of the angular momentum, as defined in 2.12:

$$\boldsymbol{\tau} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{L} = \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{I}\boldsymbol{\omega}) \tag{2.7}$$

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

$$\mathbf{Y}(t) = \begin{pmatrix} \boldsymbol{p}(t) \\ \boldsymbol{R}(t) \\ \boldsymbol{P}(t) \\ \boldsymbol{L}(t) \end{pmatrix} \tag{2.8}$$

Like what is epressed 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.

$$\boldsymbol{v}(t) = \frac{\boldsymbol{P}(t)}{m} \quad \boldsymbol{I}(t) = \boldsymbol{R}(t)\boldsymbol{I}_{body}\boldsymbol{R}(t)^T \quad \boldsymbol{\omega}(t) = \boldsymbol{I}(t)^{-1}\boldsymbol{L}(t) \tag{2.9}$$

The derivative $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{Y}(t)$ is

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{Y}(t) = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} \boldsymbol{p}(t) \\ \boldsymbol{R}(t) \\ m\boldsymbol{v}(t) \\ \boldsymbol{L}(t) \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}(t) \\ \boldsymbol{\omega}(t) \times \boldsymbol{R}(t) \\ \boldsymbol{f}(t) \\ \boldsymbol{\tau}(t) \end{pmatrix} \tag{2.10}$$

Then, we can evaluate Equation 2.11 as follows:

$$\begin{aligned}
\boldsymbol{\tau} &= \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{I}\boldsymbol{\omega}) \\
&= \boldsymbol{I}\dot{\boldsymbol{\omega}} + \dot{\boldsymbol{I}}\boldsymbol{\omega} \\
&= \boldsymbol{I}\dot{\boldsymbol{\omega}} + \frac{\mathrm{d}}{\mathrm{d}t}(\boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T)\boldsymbol{\omega} \\
&= \boldsymbol{I}\dot{\boldsymbol{\omega}} + (\dot{\boldsymbol{R}}\boldsymbol{I}_{body}\boldsymbol{R}^T + \boldsymbol{R}\boldsymbol{I}_{body}\dot{\boldsymbol{R}}^T)\boldsymbol{\omega} \\
&= \boldsymbol{I}\dot{\boldsymbol{\omega}} + ([\boldsymbol{\omega}]\boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T + \boldsymbol{R}\boldsymbol{I}_{body}\boldsymbol{R}^T\hat{\boldsymbol{\omega}})\boldsymbol{\omega} \\
&= \boldsymbol{I}\dot{\boldsymbol{\omega}} + [\boldsymbol{\omega}]\boldsymbol{I}\boldsymbol{\omega} - \boldsymbol{I}[\boldsymbol{\omega}]\boldsymbol{\omega}
\end{aligned} \tag{2.11}$$

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

$$\boldsymbol{\tau} = \boldsymbol{I}\dot{\boldsymbol{\omega}} + [\boldsymbol{\omega}]\boldsymbol{I}\boldsymbol{\omega} \tag{2.12}$$

2.1.5 Impulse

When two bodies collide, those bodies and any other bodies, they are touching, experience very high forces of very short duration. In the case of ideal rigid bodies, the force magnitudes become infinite and the duration becomes infinitesimal. These forces are referred to as impulsive forces or shocks. One can see from Equation 2.4 that shocks cause infinite accelerations, which makes direct numerical integration of the Newton–Euler equations impossible. One way to deal with this problem during simulation is to use a standard integration method up to the time of impact, then use an impulse–momentum law to determine the jump discontinuities in the velocities, and finally restart the integrator.

Let $[t, t+\Delta t]$ be a time step during which a collision occurs. Further, define $\mathbf{p} = \int_t^{t+\Delta t} \mathrm{d}t$ as the impulse of the net force and $m\boldsymbol{v}$ as translational momentum. Integrating Equation 2.4 from t to $t+\Delta t$ yields $m(\boldsymbol{v}(t+\Delta t)-\boldsymbol{v}(t)) = \int_t^{t+\Delta t} \boldsymbol{f}\,\mathrm{d}t$, which states that Δt impulse of the net applied force equals the change of translational momentum of the body. In rigid body collisions, Δt approaches zero. Taking the limit as Δt goes to zero, one obtains an impulse momentum law that is applied at the instant of impact to compute post-collision velocities. Since Δt goes to zero and the velocities remain finite, the generalized position of the bodies are fixed during the impact. After processing the collision, one has the values of the generalized

positions and velocities, which are the initial conditions to restart the integrator. Note that integration of the rotational Equation 2.12 yields an impulse–momentum law for determining jump discontinuities in the rotational velocities.

