FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc Thesis

Deep Contact

Accelerating Rigid Body simulations using Convolutional Neural Networks

Lucian Tirca <l.tirca@di.ku.dk>

Supervisor
Kenny Erleben <kenny@di.ku.dk>

Abstract

The goal of this project is to improve the performance of numerical methods used in rigid body simulation with the help of Convolutional Neural Networks (CNNs). This is done by using physics-based simulation to generate a synthetic dataset consisting of images ("heat maps") representing the forces and configurations involved. Afterwards, we train a U-Net which can predict good starting iterates for algorithms that compute the forces at contact points between the rigid bodies. The numerical method used by the simulator is explained, along with the concept of warm starting. The results are not impressive - we are unable to match the performance of the built-in warm starting model, but the neural network architecture is able to solve the intermediary physics problem. The thesis provides interpretations of the results and possible ways of improvement.

Acknowledgements

Firstly, I would like to thank my supervisor Kenny Erleben and my employer Sven Andersen for their patience and continued support throughout this research project. To my supervisor, thank you for introducing me to the research topics you are so passionate about. To my employer, thank you for being flexible and encouraging in times I needed it most.

Secondly, I would like to thank my colleagues Lukas Engedal and Jian Wu for their continuous feedback and inspiring work.

Finally, I would like to thank my family, friends and everyone at Copenhagen University who supported and motivated me throughout my masters programme.

Contents

Chapter 1

Introduction

Synthetic datasets have been successfully used for training self-driving cars (in particular, using realistic computer games for synthetic environments), robots (inside physics simulators) and many other tasks where the data is expensive, difficult to retrieve or too time-consuming to gather.

This thesis explores how this technique can be used to solve the rigid body contact problem and in particular if it is possible to use a toy physics environment to find heuristics that can improve the numerical method called the Linear Complementarity Problem.

We will explain the problem, build a data generation pipeline using a open source physics simulator, transform the data so that it is usable by a Convolutional Neural Network, train the network and use it as an "oracle" to infer solutions to the numerical method.

The performance will then be measured, discussed and suggestions will be made about how to improve them.

Part of the work was done together with Lukas Svarre Engedal and Jian Wu and the source code is available on GitHub [1].

The reader should be familiar with basic Convolutional Neural Network concepts and how they work (backpropagation, activation functions, gradient-based optimization, Batch Normalization) but modeling the physics problem will be explained thoroughly.

Chapters 2 and 3 provide a gentle introduction into the mathematics and physics concepts, building up to how a simulator actually implements the laws of motion and why the Linear Complementarity Problem appears. Inquisitive readers may choose to consult [9] for a more thorough description.

Chapter 4 takes a look at the Smooth Particle Hydrodynamics method, the tool we have chosen to transform the discrete data so that it can be used as input to a Neural Network.

Chapter 5 examines the Deep Learning architecture used, training procedures and the more "non-standard" building blocks that make the CNN fit for the task.

Chapter 6 goes on to describe the methodology (process, software libraries) and implementation choices that were made and provides a review of our experience.

[1] `https://github.com/s0lucien/Deep-Contact`

In Chapter 7 we list our findings and try to analyze them. Our approach to verification is also revealed and we try to justify how we can prove that some parts are correct while pointing out the mistakes found.

Lastly, Chapter 8 goes into a more overall view of the project with improvement suggestions and personal reflections.

Chapter 2

Physics primer

The purpose of this section is to explain the physics in 3D space. For the rest of the project, we use a 2D simulator but it is valuable to review the concepts of the real world.

Before we dive into the numerical methods, we will review how mathematics is used to model the problem of mechanics. We will list the relevant quantities, how they are coupled to each other, which ones change and which ones are constant.

When we idealize the world, we choose an origin and 3 orthogonal unit axes (usually denoted by x,y,z). The origin can be chosen arbitrarily, but does not change. We call this the *world coordinate system* (**WCS**).

Let us start by looking at a single particle. It has a position in space, computed by projecting[B.1] onto the WCS (Fig 2.1).

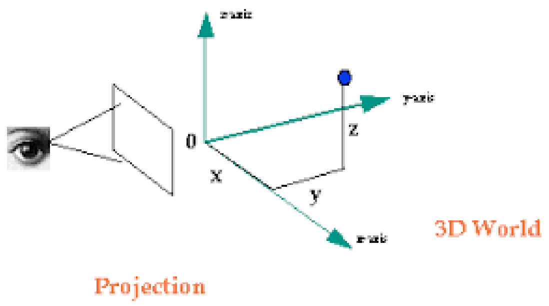


Figure 2.1: Position in the world coordinate system

By measuring the size of the projections, we can assign a 3D vector to the particle's position (denoted as $\mathbf{p}$).

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \tag{2.1}$$

As time goes by, the particle may change its position. The instantaneous change in position in any given time is called velocity ($\mathbf{v}$). This change can also be measured on each one of the unit axes, and written up as a 3D vector:

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{2.2}$$

The mass of the particle is assumed to be constant. For a body comprised of n particles, we can write up the mass as the sum of all particle masses :

$$m = \sum_{i=1}^{n} m_i \tag{2.3}$$

If we assume that all particles have the same mass, we can compute body mass as a product between the density (ρ) and the volume(V).

$$m = \rho V \tag{2.4}$$

Every object is influenced by forces ($\mathbf{f}$), which are also 3D vectors. For every particle, we can compute the resulting force by summing up all forces acting on that specific point. The resulting force leads to acceleration, which modifies the velocity component of the body. We will examine the relationship between force, mass and acceleration later on.

Let us examine some of the forces we use inside our simulator :

Gravity ($\mathbf{f}_g$) is one of the forces. This force is constant (set to 9.81), and it acts upon all non-static bodies. Its direction is downwards on the y axis.

$$\mathbf{f}_g = \begin{pmatrix} 0 \\ -9.81 \\ 0 \end{pmatrix} \tag{2.5}$$

Whenever 2 objects get in contact to each other, a normal force ($\mathbf{f}_n$) acts at the contact point, pushing each of the objects in opposite directions. Aside the normal, we also consider a friction force ($\mathbf{f}_f$), tangent to the normal, and pointing in the direction opposite to the velocity. We will examine this in more detail later on.

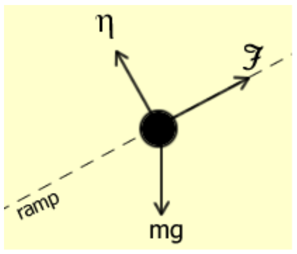


Figure 2.2: Friction and normal between 2 particles

Bodies are comprised of particles infinitely small which act under these rules. We can discretize any larger shape (blob) into particles , by using well-understood geometrical shapes , such as circles, triangles or rectangles - respectively spheres, tetrahedrons or cubes in 3D, to approximate the shape. By controlling the size of particles, we can increase the accuracy of how much the simulation approximates the real world.

The meshing we have leads to rigid bodies. In contrast to particles, bodies also have orientation and a center of mass. The appearance of bodies leads to a new frame of reference: The *body coordinate system* (**BCS**).

For every body, we will compute its center of mass, and set it as the origin of the BCS. It is not necessary to do this, but using the center of mass as the origin of the BCS makes all computations easier.

To transition between the body coordinate system and the world coordinate system, we will first rotate a body, and then translate it. Perhaps now it is a good time to introduce rotations.

Given a point at position $\mathbf{p}_0$ in body space, we can move it by matrix multiplication and addition:

$$\mathbf{p}(t) = \underbrace{\mathbf{R}(t)\mathbf{p}_0}_{\text{Rotate BCS}} + \underbrace{\mathbf{p}_{BCS}(t)}_{\text{Translate}} \tag{2.6}$$

Where $\mathbf{p}_{BCS}$ is the position of the BCS origin in world coordinates, the t signifies the time at which we measure, and $\mathbf{R}$ is a 3×3 matrix with the components:

$$\mathbf{R} = \begin{bmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yx} & r_{zz} \end{bmatrix} \tag{2.7}$$

The first column of this matrix gives the direction that the body's x axis points in measured in WCS, the second column for y and the third column for z.

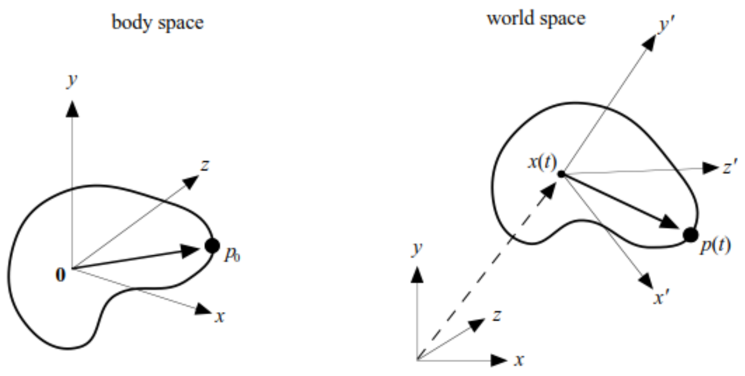


Figure 2.3: Rotation of a point in WCS

We are simulating the trajectory of objects through space, according to the rules we impose. Therefore, we set up the problem as a series of differential equations to couple the quantities together. The laws of motion state that:

- Linear Velocity is the derivative of position with regards to time, measured in meters/second :

$$\mathbf{v}(t) = \dot{\mathbf{p}}(t) = \frac{d}{dt}\mathbf{p}(t) \tag{2.8}$$

- Linear Acceleration is the derivative of velocity with regards to time, measured in meters/second squared :

$$\mathbf{a}(t) = \ddot{\mathbf{p}}(t) = \dot{\mathbf{v}}(t) = \frac{d}{dt}\mathbf{v}(t) \tag{2.9}$$

- Force is in intuitive terms a push or pull applied to a body. Newton's second law states that the acceleration of a body is proportional (with the proportionality constant the mass) to the force exerted on it :

$$\mathbf{f}(t) = m\mathbf{a}(t) \tag{2.10}$$

- For angular velocity (speed of rotation of a body), the relation is not so simple. Given that we cannot differentiate the $\mathbf{R}$ matrix, we will come up with a different measure for the spinning velocity. The first step we can do is select a unit of measure, for which we can use number of spins/second . Any spin can be described by an axis of rotation ω - a vector in 3D space. This vector points from the BCS origin, the direction shows the axis of rotation, while the magnitude measures how many revolutions/second the body is doing.

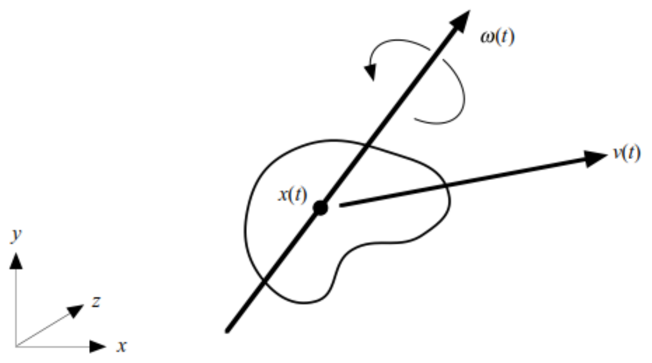


Figure 2.4: Spin vector of a body

We want to relate the rotation matrix $\mathbf{R}$ with the angular velocity ω. At every point in time, the columns of $\mathbf{R}$ are changing, keeping track of to the body. We can start by considering a vector $\mathbf{r}$ inside the body, pointing to some particle and we want to describe how it is moving.

We can decompose $\mathbf{r}$ into a part $\mathbf{a} = Proj_\omega(\mathbf{r})$ and a part $\mathbf{b}$, $\mathbf{r} = \mathbf{a} + \mathbf{b}$. As ω spins, we trace a circle of radius $||\mathbf{b}||$. At every moment, the particle $\mathbf{r}$ will be moving perpendicular to the plane defined by ω and $\mathbf{b}$ and this is the angular velocity $\dot{\mathbf{r}}$ we are looking for.

Observe that $\dot{\mathbf{r}} \perp \omega \perp \mathbf{b}$

Because the tip of $\mathbf{r}$ is moving in a circle of radius $\mathbf{b}$, the magnitude $||\dot{\mathbf{r}}|| = ||\omega|| ||b||$. We also know that the cross product of 2 vectors $\mathbf{x} \times \mathbf{y} = ||\mathbf{x}|| ||\mathbf{y}|| \sin(\theta)$

where θ is the angle between them. ω and $\mathbf{b}$ are orthogonal to each other, so $\sin(\theta) = 1$. Therefore, we can state that for a vector $\mathbf{r}$, its angular velocity is:

$$\dot{\mathbf{r}}(t) = \omega(t) \times \mathbf{b} = \underbrace{\omega(t) \times \mathbf{a}}_{0} + \omega(t) \times \mathbf{b} = \omega(t) \times \mathbf{a} + \mathbf{b} = \omega(t) \times \mathbf{r}(t) \tag{2.11}$$

Where we used the fact that $\mathbf{a} \parallel \omega$ and thus $\sin(\theta) = 0$.

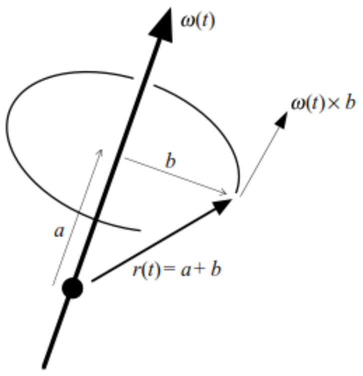


Figure 2.5: Angular velocity decomposition

We can do this for all 3 vectors in the $\mathbf{R}$ matrix:

$$\dot{\mathbf{R}} = \begin{bmatrix} \omega(t) \times \begin{pmatrix} r_{xx} \\ r_{xy} \\ r_{xz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{yx} \\ r_{yy} \\ r_{yz} \end{pmatrix} & \omega(t) \times \begin{pmatrix} r_{zx} \\ r_{zy} \\ r_{zz} \end{pmatrix} \end{bmatrix} \tag{2.12}$$

Even if collision detection is out of scope for this paper, we can take a short review of the naive technique for contact point generation:

Given a plane defined by the normal $\mathbf{n}$ and a point $\mathbf{p}$ on the plane, we can identify whether a particle $\mathbf{x}$ is in contact with the plane by taking the inner product of $\langle \mathbf{p} - \mathbf{x}, \mathbf{n} \rangle$. If the product is positive, we are not in contact. If the product is 0 (the vectors are orthogonal), contact has occurred. When the quantity is negative, the particle has passed through the plane.

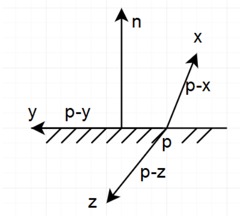


Figure 2.6: Collision detection using inner product

If we consider collision to happen without friction, and assume the bodies are elastic, both the force and the velocity can be decomposed into 2 components : one normal to the contact surface $\mathbf{v}_n = \langle \mathbf{n}, \mathbf{v} \rangle \mathbf{v}$, $\mathbf{f}_n = \langle \mathbf{n}, \mathbf{f} \rangle \mathbf{f}$ and one tangent to the contact plane $\mathbf{v}_t = \mathbf{v} - \mathbf{v}_n$, $\mathbf{f}_t = \mathbf{f} - \mathbf{f}_n$.

After the contact, the normal component of the velocity gets negated and multiplied by a coefficient of restitution $\mathbf{v}_{new} = -r\mathbf{v}_n + \mathbf{v}_t$, $0 \leq r < 1$. Thus, the velocity vector in the normal direction changes sign , and the body is pushed away from the contact point. If the coefficient of restitution is 0, the normal component gets entirely canceled, so the body does not bounce at all. Large values of r make the body act as a very elastic super-ball by conserving a lot of the pre-collision velocity when it bounces.

Let us now take a closer look at the friction problem. Friction is the resistance opposed to sliding between 2 contact surfaces. Coulomb's laws state that the friction force $\mathbf{f}_f$ between 2 bodies is proportional to the normal force of the contact plane $\mathbf{f}_n$. Let us call the proportionality constant μ:

$$\mathbf{f}_f = \mu \mathbf{f}_n \tag{2.13}$$

The coefficient of friction μ is a property of the materials in contact, and is measured empirically. Friction is difficult to model because there are 2 kinds of friction that can appear:

- static friction : When 2 bodies are in static equilibrium, in order to start a sliding motion, we need to apply a pull to one of the bodies. The force required to start the movement is used to calculate the static coefficient of friction $\mu_{static} = \|\mathbf{f}\| / \|\mathbf{n}\|$

- dynamic/kinetic friction: Once the body started to move, the force required to keep it at a constant velocity is different(usually smaller) than the initial force. This leads to a second coefficient of friction $\mu_{dynamic}$ which we can observe by measuring the normal and the pull force.

