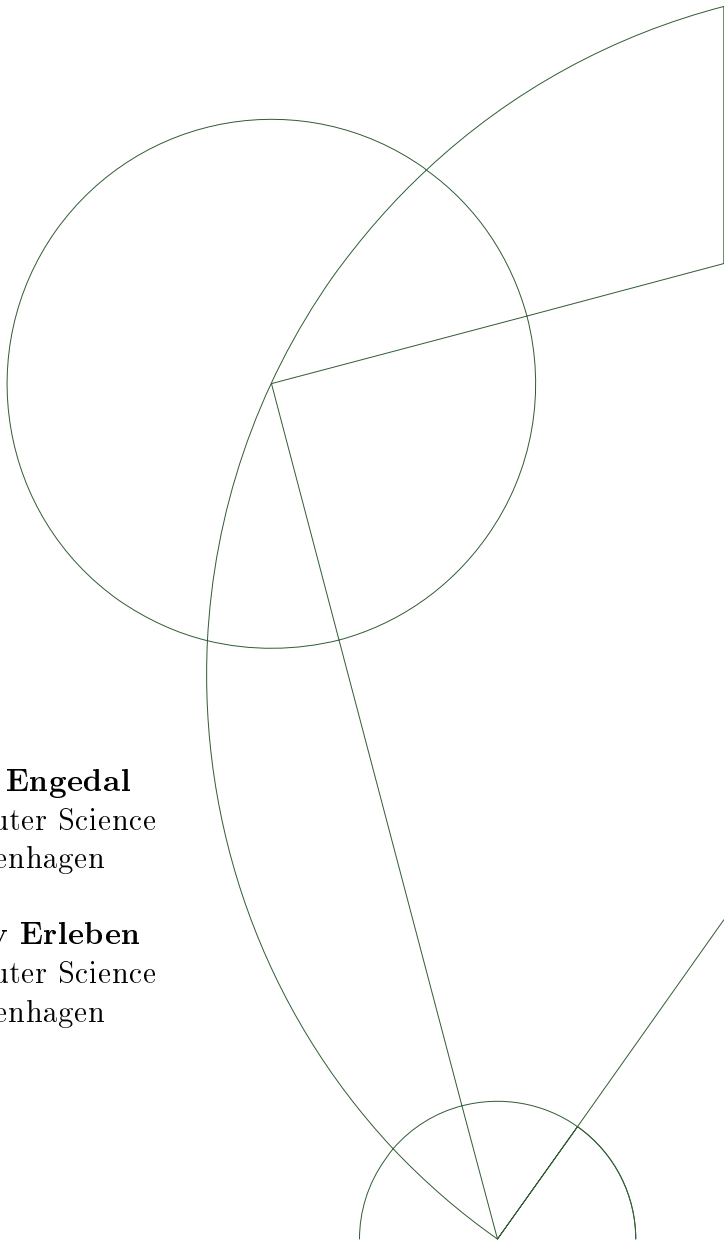
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Deep Contact

Improving the iterative process of solving contacts
for rigid bodies with the use of neural networks

Master's Thesis
6. August 2018



Author: Lukas S. Engedal
Department of Computer Science
University of Copenhagen

Supervisor: Kenny Erleben
Department of Computer Science
University of Copenhagen

Acknowledgements

First of all I would like to thank my supervisor Kenny Erleben for providing me with the interesting topic, guiding me through the process, and for providing help and advice about physics, simulation and in general. I would also like to thank his colleague Oswin Krause, for providing advice about neural networks in general and feedback about mine in particular.

Secondly I would like to thank Jian Wu and Lucian Tirca for working on the initial part of the project with me, and for providing feedback and someone to talk about the project with throughout.

Finally I would like to thank the Image group and the APL section at DIKU for providing me access to their servers, without which the training of the neural networks would still be ongoing.

Abstract

In this thesis we will be working with the `Pybox2D` physics simulator. Among other things, `Pybox2D` simulates collisions and contacts between objects, and solves the constraints associated with this through an iterative process, starting from zero and then moving closer to a result step by step. Our goal is to create a model which can provide the simulator with good starting iterates to use rather than just using zeros, i.e. values that are close to the final result, in the hopes that this will make the iterative process complete faster. Our model will be a convolutional neural network, and we will design and built it in `TensorFlow`. We will also need to design and implement a method for transferring data from the simulator to our neural network model, and back, which will be done using grids. All of this we have done, and we have then measured the performance of the simulator when using our model. The results are not overly impressive. Our model is capable of providing starting iterates that leads to better performance than what we get just using zeros, but it is not able to do better than the default method of generating starting iterates used in the simulator, which just relies on reusing results from previous iterations.