2.1.6 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, 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 2.13

$$\mathbf{v} = \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{pmatrix} \tag{2.13}$$

. The defination can be found in 2.13, 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

$$\mathbf{f} = \begin{pmatrix} \boldsymbol{\tau} \\ \boldsymbol{f} \end{pmatrix} \tag{2.14}$$

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.

2.1.7 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 2.13 and the wrench $\mathbf{f}$ metioned in 2.14 acting on the body. We can simply write the Newton and Eular equations,

$$\begin{pmatrix} \boldsymbol{\tau} - \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} \\ \boldsymbol{f} \end{pmatrix} = \begin{pmatrix} \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & m\mathbf{1}_{d\times d} \end{pmatrix} \dot{\mathbf{v}} \tag{2.15}$$

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

If we define $\mathcal{M}$ and $\mathbf{h}$ as Equation 2.16 and **??**

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

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

So, we can rewrite Newton-Euler equation as,

$$\mathcal{M}\dot{\mathbf{v}} = \mathbf{h} \tag{2.18}$$

2.2 Constrained Dynamics

most interesting simulations of rigid bodies involve some kind of constraints. Usually we want to model systems of bodies that are interacting in some way. Some bodies may be in contact with each other, or attached together by some types of joint. Since this report mainly research contact forces, we

2.2.1 Expressing Constraints as Equations

We express constraints mathematically as algebraic matrix equations with position, velocity, or acceleration vectors as the unknowns. In general, the configuration space of a set of n rigid bodies has dimension 6n. Adding constraint equations restricts the position (or velocity, or acceleration) to a subspace of smaller dimension.

The constraints discussed in this thesis will all be holonomic constraints, which means the constraint on the velocity can be found by taking the derivative of a position constraint. Constraints that do not fit this description are called nonholonomic. An example of a nonholonomic constraint is a ball rolling on a table without slipping. The position of the ball has five degrees of freedom, so there is only one degree of constraint (only the height is constrained). Taking the derivative of the position constraint would only give a velocity

constraint of degree one. Additional constraints on the velocity are needed to prevent the ball from slipping.

We will describe position constraints with a constraint function $g(\boldsymbol{p})$, which is a function from the space of possible positions of the rigid bodies, to $\mathbf{R}^d$, where d is the number of degrees of freedom that constraint removes from the dynamic system. If the constraint function returns a zero vector, then the position $\boldsymbol{p}$ satifies the constraint.

Position constraint can be divided into *equality constraint*(e.g., joint constraints), where the constraint is $g(\boldsymbol{p}) = 0$, and *inequality constraints*(e.g., contact constraints), where $g(\boldsymbol{p}) \geq 0$.

Constraining the position of an object also constrains its velocity. The velocity constraint function can be found by taking the time derivative of the constraint function. We want the velocity constraint function to be a linear function of the twists representing the velocities of all of the rigid bodies in the system. We let $\mathbf{v}$ to be:

$$\mathbf{v} = \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \cdots \\ \mathbf{v}_n \end{pmatrix} \tag{2.19}$$

Velocity constraints are of the form

$$\frac{\mathrm{d}g_i}{\mathrm{d}t} = J_i\mathbf{v} = (\mathbf{J}_{i1}, ..., \mathbf{J}_{in}) \begin{pmatrix} \begin{pmatrix} \boldsymbol{\omega}_1 \\ \boldsymbol{v}_1 \end{pmatrix} \\ \begin{pmatrix} \boldsymbol{\omega}_2 \\ \boldsymbol{v}_2 \end{pmatrix} \\ \cdots \\ \begin{pmatrix} \boldsymbol{\omega}_n \\ \boldsymbol{v}_n \end{pmatrix} \end{pmatrix} \tag{2.20}$$

where i is unique for each constraint. The matrix J is called the constraint's *Jacobian matrix*, which we will refer to simply as the Jacobian.