The consequence of this rule is that if we apply a force at a certain angle ϕ, depending on the value of the angle the body will not move no matter the magnitude of the force. Let us first examine this relation visually in 2D:

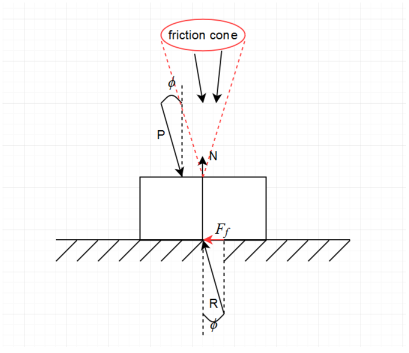


Figure 2.7: Coulomb friction cone

In the figure we have simplified the rules of physics so that only the external force $\mathbf{p}$ is acting on the body at the time. A resulting force $\mathbf{r}$ appears at the contact point, and we can decompose it into the normal $\mathbf{f}_n$ and the friction $\mathbf{f}_f$ components. The geometry of the system tells us that :

$$F_n = R\cos\phi \tag{2.14}$$

$$F_f = R\sin\phi \tag{2.15}$$

$$\tan\phi = F_f/F_n \overset{\text{Coulomb's}}{\underset{\text{law}}{=\!=}} \mu \tag{2.16}$$

Any force applied at an angle less than ϕ will be canceled by the friction force, and the velocity of the body will remain 0 (static friction). Once the angle becomes larger or equal to ϕ, the bodies begin sliding, and dynamic friction occurs. In 3D, this relation can be formalized by:

$$\mathbf{f}_{result} = \{\mathbf{f}_f + \mathbf{f}_n \| \|\mathbf{f}_f\| \leq \mu\|\mathbf{f}_n\|\} \tag{2.17}$$

This force, comprised of both a normal component and a friction component draws a cone with all possible values of the collision response due to friction. The angle of the cone depends on the coefficient of friction μ, higher values leading to a more rounded Coulomb cone.

When we apply a external force to the body, may it be a contact, push or pull, it acts on a specific point of the surface. Aside from the point, the force acts all over the body, and this force is called torque. If the force is applied away from the center of mass, it would introduce some rotation, thus in intuitive terms, torque is the axis the body would spin about if the center of mass were held firmly into place.

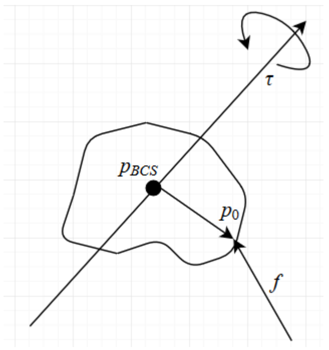


Figure 2.8: Torque vector

We use the cross product to compute it ($\mathbf{p}_{BCS}$ is position of the center of mass of the body):

$$\tau = (\mathbf{p_0} - \mathbf{p}_{BCS}) \times \mathbf{f} \tag{2.18}$$

What happens is that, as in the rotation case, the linear component of the force $\mathbf{f}$ gets canceled, and the orthogonal component of $\mathbf{f}$ to $\mathbf{p_0} - \mathbf{p}_{BCS}$ gives the magnitude and direction of τ.

Another important measure to consider is the momentum. This is the product of the mass and the velocity of an object, and it says something about how difficult is to change the velocity of an object:

$$\mathbf{P} = m\mathbf{v} \tag{2.19}$$

Momentum is important due to the law of conservation: if several particles interact, the momentum exchanged between each pair of particles adds up to zero. Assuming no other forces are acting on bodies with masses m_1 and m_2:

$$m_1\mathbf{v_1}_{old} + m_2\mathbf{v_2}_{old} = m_1\mathbf{v_1}_{new} + m_2\mathbf{v_2}_{new} \tag{2.20}$$

Below we can see a visualization of how linear momentum acts in the collision between two bodies. The red vector represents the initial momentum and the blue ones are the velocities after collision. For simplicity, we assume that only one of the bodies is moving before the contact:

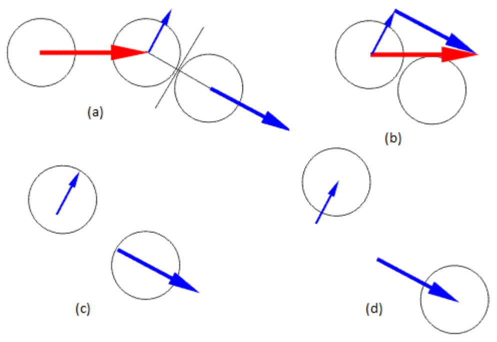


Figure 2.9: (a) the body moves with a certain momentum, until collision appears (b) the momentum is decomposed in the normal and tangential direction for the contact plane (c and d) bodies momentum change, and new velocities can be computed and applied after the contact

Bodies that rotate also possess angular momentum, which is a bit more complicated to model, and has less of an intuitive explanation. In order to explain it, we first have to explain what the inertia tensor is.

The inertia tensor is computed from the geometry of the body, and it represents the extent to which an object resists rotational acceleration about a particular axis. The mass inside the object is distributed depending on the shape and density. This 3×3 matrix is the rotational analogue to mass, and computing it is out of the scope of this paper. It is important to remark that for most geometrical shapes, the quantity has been computed and can be found in tables and specialized literature.

For illustration purposes, here is the inertia tensor for a solid sphere of mass m and radius r:

$$\mathbf{I}_{sphere} = \begin{bmatrix} \frac{2}{3}mr^2 & 0 & 0 \\ 0 & \frac{2}{3}mr^2 & 0 \\ 0 & 0 & \frac{2}{3}mr^2 \end{bmatrix} \tag{2.21}$$

Another very important point to make is that the inertia tensor is computed according to an axis of rotation. If we pre-compute the initial inertia $\mathbf{I}_{body}$ and it is constant across the simulation, we can compute the tensor at some other rotation $\mathbf{R}(t)$:

$$\mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_{body}\mathbf{R}(t)^T \tag{2.22}$$

Therefore, the angular momentum $\mathbf{L}(t)$ is the scaling factor between the angular velocity $\omega(t)$ and the inertia tensor $\mathbf{I}$ and is defined as:

$$\mathbf{L}(t) = \mathbf{I}(t)\omega(t) \tag{2.23}$$

Momentum simplifies the mathematical equations that are used during the simulation precisely due to the conservation rule stated earlier. In conclusion, momentum (angular or linear) is more fit to be used as a state variable for the body as opposed to velocity which varies more often.

Let us take a closer look at rotation and the way it is represented in our simulation. Earlier, we have described a 3×3 matrix that we can use to rotate a body around a chosen axis, but in this section we will prove that it is actually possible to represent the same concept using 4D vectors normalized to unit length called quaternions. In contrast with the rotation matrix, which uses 9 parameters to explain the 3 degrees of freedom, quaternions use only 4 and therefore reduce redundancy. This also makes them less sensitive to numerical drift.

Let us analyze these numbers to see how we can operate with them. As we did with 3D space, we will first designate 4 dimensions. If we have a 4D vector $q = [a, b, c, d]$, we will leave the first coefficient a be a real number (called the scalar) and b, c, d can represent coefficients for dimensions with unit vectors $\vec{i}, \vec{j}, \vec{k}$ (called the vector part). Therefore $q = [a, b\vec{i}, c\vec{j}, d\vec{k}]$.

Addition for quaternions works in the same way as it would with vectors: Given $q_1 = [a, b\vec{i}, c\vec{j}, d\vec{k}]$ and $q_2 = [e, f\vec{i}, g\vec{j}, h\vec{k}]$,
$q_1 + q_2 = [a + e, (b + f)\vec{i}, (c + g)\vec{j}, (d + h)\vec{k}]$.

When multiplying these vectors, we will use a very important property that the unit vectors i, j, k possess:

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.24}$$

Using this property, we can generate the following multiplication table:

$\cdot$	i	j	k
i	-1	k	$-j$
j	$-k$	-1	i
k	j	$-i$	-1

Therefore, if we multiply q_1 and q_2:

$$\begin{aligned} q_1q_2 = ae + af\vec{i} + ag\vec{j} + ah\vec{k} + \\ be\vec{i} + bf\vec{i}^2 + bg\vec{i}\vec{j} + bh\vec{i}\vec{k} + \\ ce\vec{j} + cf\vec{i}\vec{j} + cg\vec{j}^2 + ch\vec{j}\vec{k} + \\ de\vec{k} + df\vec{k}\vec{i} + dg\vec{k}\vec{j} + dh\vec{k}^2 \end{aligned}$$

By using the multiplication table and factoring out the unit vector directions, we can simplify the product into:

$$q_1q_2 = \begin{bmatrix} ae - bf - cg - dh \\ (af + be + ch - dg)\vec{i} \\ (ag - bh + ce + df)\vec{j} \\ (ah + bg - cf + de)\vec{k} \end{bmatrix} \tag{2.25}$$

Even if it not obvious, we can further prove the relation between quaternion product , dot and cross product. Given quaternions $q_1 = [w_1, \vec{v_1}]$, $q_2 = [w_2, \vec{v_2}]$:

$$q_1q_2 = [w_1, \vec{v_1}] \cdot [w_2, \vec{v_2}] = [w_1w_2 - \vec{v_1} \cdot \vec{v_2}, w_1\vec{v_2} + w_2\vec{v_1} + \vec{v_1} \times \vec{v_2}] \tag{2.26}$$

We can prove this by rewriting the matrix from 2.25 as

$$[a, \underbrace{b\vec{i}, c\vec{j}, d\vec{k}}_{\vec{x}}] \cdot [e, \underbrace{f\vec{i}, g\vec{j}, h\vec{k}}_{\vec{y}}] = \begin{bmatrix} ae - (bf + cg + dh) \\ (af + be + ch - dg)\vec{i} \\ (ag + ce + df - bh)\vec{j} \\ (ah + de + bg - cf)\vec{k} \end{bmatrix} = \begin{bmatrix} ae - \vec{x} \cdot \vec{y} \\ a\vec{y} + e\vec{x} + \vec{x} \times \vec{y} \end{bmatrix}$$

A rotation if θ radians about a unit axis u us represented by the unit quaternion $[\cos(\theta/2), \sin(\theta/2)u]$

Given a quaternion $q = [a, b\vec{i}, c\vec{j}, d\vec{k}]$ we can convert it to a rotation matrix using the following rule:

$$R(q) = \begin{bmatrix} 1 - 2c^2 - 2d^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 1 - 2b^2 - 2d^2 & 2bd - 2ac \\ 2bd - 2ac & 2cd + 2ab & 1 - 2b^2 - 2c^2 \end{bmatrix}$$

Chapter 3

Rigid Body Dynamics - What happens inside a simulator ?

In this section we come closer to understanding how a rigid body simulator is implemented and how it works. After reviewing the laws of mechanics, it is possible to start painting a clearer picture of how the simulation takes place.

In our simulation we need to know the state of the system at regular intervals of time, so that we can render a frame of graphics. We have previously explored the force-acceleration model, but instead of it, we will now introduce the impulse-velocity model.

The impulse λ is, in intuitive terms, the area under a force-time graph, the product between force and the time it gets applied.

$$\lambda = \int \mathbf{f} dt \tag{3.1}$$

Applying an impulse is equivalent to a immediate change of momentum , which leads to immediate changes in velocity (because masses are constant).

Let's take a look at what it means to apply an impulse. If we have a body and apply an impulse to a point on its surface, we can calculate the equivalent rigid body impulse (with linear and angular components), and change the velocity accordingly, as seen in the figure :

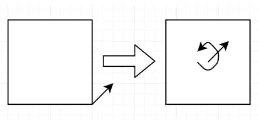


Figure 3.1: Impulse at point applied to body

In a simulation, we prevent bodies from penetrating each other by applying impulses to change their velocities. We want to eliminate the component of the velocity that is pulling bodies into penetration. Given a velocity direction and magnitude, we can project it onto the normal of a contact point between the bodies to get the penetration direction.

At every time step, we can calculate the relative velocity at the contact point, get the magnitude in the normal direction, then calculate the impulse needed to be applied at the point in order to make the relative velocity 0. This process of eliminating velocity in the normal direction is also called solving the contact problem.

We can then calculate this impulse in rigid body coordinates, and sum up all impulses (from all contacts) acting on a body to get the final impulse applied. From this impulse we can compute the new velocity and apply the velocity to the position in order to update the simulation state (and possibly re-draw the body in the new pose).

Let us now start to formalize all this information using more rigorous mathematical notation, and see how it can be implemented.

The pose of a body can be expressed as a 7D vector, with 3 coordinates for the location of the center of mass in WCS and a quaternion part for rotation:

$$\mathbf{x} = \begin{bmatrix} \underbrace{p_x \quad p_y \quad p_z}_{location} & \underbrace{a \quad b \quad c \quad d}_{orientation} \end{bmatrix} \tag{3.2}$$

The velocity can be expressed as a 6D vector, with linear and angular components:

$$\mathbf{v} = \begin{bmatrix} \underbrace{v_x \quad v_y \quad v_z}_{linear} & \underbrace{r_i \quad r_j \quad r_k}_{angular} \end{bmatrix} \tag{3.3}$$

For constant force applied to a body, the impulse is:

$$I = \mathbf{f}\Delta t \tag{3.4}$$

Applying a force changes acceleration, but applying an impulse changes velocity directly (since $v = a\Delta t$). Therefore, the Rigid Body Impulse has both linear and angular components as well:

$$\mathbf{I} = \begin{bmatrix} \underbrace{\lambda_x \quad \lambda_y \quad \lambda_z}_{linear} & \underbrace{\lambda_i \quad \lambda_j \quad \lambda_k}_{angular} \end{bmatrix} \tag{3.5}$$

In order to apply the 3D angular velocity to the 4D orientation quaternion, we use the following formulas:

$$\theta = \|\mathbf{v}_{ang}\Delta t\| \tag{3.6}$$

$$\mathbf{a} = \mathbf{v}_{ang}/\|\mathbf{v}_{ang}\| \quad \text{Normalized Angular Velocity (3D)} \tag{3.7}$$

$$\mathbf{q}_{new} = \begin{bmatrix} \cos(\frac{\theta}{2}) & \mathbf{a}\sin(\frac{\theta}{2}) \end{bmatrix} \mathbf{q}_{old} \quad \text{Element-wise 4D multiplication} \tag{3.8}$$

The position update uses the linear velocity, as expected:

$$\mathbf{p}_{new} = \mathbf{p}_{old} + \mathbf{v}_{lin}\Delta t \tag{3.9}$$

For the velocity update, we use the facts $f = ma \implies i = ma\Delta t$, $\Delta v = a\Delta t \implies v = i/m$ to write it in matrix form:

$$\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{M}^{-1}\mathbf{I} \tag{3.10}$$

Where M is the 6×6 matrix:

$$\mathbf{M} = \begin{bmatrix} m & & & & & \\ & m & & & & \\ & & m & & & \\ & & & . & . & . \\ & & & . & . & . \\ & & & . & . & . \end{bmatrix}$$

Where the lower right 3×3 part is the inertia tensor.

If we now know how rigid bodies behave we can examine what happens at the contacts between them. Whenever two bodies collide, the collision detection algorithm will generate a contact point which has position in 3D space and a normal direction - a 3D unit vector(for every body, the normal points in the direction opposite to the surface it is in contact with) as can be seen in fig. 3.2.