Resumé

I dette speciale vil vi arbejde med fysiksimulatoren `Pybox2D`. `Pybox2D` simulerer blandt andet kollisioner og kontakter imellem objekter, og løser de derilhørende restriktioner iterativt ved at starte fra nul og så trin for trin bevæge sig tættere på en løsning. Vores mål er at bygge en model som kan hjælpe simulatoren ved at forsyne den med startværdier som er bedre end bare at starte fra nul, altså startværdier som er så tætte på den endelige løsning som muligt, i håbet om at den iterative process derved færdiggøres hurtigere. Vores model vil være et såkaldt convolutional neural network, som vi vil designe og bygge i `TensorFlow`. Vi vil også være nødt til at designe og implentere en metode til at overføre data fra simulatoren til vores model, og tilbage igen. Alt dette har vi gjort, og vi har så målt effektiviteten af simulatoren når den bruger vores model. Resultaterne er ikke overvældende. Vores model er i stand til at startværdier der fører til bedre resultater i forhold til hvad man får ved at starte fra nul, men den har ikke været i stand til at generere bedre værdier end den metode der allerede er indbygget i simulatoren, hvilket er en metode der bare benytter tidligere resultater som startværdier.

Contents

1 Introduction

In recent years machine learning methods have successfully been applied to many new areas, including areas such as physics, simulations and animations. One specific area that has not yet benefited as much from these new methods is rigid body simulations, due to difficulties in applying deep learning. This is what we hope to remedy to some small degree in this thesis, focusing specifically on the particularly difficult part of solving contacts between rigid bodies. Rather than entirely replacing the traditional way of solving contacts for rigid bodies, our approach in this thesis will be to examine a way of combining traditional, precise contact force solvers with modern, fast machine learning methods, or more specifically; convolutional neural networks.

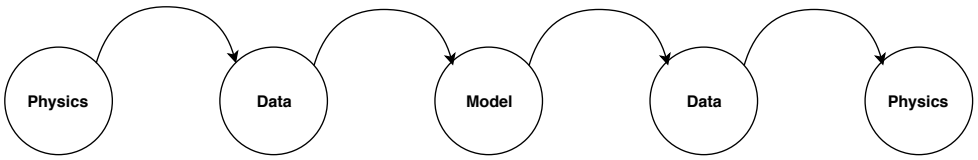
Put briefly, traditional contact solvers work by starting from zero and then iteratively getting closer and closer to a solution step by step. Our goal is to design a neural network which can take as input information about the simulation world, and then ouput values which we can use as starting iterates for the contact solver, in the hopes that this will help the iterative process reach a result in fewer steps.

In order to connect the physics simulator with our neural networks, we will also need a method of transfering data from one to the other and vice versa. Convolutional neural networks typically takes images as input, or grids, and so we will design a method for transfering physics data from the simulator to a set of grids that our neural network can use, and similarly a way of transfering the values from the grids that our neural network outputs and back to the simulator.

Before diving headlong into our project, we will talk about a few projects similar to ours that other people have worked on in section 2. We will then start off in section 3 by introducing some of the physics that our simulator will be simulating, and that we will need a basic understanding of for this project. Following that we will introduce the specific physics simulator that we have chosen to use, called `Pybox2D`, in section 4, and talk a little about how it works. With the physics out of the way, in section 5 we will then look at the first part of the work that we have done; designing and implementing a method for transferring data from the simulator to our model, and back, using grids. In section 6 we will introduce neural networks and talk about things like neurons, layers and convolutions, and in section 7 we will then take a look at the specific neural network models that we have designed and implemented. In section 8 we will start by discussing how we have tested our models, and some simple models that we have built for comparison, and then we will take a look at the results of testing our neural network models

and their performance and talk a little about these results. Finally in section 9 we will then sum up our results and give some final remarks. As a bonus for the interested, we will introduce our code in Appendix A, and provide a little guide on how to run it and where to find it.

Throughout the thesis we will repeat the illustration below, and color some parts and change others to indicate where in the process we are.



Two other fellow computer science students, Jian Wu and Lucian Tirca, worked on similar master thesis projects alongside myself, and we shared information and in some cases worked together on some of the initial parts of the project, such as understanding and modifying the `Pybox2D` simulator and generating training data. The main part of the project, the convolutional neural network, has been designed and implemented entirely be me though, and the various results that will be shown throughout this thesis are all my own.