When there are many constraints, like a system with n bodies, m constraints, the velocity constraint equation looks like

$$\begin{aligned}\frac{\mathrm{d}g}{\mathrm{d}t} &= \begin{pmatrix} g_1 \\ g_2 \\ ... \\ g_m \end{pmatrix} = \mathcal{J}\mathbf{v} = \begin{pmatrix} J_1 \\ ... \\ J_n \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ ... \\ \mathbf{v}_n \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{J}_{11} & \boldsymbol{J}_{12} & ... & \boldsymbol{J}_{1n} \\ \boldsymbol{J}_{21} & \boldsymbol{J}_{22} & ... & \boldsymbol{J}_{2n} \\ ... & ... & ... & ... \\ \boldsymbol{J}_{m1} & \boldsymbol{J}_{m2} & ... & \boldsymbol{J}_{nn} \end{pmatrix} \begin{pmatrix} \begin{pmatrix} \boldsymbol{\omega}_1 \\ \boldsymbol{v}_1 \end{pmatrix} \\ \begin{pmatrix} \boldsymbol{\omega}_2 \\ \boldsymbol{v}_2 \end{pmatrix} \\ ... \\ \begin{pmatrix} \boldsymbol{\omega}_n \\ \boldsymbol{v}_n \end{pmatrix} \end{pmatrix}\end{aligned} \tag{2.21}$$

Notes:

- $\boldsymbol{J}$ is used for Jacobians that relate the velocity of a single body to a single constraint.
- J is used for Jacobians that relate the velocity of mant bodies with many bodies to a single constraint.
- $\mathcal{J}$ is used for Jacobians that relate velocity of many bodies to many constraints.

2.2.2 Solving Constrained Dynamics Equations using Largange Multipliers

A constraint Jacobian matrix $\mathcal{J}$ has a dual use. In addition to relating the vlocities to the rate of change of the constraint function $\boldsymbol{g}$, the rows of $\mathcal{J}$ act as basis vectors for constraint forces. Thus, actually we do just need to solve for the coefficient vector $\boldsymbol{\lambda}$, which is called the *Lagrange multipliers* that contains the magnitudes of the forces that correspond to each of these basis vectors.

Solving Dynamics at the Force-Acceleration Level

Combining the Newton-Euler equations,

$$\mathcal{M}\dot{\mathbf{v}} = \mathcal{J}^T\boldsymbol{\lambda} + \mathbf{h}_{ext} \tag{2.22}$$

where $\mathcal{M}$ was defined in Equation 2.16, $\mathbf{h}_{ext}$ holds external and gyroscopic force terms.

2.2.3 Contact Constraints

From other papers [2], we can conclude some import features of contact constraint:

- Contact forces can push bodies apart, but cannot pull bodies towards each other. This leads to inequalities in the constraint equations.
- Contact are transient - they come and go. Contacts can appear suddenly when bodies are moving towards each other, resulting in an impact.

Impacts between rigid bodies in reality result in large forces applied in over a very short time period, leading to a sudden changes in velocity. There are serveral methods for handling contact in rigid body simulations. Mirtich[3] give a summary of serval different methods, and explains the advantages and disadvantages of each. We mainly introduced one method called .... from Kenny[4]

- A contract consist of a pair of contact points, one point attached, one point attached to one rigid body, and the other point attached to another rigid body. The contact points are sufficiently close together for our collision detection algorithm to report a collision. Normally, we caculate the distance between two rigid bodies(mainly on circle shape).

$$g_i = \|\boldsymbol{p}_{i1} - \boldsymbol{p}_{i2}\| - |r_{i1} + r_{i2}| \tag{2.23}$$

- A contact normal is a unit vector that is normal to one or both of the surfaces at the contact points.
- The contact constraint Jacobian for constraint i, denoted by $\boldsymbol{J}_{c_i}$, is a $1 \times 6n$ matrix, where n is the number of bodies (the subscript 'c' here stands for 'contact').
- A contact force $\boldsymbol{J}_{c_i}^T \lambda_{c_i}$ is a force acting in the direction of the contact normal which prevents the two rigid bodies from interpenetrating.
- The separation distance of a contact is the normal component of the distance between the two contact points. It is negative when the bodies are interpenetrating at the contact. The constraint function $g_i(\boldsymbol{p})$ for a contact constraint returns the separation distance. In other words, $g_i \geq 0$.