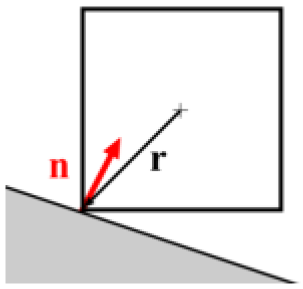


Figure 3.2: Contact normal and offset of contact point from the center of mass

The contact leads to both linear and angular interactions. For every one of these points, we can construct the contact Jacobian, by using the normal for the linear component and the offset of contact point from the center of mass $\mathbf{r}$ for the angular one (as shown in Chapter 2, the cross product is used):

$$\mathbf{j} = \left[\underbrace{\mathbf{n}}_{linear} \quad \underbrace{\mathbf{r} \times \mathbf{n}}_{angular} \right] \tag{3.11}$$

The Jacobian is used to project velocities from the body to the contact point (υ is the relative velocity at the contact point). It is a scalar that defines the magnitude, as the normal $\mathbf{n}$ defines the direction:

$$\upsilon_{rel} = \mathbf{j}\mathbf{v} \tag{3.12}$$

Or from project contact point impulses (λ scalar) to the body. $\mathbf{I}$ is a 6D vector which is the Jacobian scaled by λ:

$$\mathbf{I} = \mathbf{j}^T\lambda \tag{3.13}$$

We can now rewrite the final body velocity in terms of the contact impulse:

$$\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{M}^{-1}\mathbf{j}^T\lambda \implies \tag{3.14}$$

And turn it into relative velocity:

$$\underbrace{\mathbf{j}\mathbf{v}_{new}}_{v_{rel}} = \mathbf{j}\mathbf{v}_{old} + \mathbf{j}\mathbf{M}^{-1}\mathbf{j}^T\lambda \tag{3.15}$$

This allows us to derive the condition to make the relative velocity 0:

$$\lambda = -(\mathbf{j}\mathbf{M}^{-1}\mathbf{j}^T)^{-1}\mathbf{j}\mathbf{v}_{old} \tag{3.16}$$

When we have more than one contact point, applying am impulse to one of them can affect the velocity at many other contact points. For every contact we either have a stick or a slip (the relative is either 0 and impulse is positive or the impulse is 0 therefore the bodies are moving further away from each other). Formally:

$$v_{rel} \geq 0 \perp \lambda \geq 0 \tag{3.17}$$

If we have multiple contacts with $\mathbf{j}_1, \mathbf{j}_2, ...$ for the same body, we can use the same rules to set up the matrix equation that tells us all relative velocities:

$$\begin{bmatrix} v_{rel_1} \\ v_{rel_2} \end{bmatrix} = \begin{bmatrix} \mathbf{n}_1 & \mathbf{n}_1 \times \mathbf{r}_1 \\ \mathbf{n}_2 & \mathbf{n}_2 \times \mathbf{r}_2 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_i \\ v_j \\ v_k \end{bmatrix} \leftrightarrow \mathbf{v}_{rel} = \mathbf{J}\mathbf{v} \tag{3.18}$$

If we solve the system as a linear system of equations, we will end up with negative relative velocities, leading the bodies into interpenetration. Therefore, we need to set another set of rules for this, called the Signorini conditions:

$$v_{rel_i} \geq 0 \quad \forall v_{rel_i} \in \mathbf{v}_{rel} \quad \text{Every relative velocity should be zero or separating} \tag{3.19}$$

$$\lambda_i \geq 0 \quad \forall \lambda_i \in \boldsymbol{\lambda} \quad \text{Every contact impulse should be non-attractive} \tag{3.20}$$

The final model can then be summarized as, for every body with n contacts:

$$f = ma \implies \mathbf{M}\ddot{\mathbf{x}} = \underbrace{\mathbf{J}^T}_{6\times n}\underbrace{\boldsymbol{\lambda}}_{n\times 1} + \underbrace{\mathbf{f}_e}_{\text{External Forces, 6D}} \tag{3.21}$$

$$\dot{\mathbf{x}} = \mathbf{v} \tag{3.22}$$

$$\boldsymbol{\lambda}_i \geq 0 \perp (\mathbf{J}\mathbf{v})_i \geq 0 \qquad \forall i \in (1, n) \tag{3.23}$$

If we introduce time-stepping by Δt instead of derivatives:

$$\mathbf{M}(\mathbf{v}_{new} - \mathbf{v}_{old}) = \mathbf{J}^T\boldsymbol{\lambda} + \mathbf{f}_e\Delta t \tag{3.24}$$

$$\mathbf{x}_{new} - \mathbf{x}_{old} = \mathbf{v}_{new}\Delta t \tag{3.25}$$

$$\boldsymbol{\lambda}_i \geq 0 \perp (\mathbf{J}\mathbf{v}_{new})_i \geq 0 \qquad \forall i \in (1, n) \tag{3.26}$$

And it leads to update rules:

$$\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{M}^{-1}\mathbf{J}^T\boldsymbol{\lambda} + \mathbf{f}_e\Delta t \tag{3.27}$$

$$\mathbf{x}_{new} = \mathbf{x}_{old} + \mathbf{v}_{new}\Delta t \tag{3.28}$$

Where we get $\boldsymbol{\lambda}$ by solving the Linear Complementarity Problem (**LCP**):

$$\boldsymbol{\lambda} = LCP(\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T, \mathbf{J}(\mathbf{v}_{old} + \mathbf{f}_e)) \tag{3.29}$$

This is simply the solution from 3.15 and 3.16, with the added constraint that at least one of the terms has to be 0.

In this idealized model friction forces were not applied but using the knowledge gained from the previous chapter it is possible to add them as part of the external forces (of course, the Coulomb friction cone leads to its own formulation of the LCP). An example of a more complex model that includes friction is given in [8], but we have chosen to keep it simple (and unrealistic) in order to get a better grasp of the basics.

We will not go into too much detail with how the LCP is solved, but it is important to give a basic introduction about the method. This LCP is considered a hard combinatorial optimization problem.

Most solvers start from guessing a solution (they would choose some indexes and set λ to 0 in our case). If one knows which impulses are 0, it is possible to find an approximate solution. If the solution is not feasible,the algorithm will choose another set of impulses to set to 0 (usually keeping some of the old "good" ones and adding others to this set). This iterative process ends when a solution is found.

We can see here that the starting state of the LCP is very important to the speed of convergence. In a hypothetical scenario, if we had a "oracle" that would tell us which impulses to set to 0, the LCP would be very easy to solve. A good starting state is what we call a "warm start" and this is what we are trying to predict using deep learning and smooth particle hydrodynamics methods.

Chapter 4

Smooth Particle Hydrodynamics

The chosen simulator will tell us position in 2D space of the bodies, contacts and physical attributes of these items at every point in time. These are numerical values, but in order to use CNNs, we need to transform this information into multi-channel "heatmaps" where we can use the structure of the body formation to our advantage.

The choice of tool in this case was to use Smooth Particle Hydrodynamics methods to create images (grids) that can spacially represent the world configuration. We would then be able to come back to the discrete formulation of the problem (with some approximation error) by querying these grids using bilinear interpolation.

4.1 Particle to Grid

The grid is a 2-dimensional collection of evenly spaced points according to some x-resolution and y-resolution. In our case, we chose to keep things simple and set the resolutions equal to one another forming a grid of squares, but the formulation would work with different resolutions for rows and columns that would lead to rectangular patches.These grids can store any physical attributes of the simulation.

We want to transfer the attributes at some point to the grid, according to a function. While some methods may choose to place the attribute in the nearest neighboring grid node, we have chosen to use SPH methods to generate the grids.

Originally developed to solve problems in astrophysics [], this method has found its way into fluid simulation and other fields. Our hypothesis is that it may also be applied for Rigid Body Dynamics simulations.

In intuitive terms, the idea is to create a (regular) grid of coefficients which will be used to "splatter" the values of the particle attributes onto the grid in its vicinity, according to the distance of the grid point to the particle location. The coefficients are then used to multiply the value of the attribute in order to compute the heatmap for a specific channel.

These coefficients are the results of a function that relevant literature calls the smoothing kernel (denoted by W). The kernel takes into account the distance between the current grid point and the particle (radius r) and returns a weight between 0 and 1. It is important that the weights sum to 1, so that no value is either gained or lost when transferring from particle to grid:

$$\int W(\mathbf{r})d\mathbf{r} = 1$$

The kernels will only consider grid points inside a certain maximum radius from the particle, and we call this value the support radius (denoted by h).

4.2 Smoothing Kernels

Let us now examine how a smoothing kernel function looks like starting from a familiar point - the Gaussian function:

$$W_{Gauss}(\mathbf{r}, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{|r|^2}{2\sigma^2}}$$

Where σ denotes the standard deviation. This distribution is usually the first choice when working with natural data (it is also called the normal distribution for this reason), also for its mathematical properties(the derivative of a Gaussian distribution is also a Gaussian). However, this kernel has bad numerical properties due to its unlimited range: for every particle, it would set (very small) coefficients all over the grid.

A better choice is the Spiky kernel, defined by:

$$W_{Spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - \|\mathbf{r}\|)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & otherwise \end{cases}$$

The fact that it only acts in a specific range makes it more computationally interesting. This kernel (as its name suggests), focuses most of the weight above the particle, and quickly decreases afterwards. The problem is that the particle may not be exactly on a grid node, therefore the maximum "tip" of the spike would be lost in the discretization.

We need a kernel that is more robust against discretization, and has the same properties as the Gaussian. The literature provides an answer to this with the Poly6 function:

$$W_{poly6}(\mathbf{r}, h) = \frac{4}{\pi h^8} \begin{cases} (h^2 - \|\mathbf{r}\|^2)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & otherwise \end{cases}$$

As we can also see in the comparison plot 4.1, the Poly6 kernel provides a similar shape to the Normal distribution, and has a formulation that is easier to compute.

Let us take a look at the algorithm 1 pseudo-code for creating such grid of coefficients. It is important to notice that we are using the k-d tree [4] data structure for performing fast range queries.

At the end of this procedure, the grid is filled with lists of tuples containing the particle id and the coefficient associated with it. Afterwards, it is possible to look up the physical attribute for the particle id, multiply it with the coefficient and sum up all these values for every grid node in order to generate the final heatmap.

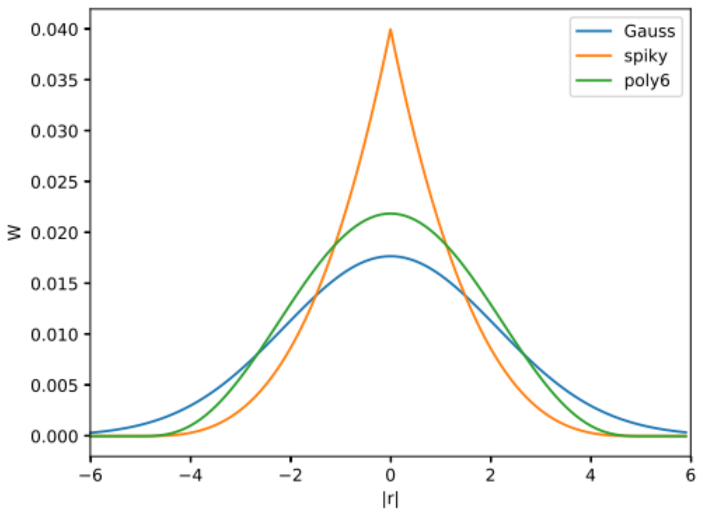


Figure 4.1: Coefficients for the 3 different kernels, with $h = 5$ and $\sigma = 3$ for the Gaussian

Algorithm 1: Compute grid coefficients for a single particle

```
$tree \leftarrow KDTree(grid)$;
$neighbors \leftarrow tree.Query(particle\_position, support\_radius)$;
for $neighbor \in neighbors$ do
    $r \leftarrow norm(neighbor - particle\_position)$;
    $coefficient \leftarrow W(r, support\_radius)$;
    $grid[neighbor] \leftarrow (particle\_id, coefficient)$;
end for
```

4.3 Querying the grid

Now for the other perspective : How to recover values of our attributes if we have a grid?

Our choice was to use bilinear interpolation. This could in principle be replaced by nearest neighbor or bicubic interpolation and have usable values.

After going from particle to grid and then back to particles it is possible to study the grid transfer error, where X is the original attribute value and $\hat{X}$ is the approximation, and $\mathbf{p}$ is the position of the particle:

$$\mathbf{G} = W(\mathbf{p}, h, x_{res}, y_{res})X$$
$$\hat{X} = interp(\mathbf{p}, \mathbf{G})$$
$$\xi_W = loss(X, \hat{X})$$

A very important observation is that we can find a minimum if we set the grid resolution and support radius to be infinitely small, therefore there is a need for regularization. The loss could be the mean squared or mean absolute error.

If the loss is the mean squared error:

$$\xi_W(h, x_{res}, y_{res}) = (\hat{X} - X)^2 = (interp(\mathbf{p}, W(\mathbf{p}, h, x_{res}, y_{res})X) - X)^2 + C$$

We could perform try to perform gradient-based optimization by taking $\mathbf{p}$ and X as constants, setting some C, computing the derivative and finding the support radius and resolution, but grid search is a better alternative. This is because we can observe that the resolution and support radius influence how many coefficients act on the same grid point (thus influencing how many X are summed up for a grid point). Therefore even if the gradient of this error function can be computed, the size of the computation at every step of the minimization can vary with every perturbation.

Chapter 5

Deep Learning

Here we review the deep learning theory that one needs in order to understand our model and the learning process. We assume that the reader is familiar with the basics of neural networks such as backpropagation, sigmoid activation functions and convolution as well as loss functions and optimization based on gradient descent. We will try to present the more novel techniques and theories.

5.1 Learning Rate Finder

One of the main problems of Machine Learning has always been hyperparameter selection. Methods such as grid search have been developed and (successfully) used to select the different values for the parameters of the models. However, these methods still suffer from the fact that the search space is limited to the settings chosen by the algorithm user.

In the case of neural networks, one of the most important values to be set is the Learning Rate. This value specifies how much of the gradient information is used at every forward-backward pass through the network. A value that is too high will send the optimization step too far away from the optimum, effectively "shooting out" into settings which are unstable and will eventually meet numerical failure. Very small values will take steps which are too small, and increase training time until reaching an optimum becomes unfeasible.

Although adaptive optimizers such as ADAM [13] have been developed, optimization is still sensitive to the initial value of the learning rate. Moreover, recent studies [12] [19] have shown the superiority of the optimal value found by using simple SGD (Stochastic Gradient Descent).

In 2017, Leslie Smith [18] has proposed a way of searching for learning rates by using either a linear or exponential schedule (proposed by Jeremy Howard and implemented in the **`fast.ai`** framework).

There is no universal optimal learning rate, as it depends on the scale of the the data. An ideal learning rate for a problem instance is one which yields significant decrease in the loss function. The systematic approach is to gradually increase the value after each mini-batch, record the loss accrued and examine the loss plot at the end of the procedure.

By examining the loss vs learning rate plot, it is possible to find a good value for the LR in the region where the loss is decreasing drastically. A rule of

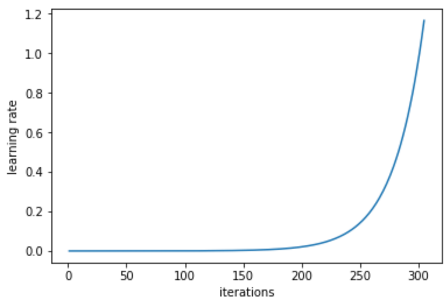


Figure 5.1: Exponential learning rate schedule

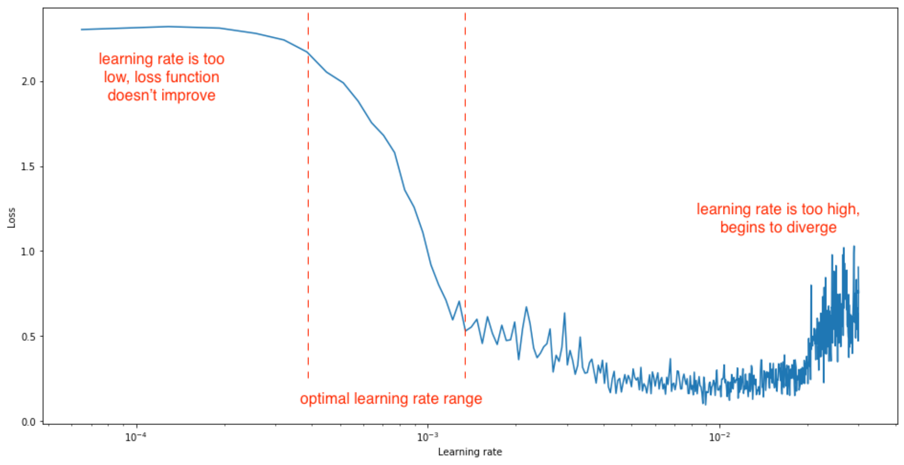


Figure 5.2: Image by Jeremy Jordan via https://www.jeremyjordan.me/nn-learning-rate/

thumb is to set the LR to the maximum value where the loss is still decreasing, and not becoming "ragged" (a sign of instability).

5.2 Training Schedule

When training neural networks, the model "explores" the (very high-dimensional) loss space in search of a minimum. It can be assumed that when setting random weights (in the initialization), any step in the direction opposite to the gradient will lead to a lower loss. Therefore, it would be good to start out with a learning rate that is large and will allow us to move quickly across the "easy part". Once we get close to a local minimum, it becomes easier to overshoot, thus a lower learning rate will allow us to "squeeze" more information from the data.