- The relative normal acceleration a_i of a contact is the second derivative of the separation distance with respect to time ($a_i = \ddot{g}_i$). The acceleration constrait is satisfied when $a_i = \boldsymbol{J}_{c_i}\dot{\mathbf{v}}_i + k_i \geq 0$

When g or $\dot{g}$ are positive, it indicates that the bodies are not touching, or moving apart at that contact, respectively, in other words, no acceleration is needed. The assumption that at each resting contact $g = 0$ and $\dot{g} = 0$ are critical to the constraint that we enumerate below.

Let $\boldsymbol{a}$ be a vector containing the relative normal accelerations $\{a_1, ..., a_n\}$ for all contacts and $\boldsymbol{\lambda}_c$ be a vector of contact force multipliers $\{\lambda_{c_1}, ..., \lambda_{c_n}\}$. The vectors $\boldsymbol{a}$ and $\boldsymbol{\lambda}_c$ are linearly related at a given $\mathbf{p}$. Additionally, we have three constraints:

1. The relative normal accelerations must be nongative:

$$\boldsymbol{a} = \mathcal{J}_c\dot{\mathbf{v}} + \boldsymbol{k} \geq 0 \tag{2.24}$$

   Since g and $\dot{g}$ are zero, a negative acceleration would cause interpenetration.

2. The contact force magnitudes must be nonnegative (so as to push the bodies apart):

$$\boldsymbol{\lambda}_c \geq \mathbf{0} \tag{2.25}$$

3. For each contact i, at least one of a_i, λ_{c_i} must be zero, we can write that

$$a_i^T \lambda_{c_i} = 0 \tag{2.26}$$

   Since there can only be a contact force if the bodies are actually touching($g_i\lambda_{c_i} = 0$). Differentiating this twice using chain rule, we get $\ddot{g}_i\lambda_{c_i} + 2\dot{g}\dot{\lambda}_{c_i} + g_i\lambda_i = 0$. Since we assume $g_i = 0$ and $\dot{g}_i = 0$, the second and third term cancel out leaving simply:

$$\ddot{g}_i\lambda_{c_i} = 0 \tag{2.27}$$

   In other words:

$$\boldsymbol{a}^T\boldsymbol{\lambda}_c = \mathbf{0} \tag{2.28}$$

Then we can write the whole system equation as Equation **??**

$$\begin{pmatrix} \mathbf{0} \\ \boldsymbol{a} \end{pmatrix} = \begin{pmatrix} \mathcal{M} & -\mathcal{J}_c^T \\ \mathcal{J}_c & 0 \end{pmatrix} \begin{pmatrix} \dot{\mathbf{v}} \\ \boldsymbol{\lambda}_c \end{pmatrix} - \begin{pmatrix} \mathbf{h}_{ext} \\ -\boldsymbol{k}_c \end{pmatrix} \tag{2.29}$$

With the condition,

$$\begin{cases} \boldsymbol{a} \geq 0 \\ \boldsymbol{\lambda}_c \geq 0 \\ \boldsymbol{a}^T \boldsymbol{\lambda}_c = \mathbf{0} \end{cases} \tag{2.30}$$

2.2.4 Linear Complementarity Problems(LCP)

A *Linear Complementarity Problem*(LCP) is, given a $n \times n$ matrix $\boldsymbol{M}$ and a n-vector $\boldsymbol{q}$, the problem of finding values for the variables $\{z_1, z_2 ... z_n\}$ and $\{\omega_1, \omega_2, ..., \omega_n\}$ like,

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ ... \\ \omega_n \end{pmatrix} = \boldsymbol{M} \begin{pmatrix} z_1 \\ z_2 \\ ... \\ z_n \end{pmatrix} + \boldsymbol{q} \tag{2.31}$$

and for all i from 1 to n, *complementarity constraints* can be,

$$\begin{cases} z_i \geq 0 \\ \omega_i \geq 0 \\ z_i \omega_i = 0 \end{cases} \tag{2.32}$$

2.3 Models for Systems with Frictional Contacts

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 2.29).Anitescu and Potra [5] 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 2.29, the acceleration can be approximated by [5] as:

$$\dot{\mathbf{v}} \approx \frac{(\mathbf{v}_{t+h} - \mathbf{v}_t)}{h} \tag{2.33}$$