In general, it can be assumed that the network will not get stuck in a local minimum due to the stochastic nature of the training process: either resampling the batches at the beginning of each epoch or simply changing their order (and even just repeating them in the same order) will ensure that local

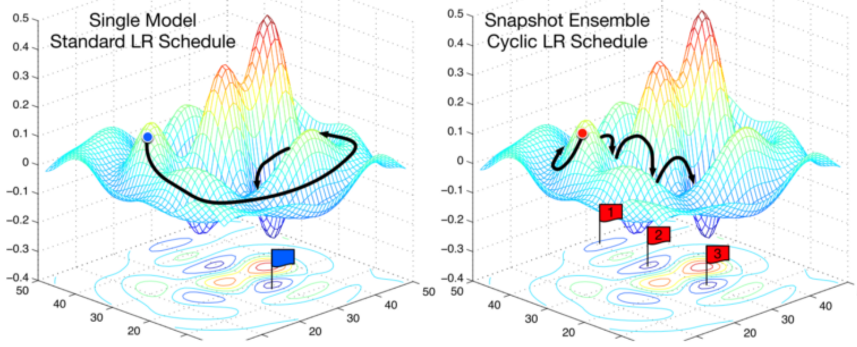


Figure 5.3: Image from paper Snapshot Ensembles [10]

minima are avoided. Aside from these assumptions, the paper [15] suggests another technique : restarting the training process with the max learning rate, followed by gradually decreasing (annealing) it according to a schedule. The technique is called Stochastic Gradient Descent with Restarts (SGDR).

The idea of SGDR is that, by increasing the LR according to a schedule (in other words restarting the training), the NN will be forced to find another local minimum if the current one is not robust and lead to better generalization performance. To make an analogy, the procedure is a bit like "bumping a pinball table" when the ball is stuck in a position for a longer time.

Leslie Smith proposes triangular learning rates, and we have also experimented with sinusoidal (cosine annealing). The number of cycles can vary, and it should be possible to control how "wide" the schedule is by setting the number of iterations per cycle. Examples of this schedule can be seen in the figures below:

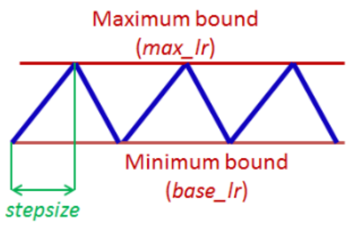


Figure 5.4: Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter stepsize is the number of iterations in half a cycle. Image and caption taken from [18].

5.3 Dilated Convolutions

A typical discrete convolution involve a $n \times n$ (where n is odd) sized kernel K which the network will learn through back-propagation. The idea is that by

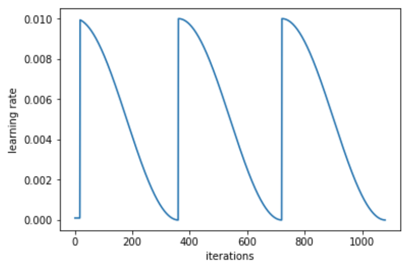


Figure 5.5: Cosine training schedule with max_lr set to 0.01 and stepsize 180 iterations

looking at a patch of the image, the network will learn spacial importance of adjacent pixels, and assign a coefficient to each of the pixels.

The mathematical operation of convolution (denoted by $*$) on a 2D image I is defined as [3]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

In other words, is the operation of multiplying every pixel in the neighborhood around the point at coordinates i, j with a coefficient from the 2D matrix K. These are all summed together, and the result is stored in the result S at the same i, j coordinates.

In the neural network, these results are then passed through a activation function (also called a sigmoid gate) which will decide if the signal will pass to the next layer.

The convolution operation happens in multiple layers, and many kernels are learned throughout the training process. We call the number of convolutions acting on a single area the depth of the kernel. In an intuitive sense, the kernels each learn to look for a certain (rectangular) pattern, and assign a weight to each of the pixel in the image I. By running the convolution operation through all the layers of the network and "dropping" the signal from some of the pixels at every stage (at the pass through the sigmoid), the network eventually manages to "look" at every pixel in the input in the later layers, and try a very large number of possible combinations in order to make a prediction.

Our problem is one of Dense Prediction : We are trying to predict values of the impulse at the grid points in our 2D images. While in Semantic Segmentation, the model looking for labels of a mask that we can overlay on the original image, in our case the values are numeric. We phrase the problem as a regression problem, but we need to combine multi-scale results(look at a larger area).

Based on promising results from Semantic Segmentation [20], we can further improve the search space of the network, and this is where we introduce the dilated convolution.

In D-dilated convolution, usually the stride is 1, but we can increase the stride if we want spatial information from a larger area:

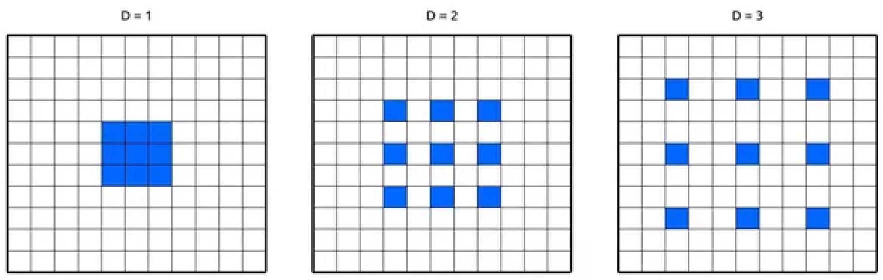


Figure 5.6: https://www.quora.com/What-is-the-difference-between-dilated-convolution-and-convolution+stride

This convolution has the same effect as using a larger kernel size, and setting some of the weights to 0, leaving the corners to have nonzero weights. In theory, this approach allows us to aggregate multi-scale contextual information, and as will be shown in subsequent chapters, it worked very well.

5.4 Downsampling

If we want to use a scale-space technique, we want to compress information in a $n \times n$ (where n is divisible by some k) into a $n/k \times n/k$ representation. The idea of down-sampling a 2D image is to combine $k \times k$ non-overlapping patches of the image into a single pixel. This can be done by either selecting the maximum value of the patch or computing the average. The advantage of average pooling is that it allows gradients to flow backwards through the network into all the input pixels. The disadvantage, is that it increases training time due to increase in computation.

A take-away from this is that max-pooling would be better suited for classification tasks, while average pooling is good for regression tasks, where precision is very important.

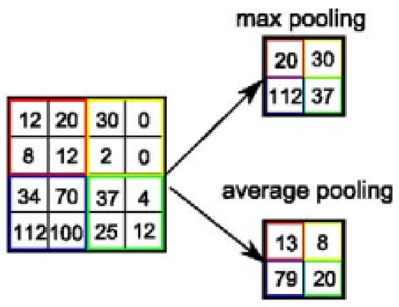


Figure 5.7: Example of average and max pooling for 2×2 patches

5.5 Upsampling

Sometimes, we need to uncompress a low resolution representation. In the up-sampling step, we use the Transpose Convolution operation (sometimes also called Deconvolution). This step finds the weights of a patch which is multiplied with every input pixel and the resulting patches are summed. At the border input is zero-padded so that the an output of a specified size can be reconstructed.

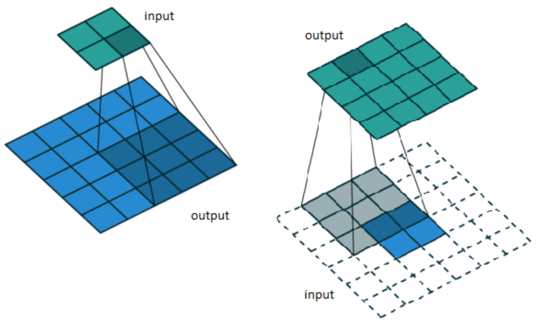


Figure 5.8: Transpose convolution without padding (left) and with zero padding(right)

An important point to add is that, due to the periodic nature of the patch (we are essentially adding together the same large "brush", scaled by the value of the low resolution pixel), this simple upsampling step will result in heavy checkerboard patterns. There are techniques to eliminate these artifacts and the term used is super-resolution. Many of them use resize using nearest neighbor or bilinear interpolation followed by convolution, and they are detailed in [16]. However, for the purpose of this thesis the Transpose Convolution worked fine.

5.6 U-Net Architecture

Let us now look at how all the deep learning blocks come together into a dense prediction model. Our problem is taking multi-channel input of some height and width (to simplify we assume square $n \times n$), and returning multi-channel output of the same shape. Each of the channels in the input and output represents a 2D heat-map of the physical properties involved in the simulation (mass, angular/linear velocity, contact normal/tangent magnitude, normal/tangent impulse, etc).

As seen in the earlier chapters, the physics simulator takes these properties into account, along with spatial information in order to predict the subsequent frame: for every position in the input, we want to find the corresponding value of the position in the output. Therefore, we turned our attention to semantic segmentation where the task is to predict a label for every pixel in the input (in essence to classify every pixel). Instead of classification we have regression and use Mean Square Error as a loss function. The hypothesis is that the same kind of network layout would work.

The best candidate that made sense (both in terms of simplicity and compatibility) for the problem was the U-Net [17]. Originally used for biomedical image segmentation, it has quickly found its way into all tasks involving dense prediction (a very good example is as part of a Generative Adversarial Network as a generator, with outstanding results in black and white image colorization [1]). U-Nets have also showed promising results for Semantic Segmentation [11] on the Carvana [1] image masking challenge.

The idea of this kind of network is to use a contracting path and a symmetric expanding path, with multiple levels of resolution. The input is contracted by using downsampling 5.4, and at the lowest level (where the information is the most condensed), dilated convolutions 5.3 are applied in order to capture as much context as possible. The result of these operations is then summed, and upsampled 5.5 multiple times back to the original shape. At every scale, convolution followed by an activation function is performed before the upsampling or downsampling operations.

At every level of this pyramid, skip connections are used between same-sized resolution. These essentially copy the output of the pre-downsample and concatenate it to the post-upsample stage. They are important as they allow the neurons to "choose" to ignore any lower levels and simply allow information to pass through untampered (effectively learning the identity mapping). Knowing this fact, we can assume that adding more levels to a U-Net is better, since very compressed paths which yield no information will start to be ignored as the training progresses. However, it is good to take into account that training time is proportional to the number of parameters so a sensible number of scales should be used.

The architecture thus functions as a encoder/decoder with skip connections and dilated convolutions at the most compressed level, and an conceptual model is shown below.

Given the shape of the contracting path, it is important to select the input size such that all pooling operations are applied to a layer with an even x and y size (therefore, both dimensions have to be divisible by $2^{\#downsample}$).

The exact architecture and implementation details are explained in chapter 6.4.

[1] `https://www.kaggle.com/c/carvana-image-masking-challenge` Kaggle challenge where the goal is automatically removing background from pictures of cars

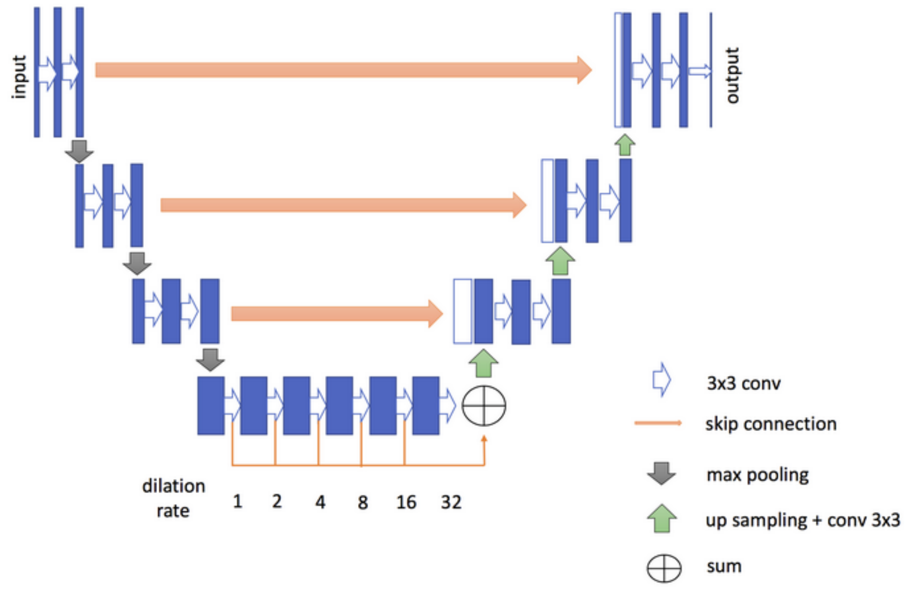


Figure 5.9: An instance of a U-Net

Chapter 6

Implementation

6.1 The Generation-Train-Evaluate Pipeline

Let us start by reviewing the process used in the writing of this thesis. We will first give an overview of the pipeline and try to categorize the steps (by attempting a correlation with Computer Science terms) into states, variables and functions, data and measures.

We can first look at what the objective is : accelerating rigid body dynamics simulations. Therefore, the initial starting point is a 2D simulation where we control the initial entropy.

In order to simplify the problem we have chosen to use a simple world structure: circular balls of the same mass falling inside a square box where the initial state is chosen by placing a mean point inside the 2D space, and sampling a 2D Gaussian placed at that point (more about this in the Results section 7.1). Nonetheless, it should be possible to diversify the simulation layout with different shapes and sizes for worlds and bodies.

These bodies fall inside the box under the effect of gravity until they reach a state of equilibrium (entropy gets eliminated). At every step, the simulator uses a LCP solver to compute contacts, and both body and contact attributes are registered and saved into XML documents. We choose to start saving the files only when the contacts start forming (ignoring the first few frames where bodies fall under the effect of gravity).

The next step is to process the data using SPH techniques to generate multi-dimensional tensors that describe the magnitudes of attributes for bodies and contacts. At this stage, we have to choose the first set of parameters: grid size, support radius and kernel so that we minimize the interpolation error. For every frame in every simulation, we will run this procedure and save results into a HDF5 "blob" file.

The multi-channel "heatmaps" are used to train a Neural Network where the body attributes are inputs, and the contact attributes are outputs. The dataset is split into train/test/validation. At this stage, we can change the architecture (we chose U-Net), the learning rate and the loss function (we used Mean Squared Error, but Mean Absolute Error or other functions that also add regularization can be used). The goal is to minimize the Test Loss.

The model is then used as a warm start generation tool for the LCP solving a new simulation, and performance metrics are extracted from the simulator.

Here, we can vary the world structure, and measuring the number of iterations until convergence, the convergence rates or the error norm. This Deep Learning model is compared with the built-in model for generating warm starts (that uses temporal coherence) , as well as with a model that predicts random warm starts.

The flow is described by figure 6.1, where the color coding shows the different types of components.

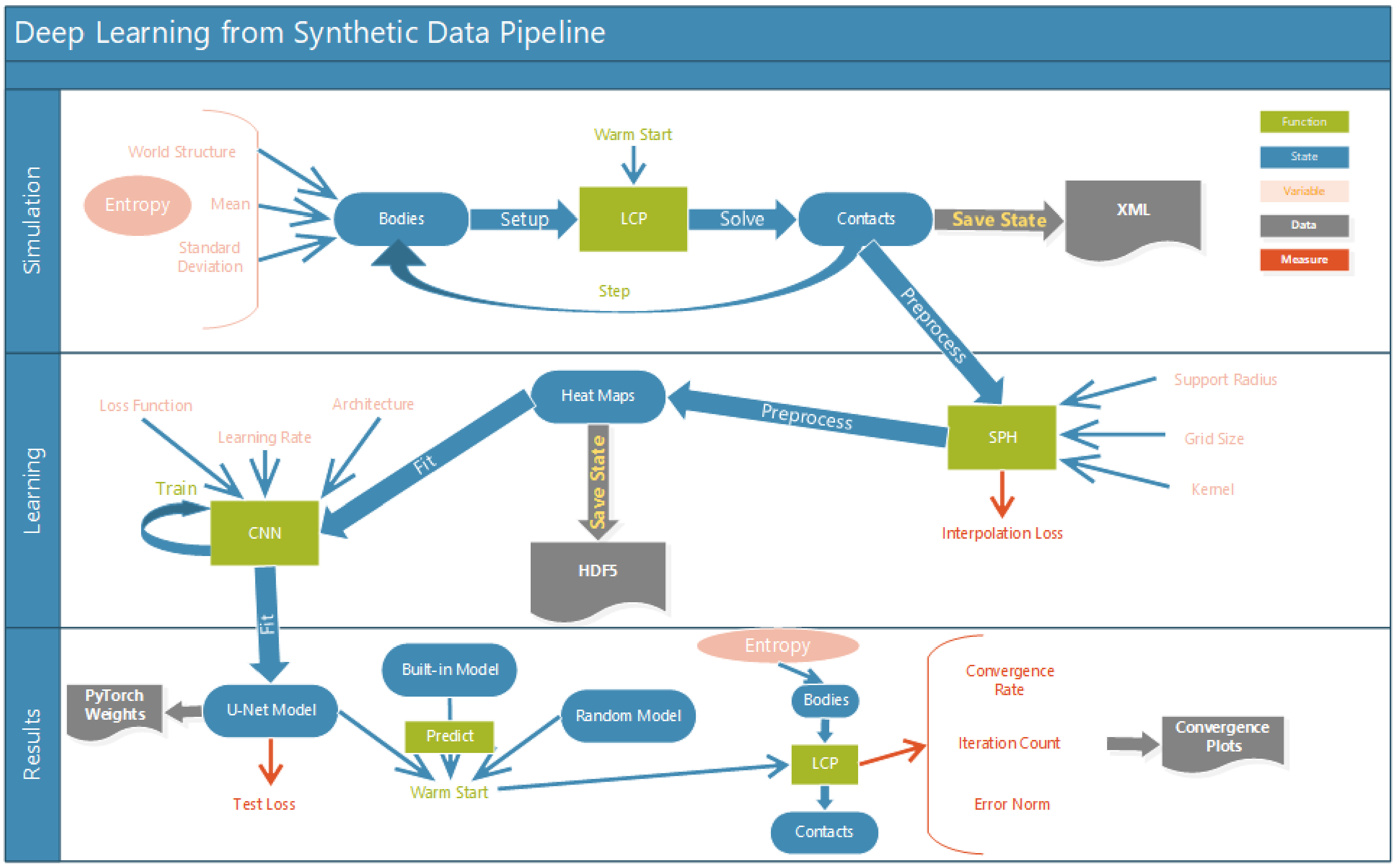


Figure 6.1: Deep Learning from Synthetic Data Pipeline

This "pipeline" can be restarted from any point where we have saved data (XML, HDF5, neural network architecture and weights) in order to experiment with different values and setups. These are implemented in **`Jupyter`** notebooks, so exploratory data analysis and iterative development is possible.

6.2 Technology review

We rely on many open source technologies and libraries to produce our results. In this chapter, we will try to produce an opinionated review of how we used them, why they were helpful, and what were the pitfalls so that others trying to replicate our results will not make the same mistakes. This is not intended to be a exhaustive list of dependencies, but rather a list of practices and tools that worked for us.

`PyBox2d` simulator

Let us start by explaining how we used **`PyBox2D`**. This 2D simulator is a Python wrapper around the **`Box2D`** [5] library that uses **`swig`** to make calls to the c++ library from Python. **`swig`** uses the concept of interface files to map calls from Python into c or c++, and perform the marshalling operations. We have used this feature to modify the simulator according to our purpose: extracting profiling data from each step.

The Box2D simulator is a very simple 2D simulator which is mainly intended for games, therefore physical accuracy is not emphasized and the focus is placed on visual plausibility. Nonetheless, the rules of physics are still at work when solving every step, and it serves as a good candidate for our attempt to teach a neural network (some form of simplified) physics.

When running the simulation the program runs a step forward in time according to a designated time step. It computes forces and accelerations according to the layout of the bodies and then uses this information to compute the new positions after the time step. Once the bodies are perturbed, this leads to interpenetration. It allows contact points to be detected and normals to be computed.

Contacts are then solved by passing over all of them, determining and applying impulses and fixing inter-penetrations, for as long as this routine is needed. It starts with guessing some values for the impulses (which we will refer to as **warm start**), and then iteratively improving them until some stopping conditions are met. After setting some impulse, velocity and positions are updated and checked for stability. If interpenetration has been reduced below a certain threshold or a maximum number of iterations has been reached, the simulation step is performed.

The modifications we have made have to do with extracting a large number of parameters and profiling data, unavailable normally in the Python interface. We added methods for tracking the number of contacts, number of iterations until termination for every step, and norms for the impulses being applied. These change according to how we warm start the steps therefore we can use these values as proxies used to compare between different methods.

`jupyter` notebooks

A very big part of the work has to do with the development experience. In order to build software quickly, it is important to have a good environment that provides quick feedback and results. While in the case of traditional software tests are the preferred way of ensuring quality, when working with data driven development we regard as very important to have a interactive "communication" with the computer that allows for plots, animations or matrices to be shown quickly.

One such tool is the **`jupyter`** server, which allows REPL (Read-Eval-Print-Loop) development in a web-based interface. The web client can communicate with local or remote kernels, and more details about this are given in chapter 6.3.

It can edit code, run shell commands, but the most useful feature is the notebook: a form of literate programming [14] (term coined by Donald Knuth where code, results and comments are interspersed according to the thought process of the developer).

Although this type of file is transient, we can move the important functions and classes over into separate .py files, while using the notebooks to test their behavior for different inputs. Auto-reloading of changed files is supported, as well of `pdb`-style debugging.

Many types of multimedia content are allowed (due to the web-based interface), and we have found it very useful to generate animations of the simulations where scrolling through frames is allowed, visualize images of the attributes involved or use progress bars to track long-running tasks.

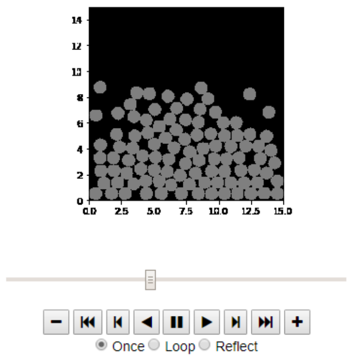


Figure 6.2: An animation running inside a notebook cell

All the notebooks used in the testing of methods and data generation are included in the project repository [1] and are the entry points into running the project.

`pyTorch`

Developing in a REPL environment requires a agile set of libraries. Having experimented with **`TensorFlow`** first, we discovered that it uses a declarative syntax in order to compile a computational graph, before any computation can be performed. This made running a single step or function of the network pretty tedious, because it required lots of prior setup. Nevertheless, **`TensorFlow`** is still recommended for production or training huge models, as the computational graph idea is very useful when running on multiple machines.

For developing our deep learning models we chose to use **`pyTorch v0.3.1`** instead (deprecated now in favor of v1.x.x). The execution model is closer to a programmer mindset: Functions are executed eagerly when called, debugging in possible using the built-in `pdb` and tensor can be inspected as any other Python variable. What **`pyTorch`** basically offers is the execution

[1] `https://github.com/s0lucien/Deep-Contact`

of mathematical computations while preserving the values of intermediary gradients - in essence, differentiable programming.

The development method used was to construct the model by creating smaller building blocks using the **`pyTorch`** standard library. The best way to test these blocks was to use random tensors of desired shape, feed them as input and see that there are no errors or shape mismatches in the output.

This library also provides data loader interfaces that have to be extended and how we did this is explained in more detailed in section 6.4.

`fast.ai`

Adding a layer on top of the NN library was the **`fast.ai`** framework. Started from a course with the same name, it provides a pragmatic and opinionated project including best practices, state-of-the-art algorithms and accessible explanations.

The course encourages students to take ownership of the code, and many of the technologies and ideas presented in this thesis have been suggested in their lectures. A very good measure of quality of a OSS project is an active community and a low number of open issues with the code (keeping up with the workload). The library has both qualities and because it is based on **`pyTorch`**, it was a very good choice.

An important note to make is that the code uses the "old" version of the library (0.7), incompatible with the new pyTorch, as it is based on the lectures from 2017-2018. In this sense some of our code can already be considered at the time of writing legacy code, due to the fast progress and refactoring of the underlying frameworks.

`HDF5`

The result of the preprocessing step is a multi-channel grid, in essence a collection of multiple dense matrices. The main requirement is that it provides fast parallel read access to them, so that getting data into memory is not a bottleneck in the training process.

The **`HDF5`** Hierarchical Data Format was one of the options considered, along with simply storing every multi-channel tensor as a file, and performing the preprocessing step at runtime, before every step, as a data augmentation procedure. We ended up using this format due to the fact that it allows a consolidated storage for our dataset. While being criticized for its write performance(that may lead to dataset corruption), reading from it has proved very easy. We generate the dataset for a choice of hyperparameters (grid size and support radius), and then do not touch it again. The case against processing at runtime was that it was slow(of course, clever caching would improve it, but that would lead to persistent storage again) and not able to saturate the GPUs.

This format allowed us to store and work with the large amount of data as if it was a Python dictionary/list, and allows for lazy loading.

6.3 Working in a shared environment - Using SLURM

The University of Copenhagen has fortunately supplied compute resources to this project. This was done by allocating permissions to the **`SLURM`** cluster that the Image Section provides. Running batch jobs using this task scheduler is well detailed but in order to use **`Jupyter`** notebooks and run them interactively another approach had to be used. After spending the time to set up a fast feedback loop for the work environment, we hope that this guide will be helpful for others who want to follow the same path.

The centerpiece of the method is the **`remote_ikernel`** [6] plugin for **`Jupyter`**. It allows for the local machine to establish `ssh` sessions, parse responses from the **`SLURM`** server and connect to the **`ipython`** kernel running on the allocated machine. It is important to mention that at the time of writing a Pull Request [2] needs to be manually merged for it to work with KU's infrastructure.

The protocol is not exactly straightforward and can go wrong in many ways, so it is relevant to understand what is happening under the hood. Documentation is intended to the users of this plugin but we found ourselves digging through the code to understand how it works when faced with issues, which is why the following UML Sequence diagram is useful: 6.3

The documentation suggests to use connection multiplexing (and how to set it up) for the `ssh` connections, which allows creating persistent tunnels to the servers in use.

Enabling this workflow is quite manual, and requires setting up 2 SSH sessions: one with the SLURM server, and one with the machine where GPU resources were allocated. We have found that a very useful tool that can facilitate this is `tmux`, the terminal multiplexer which allows easy switching between the 3 consoles : one running **`Jupyter`**, one connected to the **`SLURM`** server, and another one connected to the GPU instance.

The dependencies are : the SLURM session needs to be in place before the **`jupyter`** kernel is started. Once the kernel is started on some machine, the machine hostname will appear in the **`jupyter`** console log. In order to be able to communicate with the GPU server, a `ssh` tunnel needs to be created from the machine running the **`jupyter`** server to the GPU host. Once this is in place, it will be used for all further communication (because multiplexing allows passwordless authentication).

[2]`https://bitbucket.org/tdaff/remote_ikernel/pull-requests/5/ensure-selfhost-is-always-unicode/diff` fixes hostname parsing after the resource is allocated

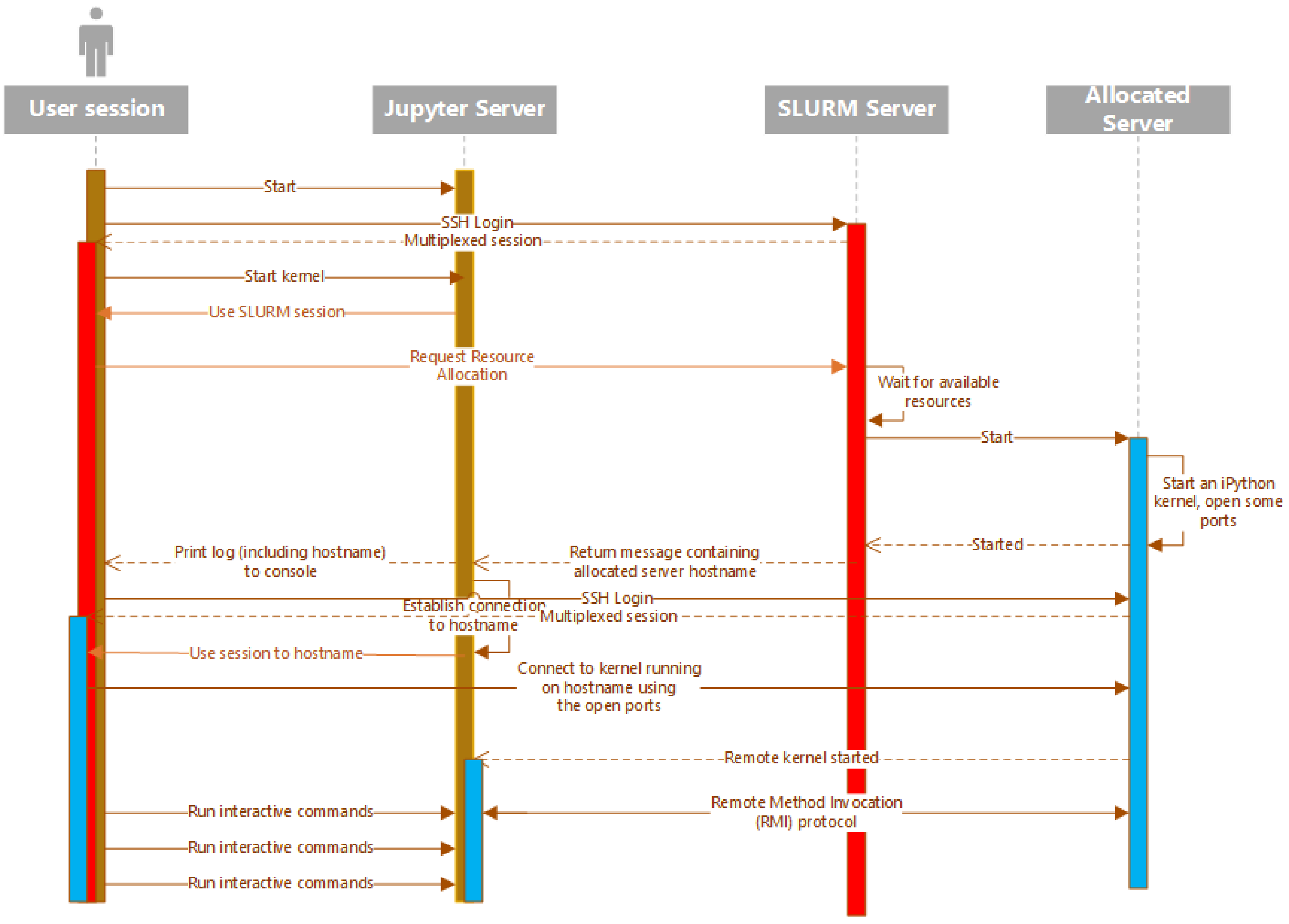

User session
Jupyter Server
SLURM Server
Allocated Server
Start
SSH Login
Multiplexed session
Start kernel
Use SLURM session
Request Resource Allocation
Wait for available resources
Start
Start an iPython kernel, open some ports
Started
Print log (including hostname) to console
Return message containing allocated server hostname
Establish connection to hostname
SSH Login
Multiplexed session
Use session to hostname
Connect to kernel running on hostname using the open ports
Remote kernel started
Remote Method Invocation (RMI) protocol
Run interactive commands
Run interactive commands
Run interactive commands

Figure 6.3: Running interactive sessions on SLURM

6.4 Implementing the U-Net

Let us now examine how the U-Net architecture was designed in our case. Firstly, it is important to note that the model uses the (fast) **`pyTorch`** functions for convolution, concatenation, batch normalization and up/downsampling. Here we will give a short review of the building blocks and abstractions that we created in our thesis. This is by no means a flexible framework, but instead it is created for our specific instance of problem.

For those more experienced or looking for a better and more adaptive architecture we have found the `fastai.vision.DynamicUnet` to be useful, which also supports transfer learning by including a pre-trained contracting path.

Most of the work went into using the **`pyTorch`** classes in order to supply a optimized training pipeline that would saturate the GPUs with the grids.

The fist step was to extend the `BaseDataset` class so that the `HDF5` file cand be read. This code is located in the `SimulationHdf5Dataset` class, and it only uses the `h5py` library to open the file for read and pull contacts and body frames from the persistent storage into memory. It is important to note that the BaseDataset allows for multithreaded access, so that we can supply the number of reading threads.

The next step was to ensure that the grids are padded to the size needed as input. As mentioned previously, the height and width need to be divisible with $2^{\#downsample}$, so we pad the input which is originally 121×121 to 128×128.

In order to improve the accuracy, it is good practice to normalize the data. That is why we compute the mean and standard deviation of the whole dataset, save it, and afterwards create a `torchvision.transform` for both inputs and outputs. These can be seen in the `Norm` and `PadToSize` classes. If we want to use the predictions, we will use the `UnPad` and `DeNorm` classes in order to reshape and bring the grid back to its original scale. As we will see later, this lead to a significant improvement in accuracy.

Once this data reading and transforming part is in place, we can use the `torch.utils.data.DataLoader` class to feed the inputs and outputs to the network during the training. We can specify the number of working threads here, and we found that 8 would be enough to saturate the GPUs.

The network class `UNet` is built from smaller blocks, as can be seen in Fig 6.4. We will now detail how they are constructed.

The main building block is the `StdBlock`, and is comprised of `torch.nn.Conv2d` with a kernel of 3 with padding of 1, that keeps the height and width of the heatmap the same. After this a `torch.nn.BatchNormd2d` layer is added to stabilize the training. The last step of the standard block is `torch.f.relu` activation. This block is used in order to change the size of the "volume" of the information, by expanding or shrinking the number of channels.

In order to provide a scale-space approach, the `StdDownsample` block uses average pooling `torch.nn.AvgPool2d` followed by Batch Normalization. This will divide the height and width by 2 when executed.

At the lowest level, dilation convolution is done by setting the `dilation` and `padding` on the convolution layer of a `StdBlock`.