$\mathbf{v}_t$ and $\mathbf{v}_{t+h}$ are the velocities at the beginning of the current time step, and the next time step, h is the time step. We then arrive at the mixed LCP

$$\begin{pmatrix} \mathbf{0} \\ \boldsymbol{a} \end{pmatrix} = \begin{pmatrix} \mathcal{M} & -\mathcal{J}_c^T \\ \mathcal{J}_c & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_{t+h} \\ \boldsymbol{\lambda}_c \end{pmatrix} + \tag{2.34}$$

2.3.1 Friction Constraints

Coulomb friction is intrdiced by adding some addtional forces and constraints to the LCP 2.34

In true Coulomb friction, the magnitude of the friction force must be less than μ times the magnitude of the normal force:

$$\|\boldsymbol{f}_f\| \leq \mu \|\boldsymbol{f}_n\| \tag{2.35}$$

Geometrically, the contact forces is restricted to lie inside of a cone. However, to fit friction into our linear algebraic framework, it helps to approximate the friction cone with a ploy hedral cone. We do this by choosing a set of direction vectors $\boldsymbol{d}_{i1}, \boldsymbol{d}_{i2} ... \boldsymbol{d}_{in}$. The expected frictional impulse for contact i as:

$$J_{f_i}^T \boldsymbol{\lambda}_{f_i} = \begin{pmatrix} \boldsymbol{d}_{i1} & \boldsymbol{d}_{i2} & ... & \boldsymbol{d}_{in} \end{pmatrix} \begin{pmatrix} \lambda_{f_{i1}} \\ \lambda_{f_{i2}} \\ ... \\ \lambda_{f_{in}} \end{pmatrix} \tag{2.36}$$

2.3.2 Projected Gauss–Seidel(PGS) solver for contact

Chapter 3

Partcle-grid-particle

Training data is always one import part to normal deep learning cases. Commonly, many important visual

3.1 Overview

Deep learning methods have been widely applied in computer vision, like segmentation, objects recognision. Howver, there are not too many cases for making deep learning work for rigid simulation. The biggest difficulty of applying neural networks in rigid simulation is to make learning networks extract essential information from visualization images of simulation. For example, there is more

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 δ and a set of contacts between these bodies C.

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

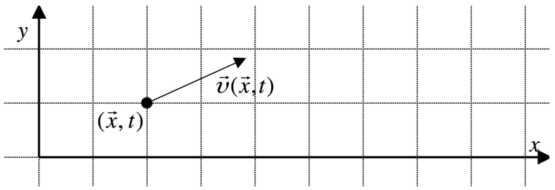


Figure 3.1: Grid description, *retrieved from MIT*(2011)

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.

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 $W(\mathbf{r}, h)$,

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

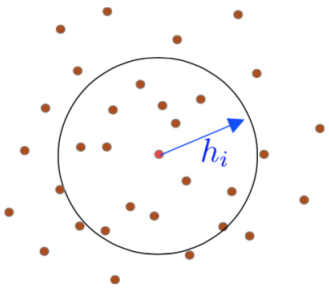


Figure 3.2: Visilaztion of SPH

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 spicky 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.

Spicky

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,

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

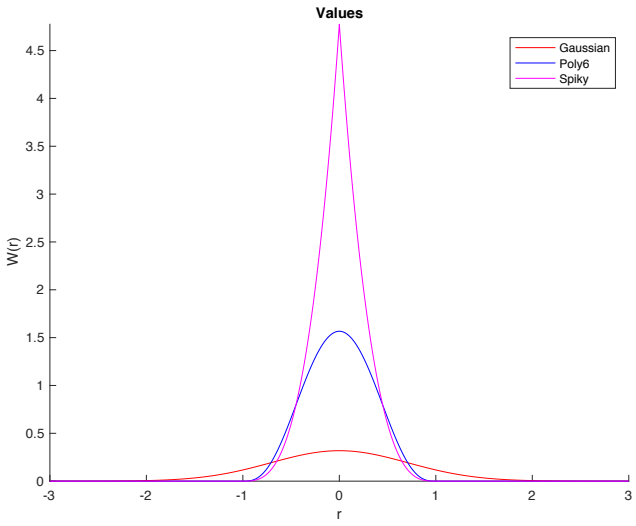


Figure 3.3: Comparation of different kernels, we set smoothing length $h = 1$ here.

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.