To come back to the original size, the `StdUpsample` uses `torch.nn.ConvTranspose2d` followed by Batch normalization.

The UNet saves the output before the downsampling step and concatenates it with output after the upsample of the same size.

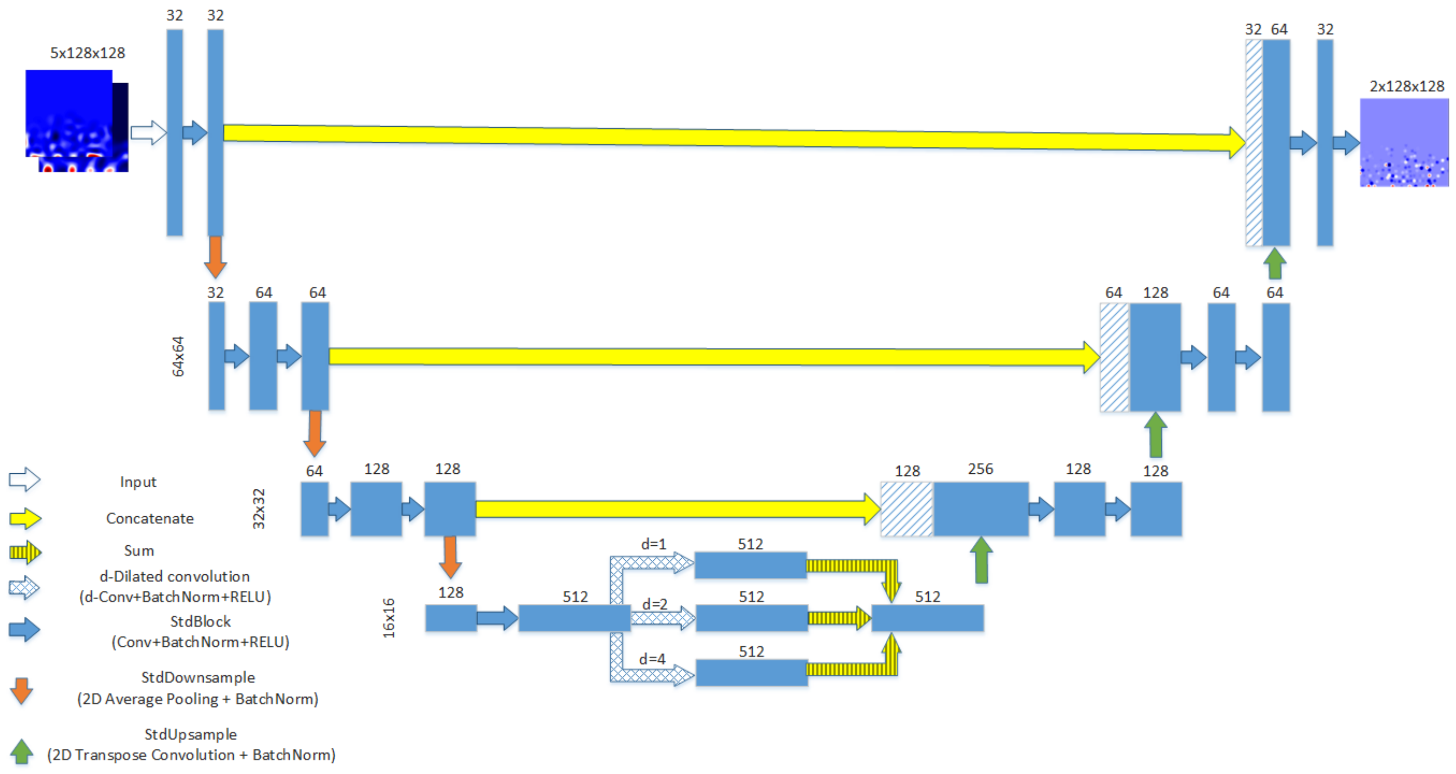


Figure 6.4: The U-Net used in the thesis

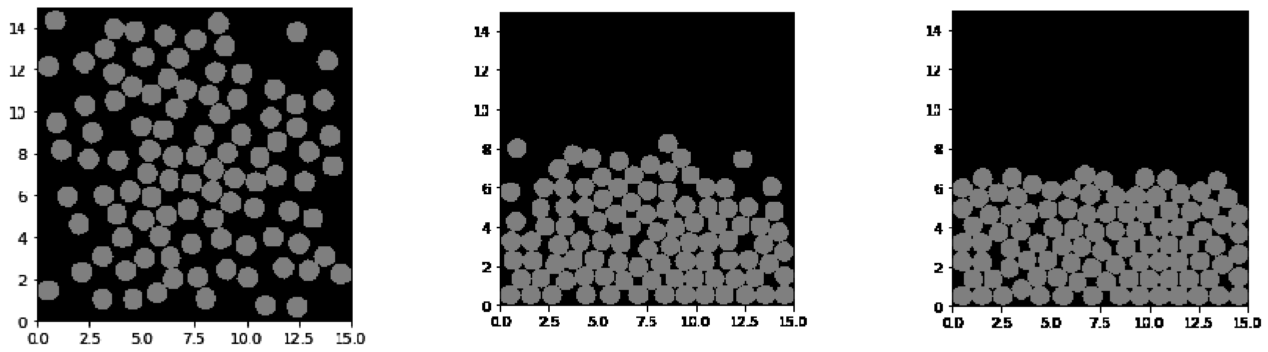
Chapter 7

Results and Verification

In the following section we will detail what values we used for our experiments and showcase our results. Moreover, we will try to demonstrate why we believe the implementation is correct by explaining how we tested our algorithms.

7.1 Data generation

The first step is to generate the training data. We chose to create a very simple scenario : Randomly sample a 2D Gaussian to get the positions of 100 balls of the same radius (0.5 m) placed inside a 15×15 box. The Gaussian distribution mean is placed in the middle of the square box, so the generated balls cluster around the center. These balls fall under the effect of gravity, as can be seen below :



(a) Beginning (b) Middle (c) End

Figure 7.1: Sample simulation at beginning, middle and end

Afterwards, the simulation is ran for 250 steps with a time step of $1/100$ sec. 110 such simulations are created and every frame from them is stored inside xml files. We choose to only start saving the xml files when contacts start appearing, so the first 10-20 frames are not recorded.

The xml store the following body attributes : mass, position on the x axis, position on the y axis, velocity on the x axis, velocity on the y axis, orientation, spin(angular velocity), inertia as well as the body shape and unique index of the body(which does not change during the simulation). This identifier can be used to construct graphs if it would be required.

For the contact attributes we store: the 2 identifiers of the bodies in contact, the position on the x axis, the position on the y axis, the normal x component, the normal y component, normal impulse(we refer to this as λ) and tangent impulse.

Verification

We have verified the implementation by using **`OpenCV`** and **`matplotlib`** to draw frames and save them as `.png` files. We used these files to create animations in our notebooks and checked for visual plausibility. In order to verify our xml export implementation, we have created a xml importer that is able to create a world from a xml file state and continue the simulation from there.

Discussion

We found that the quality of the warm starts is very sensitive to changes in the time step size, but quite robust against changes in the standard deviation of the Gaussian distribution used to sample the original positions. The fact that the box is located in a static position leads to very similar end states for the simulation (with the balls behaving as if they were part of a fluid, and filling up the box up to an even level). A take-away from this is that the time step must be constant throughout the pipeline.

7.2 Smooth particle hydrodynamics

We chose the values of 0.5 for the support radius, with a resolution of 0.125 for the x and y axes. These were chosen by minimizing the particle-grid-particle error using cross-validation. The procedure was done in collaboration with Lukas Engedal and results from it can be found in [7].

This choice of hyperparameters lead to grids of size 121×121. Below we show how an example of how

Verification

In order to verify that the grid transfer is working correctly and plotting works the way it should, we created a simple test case with a small grid where we place 2 particles of different magnitude at specific points in the box. We quickly found out that by convention **`matplotlib`** sets the x-axis of an image at the top. Therefore, our plot method was modified to flip the grid so that it corresponds to the animations we created in the data generation process (and our mental model of axis placement).

Data integrity was checked by using the `scipy.interpolate.griddata` (which performs linear interpolation between the points in the image) and visually inspecting the resulting plots. Below in 7.3 you can see values of the

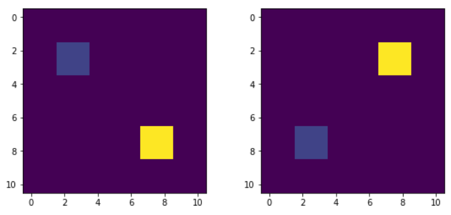
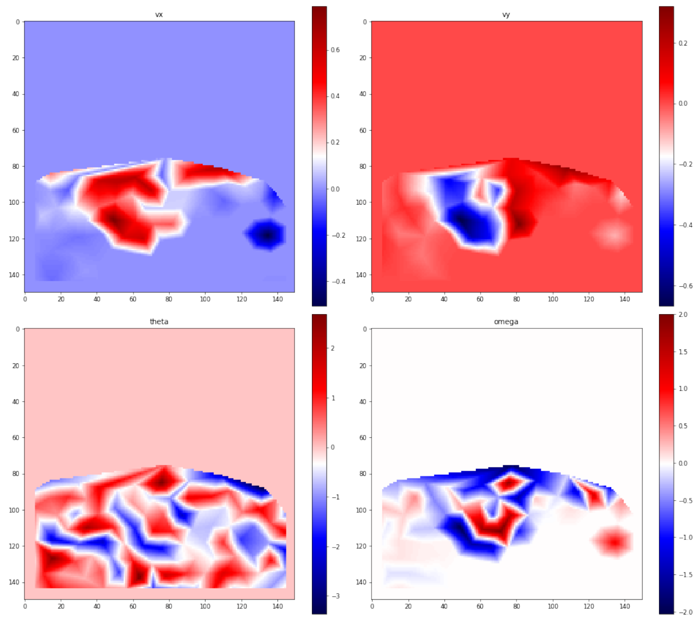
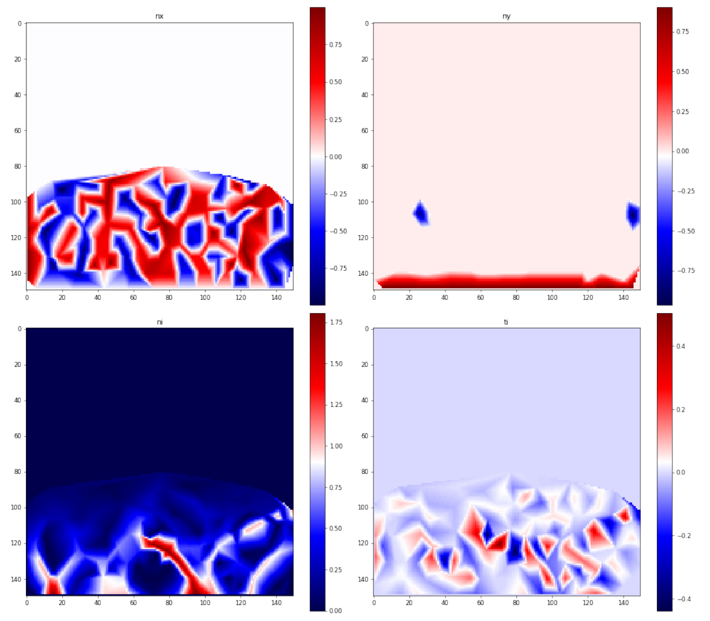


Figure 7.2: In a grid from (0,0) to (2,2) and resolution of (0.2,0.2) we place $p_1 = (0.5, 0.5)$ with value 1 and $p_2 = (1.5, 1.5)$ with value 5. We find out when plotting that the image is flipped vertically

channels extracted from the simulation. The images show correct data: the velocities in the x direction show movement of the particles towards the center, while y velocities show a movement of subduction (like tectonic plates) from the left part, where the pile is imbalanced. The angular velocity and orientation are noisy (as expected), the contact normal x direction is noisy showing how the balls are arranged, while in the y direction it is pointing upwards under the effect of gravity on the bottom, and downwards on the top where balls are still falling. Normal impulse is showing root-like formations as expected.



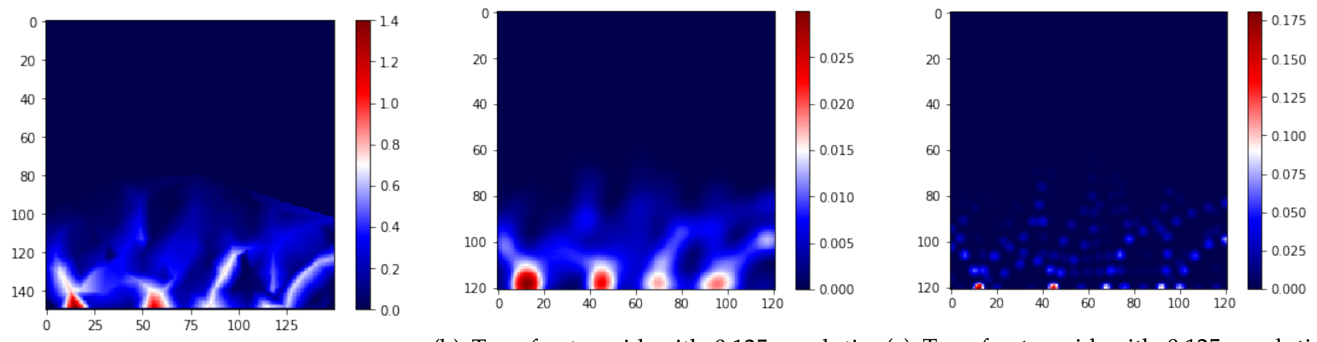
(a) Velocity in the x direction, velocity in the y direction, omega is angular velocity, theta is orientation(pose)



(b) Normal in the x direction, Normal in the y direction, normal impulse tangent impulse

Figure 7.3: Channels extracted from the simulator

We used the same function to compare the "naive" linear interpolation procedure with the values obtained from the SPH transfer. The plots at 7.4 show the normal impulse of contacts in the same frame using 3 different grids. We can see that they are similar (heat-spots appearing in the same spots), with differing amount of "splattering" for the particles due to differing support radiuses.



(a) `griddata` transfer

(b) Transfer to grid with 0.125 resolution and 1.5 support radius

(c) Transfer to grid with 0.125 resolution and 0.5 support radius

Figure 7.4: Comparison of the contact normal impulse magnitude of the same frame for different grids

Discussion

We observe that the original values become greatly decreased in the process of grid transfer. Even with a very small support radius of 0.5, the original values become less than 20% of their original values. This brings any warm-start value close to 0, and as we will see in the next sections, this greatly affects our performance.

Another noteworthy information is that the choice of resolution is good for quick experimentation because it leads to (reasonably) small grids.

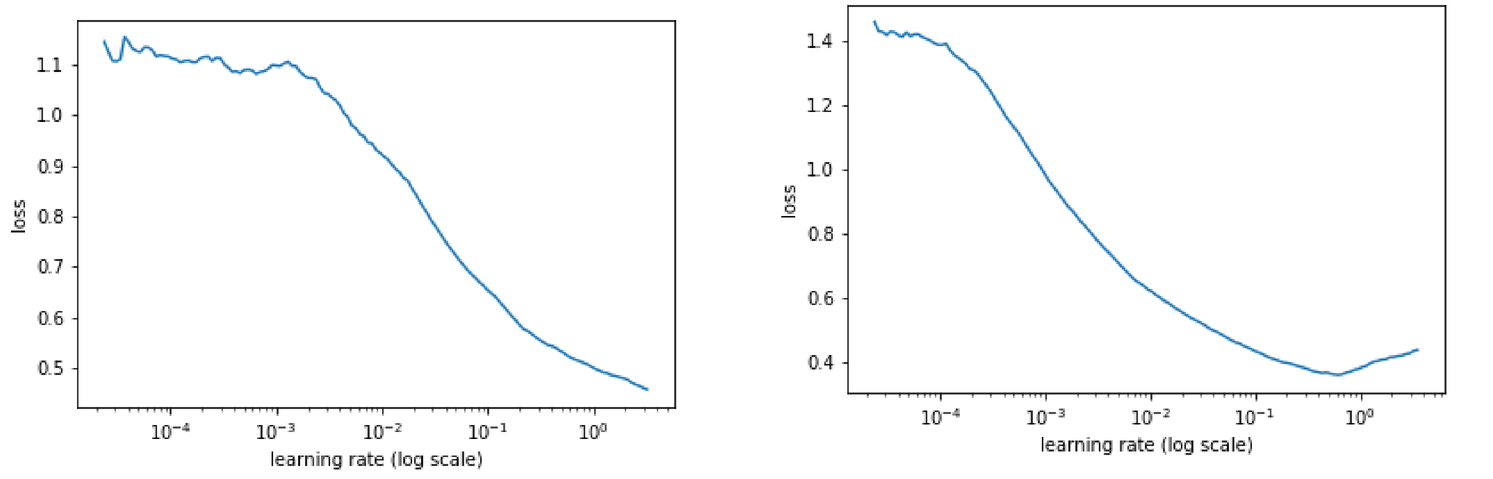
7.3 Neural Net Training

The U-Net described earlier was trained on 2 different datasets: one containing a 5-channel tensor with x-velocity, y-velocity, angular velocity mass and inertia, and the other a 7-channel tensor that also includes normal and tangent impulses from the previous simulation step. The target is a 2-channel tensor that contains normal and tangential impulse. We have chosen to ignore the orientation as it is just a state variable and it is not so relevant for the shapes chosen.

We have experimented with the Mean Square Error and the Mean Absolute Error, and found MSE to lead to a more robust decrease in the loss. That is the loss function chosen for both models.

The dataset is randomly split into 3 subsets : 10% for the test set, 20% for the validation and 70% for training. The indices in these subsets are then saved so that the training can be restarted without resampling the data.

We use the learning rate finder to choose a good learning rate that correctly describes the scale of the problem, as can be seen in the plots below:



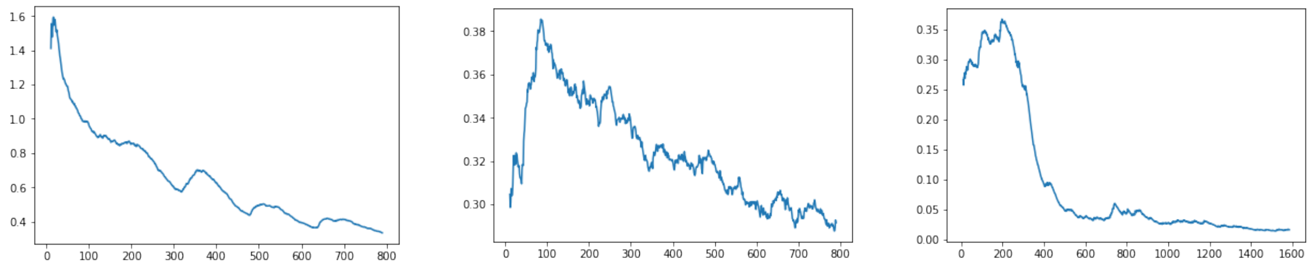
(a) Channels used are velocity, angular velocity, inertia and mass

(b) Channels also include normal and tangent impulses from the previous simulation step

Figure 7.5: Learning Rate Finder plots for the 2 models tested

For the "no temporal" model, we ended up choosing a learning rate of 0.5, as it seems like the loss is still decreasing at that point, while still being robust.

For the "temporal" model, we ended up choosing a learning rate of 0.05 - adding the contact information has decreased the scale of the problem.



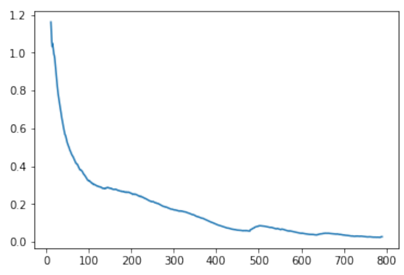
(a) Epochs 0 to 5, 5 cycles

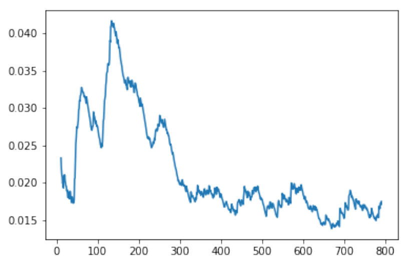
(b) Epochs 5 to 10, 1 cycle

(c) Epochs 10 to 20, 1 cycle

Figure 7.6: Loss plot for model with no temporal information

We train each model for a short time (20 epochs) and observe similar results : the loss very quickly decreases, and becomes very small after around 15 epochs. In the end, the model with temporal information outperforms the model without it by a very small margin (the final training MSE is 0.01445 temporal and 0.015347 for the atemporal model). The difference in performance is so small that we will choose to use the simpler model for the rest of the thesis.


(a) Epochs 0 to 5, 5 cycles


(b) Epochs 5 to 10, 1 cycle

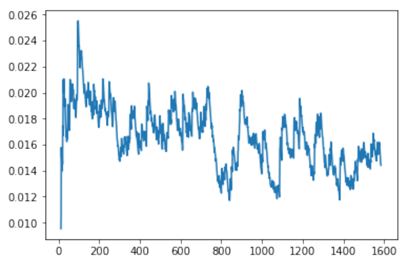
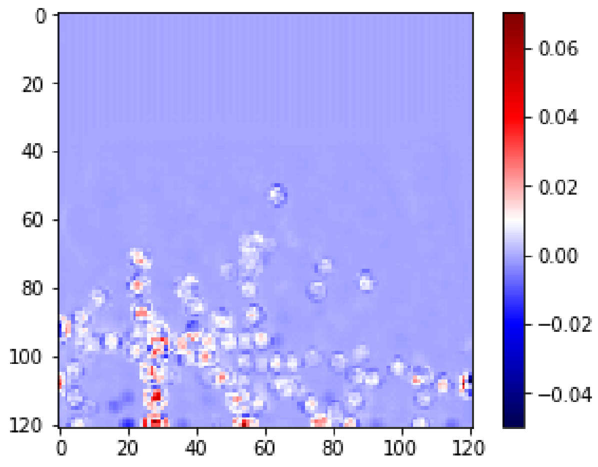

(c) Epochs 10 to 20, 1 cycle

Figure 7.7: Loss plot for model with temporal information

Verification

We can perform a sanity test by visually inspecting the result of $\Delta\mathbf{Y} = \mathbf{Y} - \hat{\mathbf{Y}}$.



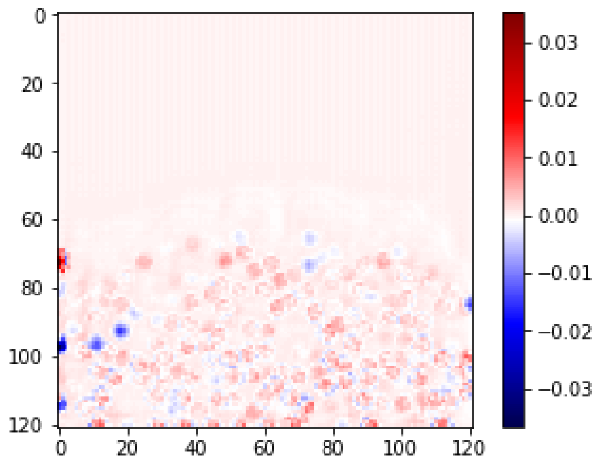


Figure 7.8: The difference between expected and predicted normal impulse for 2 randomly selected frames

Discussion

We observe that the difference $\Delta\mathbf{Y}$ is very small (Close to 0 in most of the simulation), and the MSE loss is also very low. This means that the chosen channels can correctly predict the values of the normal and tangent impulses. We also observe no significant improvement by adding the temporal information, neither in convergence (the networks seem to train at very similar rates from the loss plots) nor in the final MSE.

We can also notice a degree of "confusion" on the last part of the training for the temporal model. This may be because of a bug we have not fixed: we are not correctly setting the value of the contact normal and tangent impulse in the first frame in the simulation. Instead, we use the value of the last frame in the previous simulation. This can be fixed by including flags to denote whether the grid is the first frame of a simulation for every image in the dataset. It is possible to get slightly better results had it not been for this involuntary noise.

We can also see the effect of cyclical learning rates on the training in epochs 0-5 (more in the atemporal model) which matches the expectations from 5.2.

The last thing to add is that the test split was not necessary: we can always test by running another simulation, and that is what we actually do to evaluate performance.

7.4 Warm start models

A number of different warm starting methods have been explored in order to provide a overview of the qualities required from one such method:

- The `copy` model will first solve the contact problem, find the optimal values and then use them to warm start the simulation. In some sense this is the "oracle" that gives the right solution. `copy0` and `copy2` round this value to 0 and 2 decimals respectively, so that we can evaluate the sensitivity to numerical accuracy errors.

- The `bad` model provides a very bad (large) normal and tangent impulse starting value of 50.

- The `random` model provides a random normal impulse value between 0 and 5 and a tangential impulse between -2 and 2 because we have observed that is the range returned by the LCP solver.

- The `builtin` model starts the solver from the contact values from the previous frame.

- The `none` model initializes the solver with 0s .

- The `grid` model will transfer the previous frame contact values to the grid, then use interpolation to get the warm start.

- The `UNet` model will transfer the previous frame body attributes to the grid, make a prediction with the trained neural network, then use interpolation on the NN output to get the warm start.

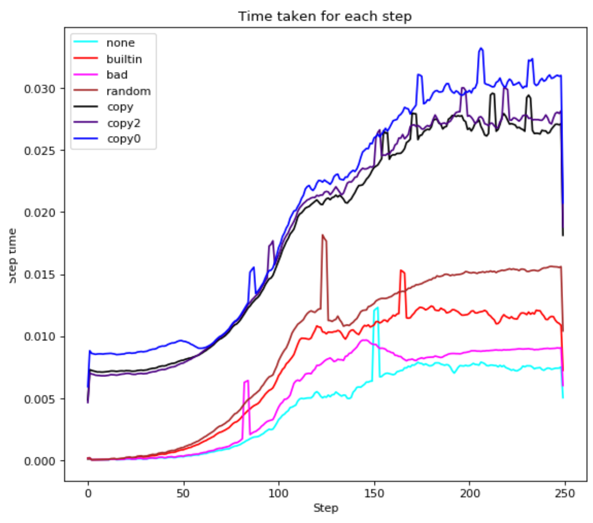
Verification

We run a 30 simulations with the same starting world (by setting the same 30 random seeds for the world generator), so we make sure that we are solving a problem with the same initial conditions. The pragmatic approach would be to measure the time taken for each step , as is shown in plot 7.9.

Another question is: are all initial conditions resulting in the same configuration after 250 steps ? How robust is the simulator against bad values of warm starts ? We can use the contact count to infer some information about the similarity between the same frame of different warm start methods, as in plot 7.10.

Discussion

Our findings show that the time is not very relevant due to the way we are measuring it. We can see that even solving the simulation from bad warm starts is faster than using very accurate warm starts. This is because we are



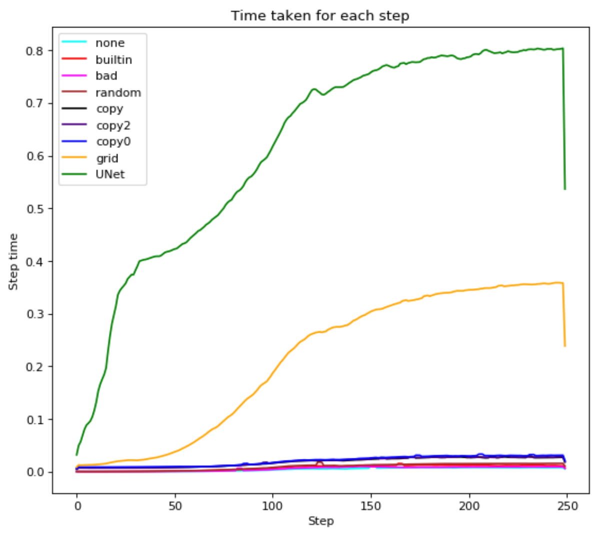


Figure 7.9: Time to solve each step with and without the `grid` and `UNet` models

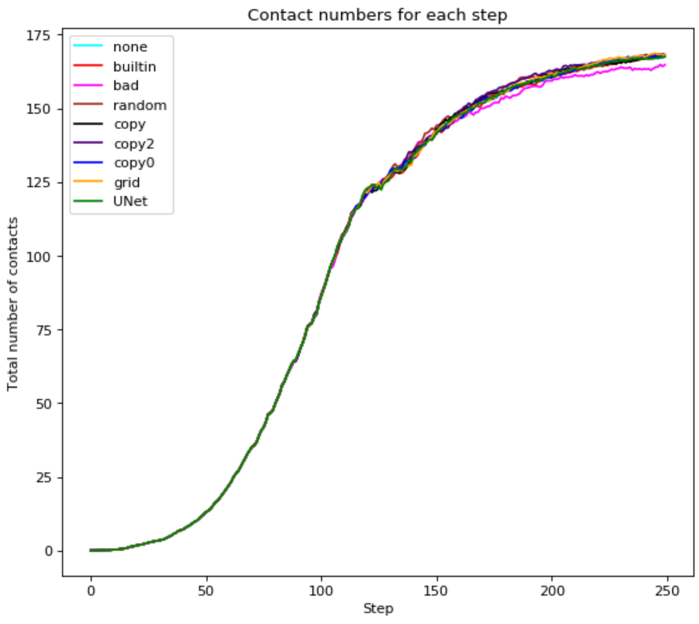


Figure 7.10: Number of contacts

solving it twice in the case of the `copy` model. Sampling the random value also takes more time than actually solving it from a given value.

Another observation is that the UNet inference and the grid transfer times are huge compared to solving the actual LCP.

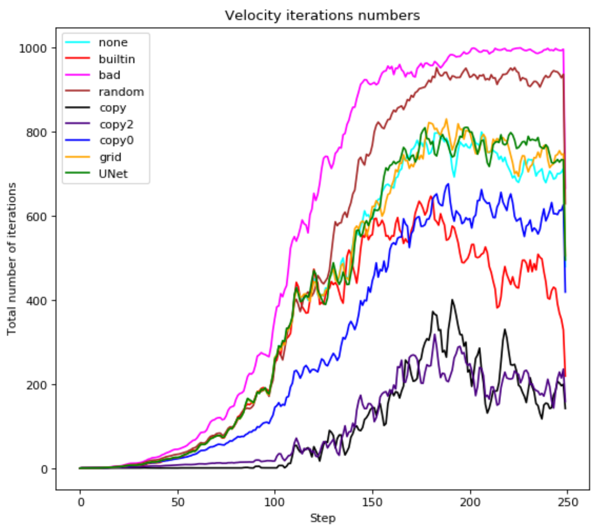
Of course, this can quickly be solved through optimizing the code, or even only measuring the time it takes for the actual step to take place, not including the `PreSolve` step where we choose the warm start.

As for correctness, most models yield the same world configurations except the `bad` model which diverges around frame 150.

7.5 Convergence plots

In spite of the fact that time is not relevant as a performance measure we were able to find other ways of measuring how good the warm starts actually are.

The first is the number of velocity iterations that the solver makes. As mentioned in Chapter 3, the simulator makes a (maximum of 1000) iterations to find velocities that satisfy the interpenetration constraints, given some start value of λ. By recording it, we can construct plot 7.11.



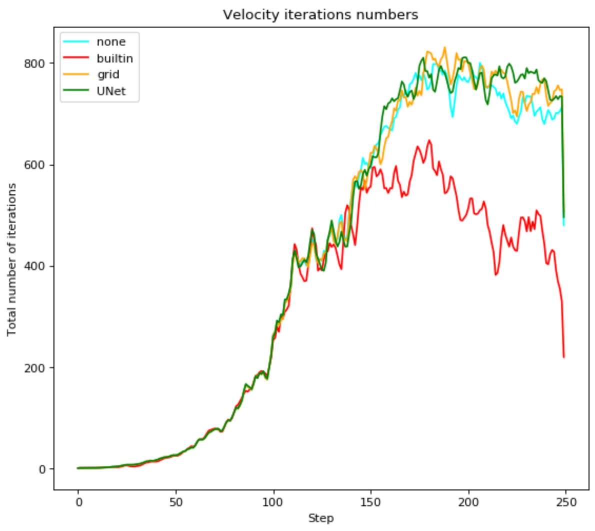


Figure 7.11: Number of iterations required to solve each step with and without the `copy`, `bad` and `random` models

Another way to check our warmstarts is the absolute difference between the solution and our guess. This is shown in plot 7.12.

We can also study the convergence of the solver, given our warm start. How many iterations of the LCP are needed in order to solve the step (make the velocity in the normal direction 0 at the contact points)? The answer can be seen in plot 7.13.