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 signifi-

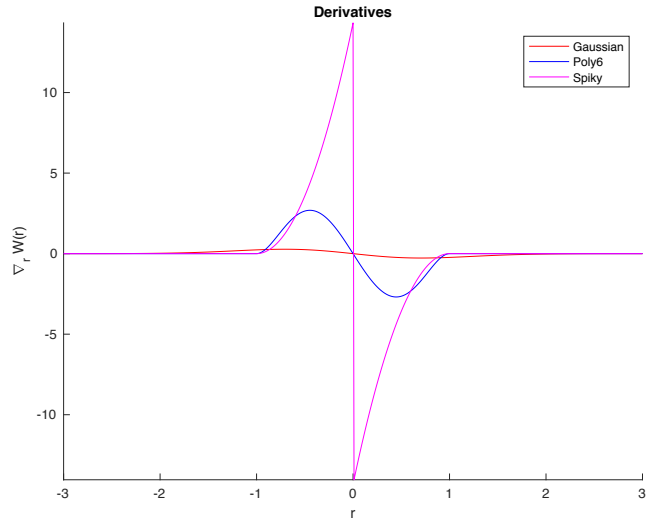


Figure 3.4: Comparation of gradient of different kernels, we set $h = 1$ here.

cant 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.

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.

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

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

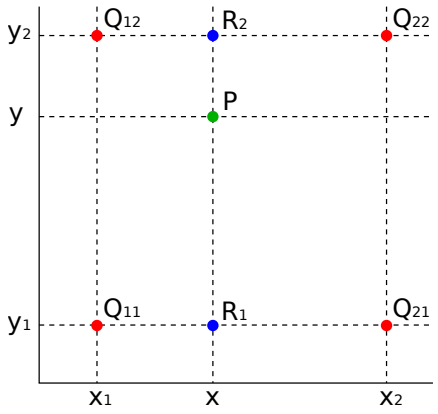


Figure 3.5: The figure shows visiualization of bilinear interpolation. The four red dots show the data points and the green dot is the point at which we want to interpolate.

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 re-start 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.

Chapter 4

Deep Learning

This chapter consists of Four parts. The first section gives a brief introduction to deep learning neural networks, including its history, development and current application. The second section mainly decribes details of about Convolutional Neural Networks, including different types of layers and their functionas. The last section talks about the techqiues used in deep learning training process, which make training get convagence faster and more accurate.

4.1 Introduction

Deep Learning can be summed up as a sub field of Machine Learning studying statical models called deep neural networks. The latter are able to learn complex and hierarchical representations from raw data, unlike hand crafted models which are made of an essential features engineering step. This scientific field has been known under a variety of names and has seen a long history of research, experiencing alternatively waves of excitement and periods of oblivion [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 backpropagation [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 Ma- chine 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.

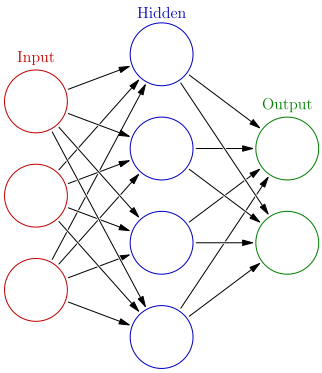


Figure 4.1: visulization of one simple 3-layers neural networks, including input layer, hidden layer and output layer, *retrieved from Wikipedia*

[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`

With deep learning becoming more and more popular in many fields of researching, some classical methods can be replaced by deep learning. Many successful application of deep learning can be found in computer vision part, like image segmentation[18],objection recognition[19].

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:

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

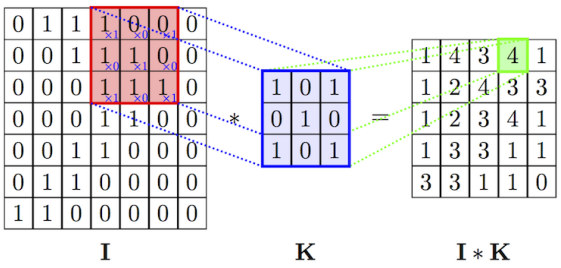


Figure 4.2: One simple example of convolution.

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].