Discussion

The velocity iterations numbers show that both the grid and the UNet predict values close to 0. We can also observe that the 0 values are good until a certain point, when something happens (around frame 150). This is actually when the balls have all fallen in place, and potential energy starts to dissipate, creating wave-like movements in the world. This is also the point where the number of contacts is the highest. The builtin model starts to outperform our models

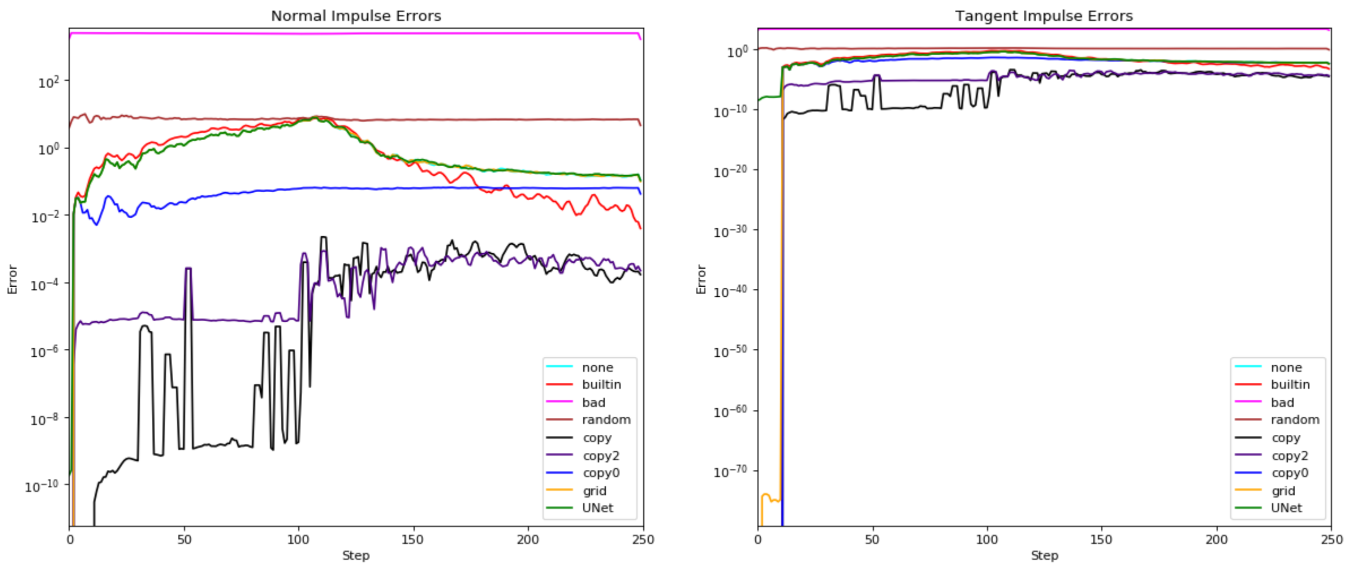


Figure 7.12: Impulse normal and tangent errors

because it uses previous contact information, while our grid does not have the right scaling.

From this plot we can also see that the copy world model is not very sensitive to numerical precision errors, as 2 decimals are enough to lead to a quick LCP solution. The `bad` and the `random` models seem to perform very bad, showing that the starting value is quite important to the solution.

From the impulse errors we can see even better that the `grid`, `UNet` and `none` models actually predict the same value, both in the normal and tangent direction.

The convergence plots show similar behavior of the solver with `grid`, `UNet` and `none` leading to the same decrease in the residual impulse.

The fact that `UNet` and `grid` models coincide shows that the deep learning model is working very well : The grid is "teaching" the neural network how to predict contacts using only body data, and the network is able to infer both position and magnitude of the contacts with only minor mistakes. However, expecting the "student" to outperform the "teacher" is not realistic and any prediction that is better is simply due to randomness (the `UNet` number of iterations is slightly under the one of the `grid` and the `none` model at few occasions).

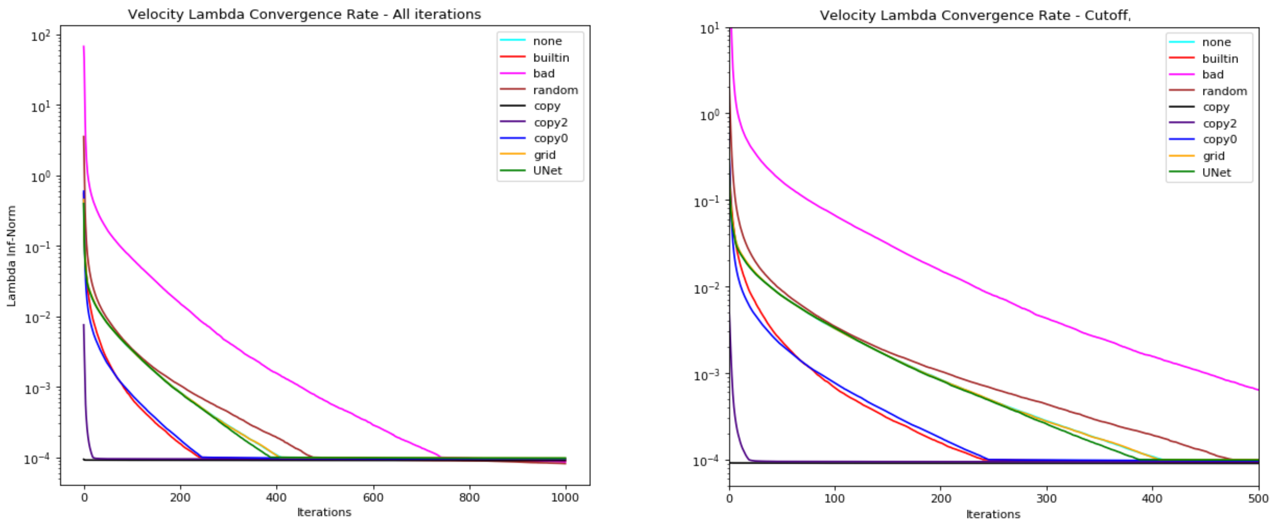


Figure 7.13: Median value(across the 30 simulations) of the impulse value after n iterations, where n is the x-axis. The image on the right cuts off n at 500 in order to zoom in on the interesting region

Chapter 8

Conclusion

It is now time to reflect on what was done and our findings: We have built a synthetic data generation framework using **`PyBox2D`** and successfuly trained a **`pyTorch`** UNet model that is able to infer information about the simulation. What went good and what can be improved ?

It seems that the weak point of our pipeline is the SPH method chosen. Even if the intuition behind it seems correct (a attribute acts around the particle location with decreasing intensity, and these values can combine in order to cancel or empower each other), the restriction that the weights of the kernel sum up to 1 seems arbitrary. Tha values in the grid are simply too small to be useful as warm starts.

On the other hand, the U-Net architecture is fit for the task, reaching very low(close to 0) MSE. This probably has to do with the type of simulation used (balls falling inside a box under the effect of gravity) without the effect of external forces. We can add that for this model, the possibility of over-fitting is not accounted for because we are able to always generate new worlds with initializations that are different from the ones in the train/validation set.

We are unable to "beat" the built-in warm-start model and reach the performance of the 0-warmstart model (because we are predicting zeros). Here we should add that for this scenario 0 seems like a good warm start value.

As for the technologies used we can wrap up by saying that **`Jupyter`** notebooks are a very good fit for this kind of experiments particularly because they allow us to quickly execute functions on remote compute resources and offer a high degree of interactivity which is very helpful for data-driven projects.

8.1 Low-hanging fruit

There are some changes that can be done in order to improve the model, and we will list them in the order of increasing difficulty of implementation:

- Make changes to the grid transfer method: We can try instead to use the `griddata` function that created the plots, and assume that any value between contacts is situated on a bilinear interpolation between the values at particle location.

We can also try to scale the grid result from SPH such that the value at the particle location matches the original value. These 2 suggestions are drawn in fig 8.1.

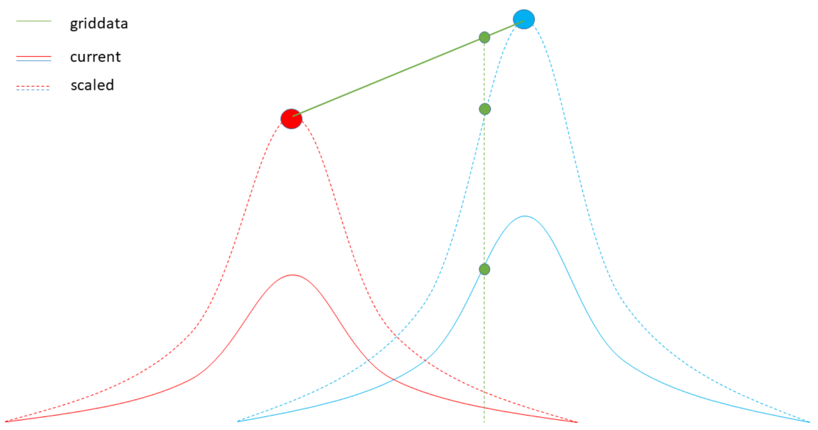


Figure 8.1: Suggestions to improve the grid transfer

- Make the `none` (and `builtin`) model perform worse: One of the things to try is increasing the entropy of the original and intermediary states.

We could rotate the box like a "washing machine drum", add bodies of different shapes (like smaller and bigger circles, squares, irregular shapes), add static bodies inside the box. Either of these would make it harder for the builtin model to perform well, and we may see a decrease in the UNet performance.

In this more difficult case it is possible that the UNet can learn more interesting heuristics and beat the built-in model.

8.2 More complex suggestions

In our implementation we were able to extract the simulation contact graph. This adjacency data is lost when transferring onto the grid, but recent findings in graph neural networks have shown promising results[2] in working with graphs coming from physics problems. It should be possible to use **`pyTorch geometric`**[1] [2] to reformulate the problem using the existing dataset.

To conclude, there are various ways of improvement before dropping the idea of warm starting using CNNs, mainly because the neural network has actually shown impressive results and is yet to reach its limit. More challenging datasets and better grid transfer may lead to significant payoffs.

[1]`https://github.com/rusty1s/pytorch_geometric`

[2]`https://rusty1s.github.io/pytorch_geometric/build/html/modules/nn.html#torch_geometric.nn.meta.MetaLayer`

Appendix A

Notation

To aid the reader, we will list the more unusual notations we use across the paper. We will do our best to keep this consistent:

- $\mathbf{x}$ - the bold styling symbolizes a vector
- $\dot{\mathbf{x}}$, $\ddot{\mathbf{x}}$ - the dot above a vector is the first, second, etc derivative
- $\mathbf{A}$ - uppercase bold is a matrix
- $\mathbf{x}^k$ - vector at step k in an algorithm
- **`pandas`** - the bold font refers to libraries or frameworks. This refers to the pandas python library.
- `some_method_name` - method or class name. We will try to use this font for code or shell commands in our explanations.

Appendix B

Linear Algebra Concepts

B.1 Projection operator

In intuitive terms, the projection of a vector $\mathbf{x}$ onto the line (or plane) L can be seen as the "shadow" of x on L. Let us go ahead and examine it in more formally :

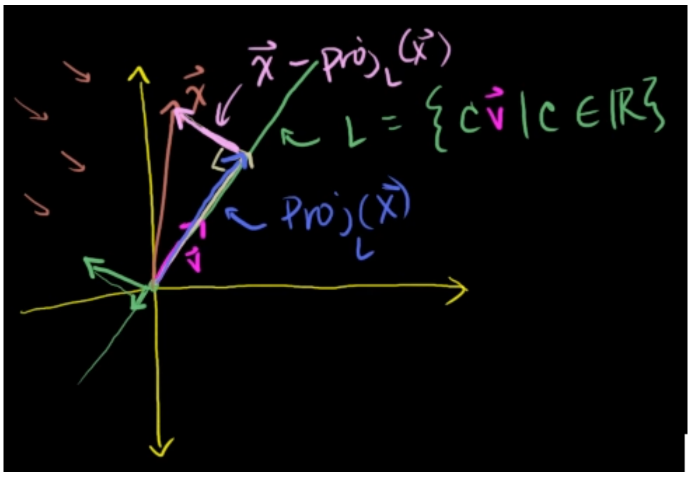
The $Proj_L(\mathbf{x})$ is some vector in L where $\mathbf{x} - Proj_L(\mathbf{x})$ is orthogonal to L (the inner product is zero). L can be seen as a line defined by vector $\mathbf{v}$ scaled by a real coefficient c.

$$Proj_L(\mathbf{x}) = c\mathbf{v} \tag{B.1}$$

To find c, we use the orthogonality property:

$$\langle \mathbf{x} - c\mathbf{v}, \mathbf{v} \rangle = 0 \implies \langle \mathbf{x}, \mathbf{v} \rangle - \langle c\mathbf{v}, \mathbf{v} \rangle = 0 \tag{B.2}$$

$$\implies \langle \mathbf{x}, \mathbf{v} \rangle = \langle c\mathbf{v}, \mathbf{v} \rangle \implies c = \frac{\langle \mathbf{x}, \mathbf{v} \rangle}{\langle \mathbf{v}, \mathbf{v} \rangle} \tag{B.3}$$



$\vec{x} - Proj_L(\vec{x})$
$\vec{x}$
$L = \{c\vec{v} \mid c \in \mathbb{R}\}$
$Proj_L(\vec{x})$
$\vec{v}$

Figure B.1: Projection of x onto L

Bibliography

[1] Jason Antik. *DeOldify*. 334030a0983333293f969664f9147a11e8dfd0d4. GitHub, 2019. URL: https://github.com/jantic/DeOldify.

[2] Peter W. Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv:1806.01261 [cs, stat]* (June 4, 2018). arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261 (visited on 04/23/2019).

[3] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. *Deep learning*. Vol. 521. 2015.

[4] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007.

[5] *Box2D*. URL: https://box2d.org/.

[6] Thomas Daff. *remote_ikernel*. 0.4.6. BitBucket, 2017. URL: https://bitbucket.org/tdaff/remote_ikernel.

[7] Lukas S Engedal. "Improving the iterative process of solving contacts for rigid bodies with the use of neural networks". In: (), p. 84.

[8] Kenny Erleben. "Numerical Methods for Linear Complementarity Problems in Physics-based Animation". In: (), p. 42.

[9] Kenny Erleben, ed. *Physics-based animation*. 1st ed. Charles River Media graphics. OCLC: ocm58054056. Hingham, Mass: Charles River Media, 2005. 817 pp. ISBN: 978-1-58450-380-4.

[10] Gao Huang et al. "Snapshot Ensembles: Train 1, get M for free". In: *arXiv:1704.00109 [cs]* (Mar. 31, 2017). arXiv: 1704.00109. URL: http://arxiv.org/abs/1704.00109 (visited on 04/06/2019).

[11] Vladimir Iglovikov and Alexey Shvets. "TernausNet: U-Net with VGG11 Encoder Pre-Trained on ImageNet for Image Segmentation". In: *arXiv:1801.05746 [cs]* (Jan. 17, 2018). arXiv: 1801.05746. URL: http://arxiv.org/abs/1801.05746 (visited on 04/09/2019).

[12] Nitish Shirish Keskar and Richard Socher. "Improving Generalization Performance by Switching from Adam to SGD". In: *CoRR* abs/1712.07628 (2017). URL: http://arxiv.org/abs/1712.07628.

[13] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Dec. 22, 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 04/06/2019).

[14] Donald E. Knuth. “Literate Programming”. In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: `10.1093/comjnl/27.2.97`. URL: `http://dx.doi.org/10.1093/comjnl/27.2.97`.

[15] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Restarts”. In: *CoRR* abs/1608.03983 (2016). URL: `http://arxiv.org/abs/1608.03983`.

[16] Augustus Odena, Vincent Dumoulin, and Chris Olah. “Deconvolution and Checkerboard Artifacts”. In: *Distill* 1.10 (Oct. 17, 2016). DOI: `10.23915/distill.00003`. URL: `http://distill.pub/2016/deconv-checkerboard` (visited on 04/06/2019).

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab et al. Vol. 9351. Cham: Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24573-7 978-3-319-24574-4. DOI: `10.1007/978-3-319-24574-4_28`. URL: `http://link.springer.com/10.1007/978-3-319-24574-4_28` (visited on 04/06/2019).

[18] Leslie N. Smith. “Cyclical Learning Rates for Training Neural Networks”. In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). Santa Rosa, CA, USA: IEEE, Mar. 2017, pp. 464–472. ISBN: 978-1-5090-4822-9. DOI: `10.1109/WACV.2017.58`. URL: `http://ieeexplore.ieee.org/document/7926641/` (visited on 03/29/2019).

[19] Ashia C. Wilson et al. “The Marginal Value of Adaptive Gradient Methods in Machine Learning”. In: *arXiv:1705.08292 [cs, stat]* (May 23, 2017). arXiv: `1705.08292`. URL: `http://arxiv.org/abs/1705.08292` (visited on 04/06/2019).

[20] Fisher Yu and Vladlen Koltun. “Multi-Scale Context Aggregation by Dilated Convolutions”. In: *arXiv:1511.07122 [cs]* (Nov. 23, 2015). arXiv: `1511.07122`. URL: `http://arxiv.org/abs/1511.07122` (visited on 04/11/2019).