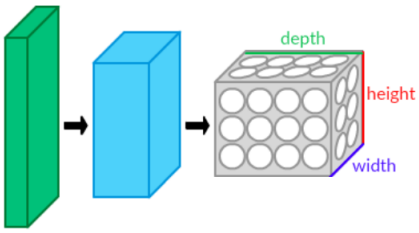


Figure 4.3: Visualization of convolutions network, *retrieved from Wikipedia*

4.2.3 Activation Layer

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

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

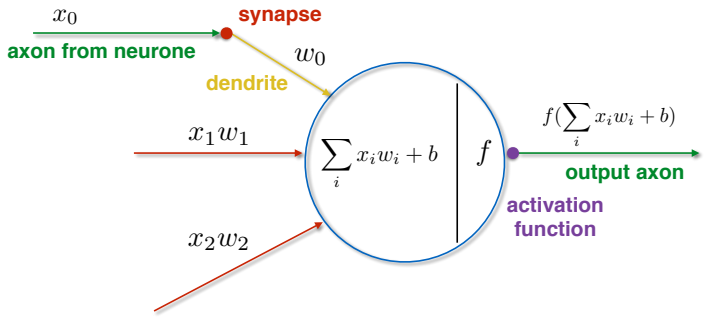


Figure 4.4: Mathematical model for describing activation function

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.

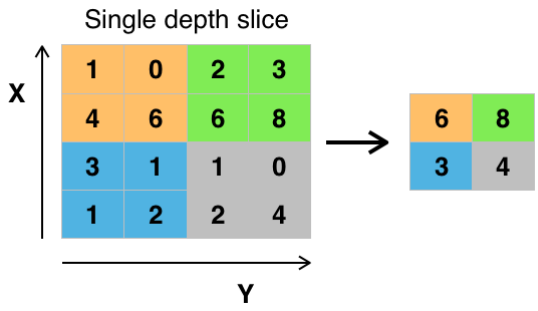


Figure 4.5: Maxpooling with a (2×2) kernel and stride $s = 2$. Maxpooling layers reduce spatial dimension of the input [22]

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 x_i are the neuron outputs and $\hat{x}_i$ 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

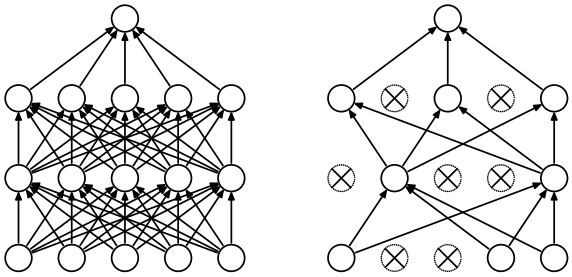
Dropout layers[25] are a tool to prevent overfitting (Figure 4.6). In dropout, nodes and its connections are randomly dropped from the network. Dropout constrains the network adaptation to the training set, consequently it prevents that the weights are not too much fitted this data. The difference in 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.

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



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

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

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:

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

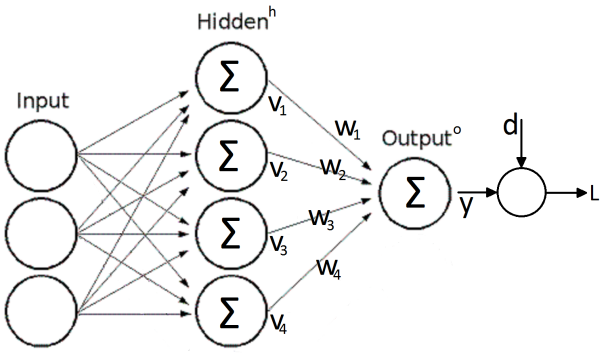


Figure 4.7: The example for showing the details of backpropagation

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

I chose **pybox2d**[1] as the main physics engine to implemente computer simulation. **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

5.1.2 Simulation Details

5.2 Data Generation

5.2.1 XML Restoration

An 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

5.2.3 XML to grid

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

5.3 CNN Training

5.3.1 CNN Structure Design

5.3.2 Traing Configuration

Loss Function

Firstly, we define a filter funtion,

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

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

$$L() \tag{5.2}$$

5.3.3 Training Details

5.4 Simualtion based on Trained Model

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

[3] B. Mirtich, "Rigid body contact: Collision detection to force computation", in *Workshop on Contact Analysis and Simulation, IEEE Intl. Conference on Robotics and Automation*, 1998.

[4] K. Erleben, "Rigid body contact problems using proximal operators", in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, 2017, p. 13.

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

[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.