2 Similar Work

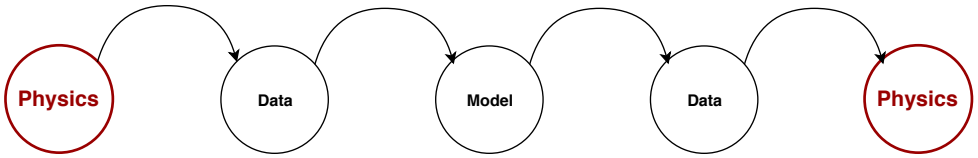
Machine learning is currently a very hot topic, which people are attempting to use for all kinds of problems. In [GTH98] the authors introduce what they call *NeuroAnimators*, a type of model which is supposed to replace the classical numerical kind of simulation and animation with neural networks trained to do the same but much faster. The model is mainly focused on animation, and is capable of not just animating a specific situation moving forward in time but also of producing a simulation satisfying a number of predefined animation goals. The idea of combining existing methods and tools with the power of machine learning is also something that is being explored in many fields. In [BPL+16] the authors present what they call *Interaction Networks*, which are models that are intended as "general-purpose, learnable physics engines", and which attempts to reason about objects and interactions between objects similar to how a human can do it. They are based on a number of techniques, including deep learning in the form of neural networks, and can deal with simulating various physical domains such as collisions between rigid bodies. Taking it a step further, the authors of [WTW+17] create what they call *visual* interaction networks, which use a fairly complex combination of convolutional neural networks, recurrent neural networks and interaction networks. The goal is to create a model which takes video data of objects and their movement and behavior as input, decode the video data into useful information and then simulate the expected future behavior of said objects. Using supervised learning, the model is then supposed to learn everything it needs to know about objects and their relations through training.

The field of robotics is also being included in the mix, where for instance in [JL18] their aim is to improve contact modeling for rigid bodies by combining analytical solutions with neural networks. The idea is to create a model which solves well known and understood scenarios analytically, such as the contact forces preventing interpenetration, and then uses a number of neural networks, specifically classifiers and regressors, to handle the less understood or more complicated situations, such as friction. This could then be used in improving robots and their behavior.

If you have a lot of bodies piling up, the behavior of the system starts to resemble that of a fluid, which is another area that has seen a lot of use of deep learning. For instance, in [TSSP16] the authors combine traditional methods for solving the Navier-Stokes equations for fluid flow with machine learning. They design a model which includes a convolutional neural network, which amongst other things is used to simulate long range influences and large scale behavior. Their model also uses grids to represent the various quantities of the fluids, and is trained using unsupervised learning.

3 Physics and Simulation

First things first, we will need to talk a bit about physics. In order to get an idea of what we are trying to model using our neural network, we need to understand how our simulator works, and for that we need to know a bit about the physics that the simulator is trying to simulate. When we say the simulator is simulating physics, what we actually mean is that it is simulating bodies and their behavior, as they are influenced by various factors, be it body-to-body interactions such as collisions, external forces such as gravity, etc. All of these factors are simulated using various laws and equations of physics, and so we need to take a look at these. Another reason is that the input to our model will be various physical quantities of the objects we are simulating, and the output from the model will be another set of physical quantities that we will feed back into the simulator, so we really need to know a bit about these quantities.



This section is intended as a relatively simple brush up on the rudimentary physics we need for our simulator, aimed at fellow computer science students who might not have worked with physics for a while. The goal is simply to achieve a basic understanding of what the simulator is doing, and as such the focus is going to be on the intuition about physics more than on stringent math and complex derivations. For a more rigorous and in-depth explanation of the physics involved, we recommend looking at literature written for that purpose, such as [ESHD05] which we used as a reference. In this section we are also going to assume that the reader is familiar with basic university-level math concepts, such as vectors, matrices, derivatives etc. There should not be anything too complex math-wise, and neither are there going to be any nasty proofs.

We will start up by talking about the basics quantities associated with objects in section 3.1, such as mass, position, velocity and acceleration, and we will also talk about some of the forces that influence these. With that out of the way, in section 3.2 we will then talk about what happens when objects

collide, how that is handled in a simulator, and perhaps most importantly; what impulses are. Let us start with the basics.

3.1 The Basics

First of all, one important assumption that we will be making throughout this thesis is that all objects behave like *rigid* bodies. What this means is that when an object is created with a given shape, it will retain that shape forever no matter what happens to it. An object might be rotated, translated etc, but for instance it will not suffer any deformation due to collisions. Another way of defining it is to says that if you choose any two points on or inside the object, the distance between them will always remain the same. This assumption makes a lot of things simpler. Another important point is that we will only be working with two dimensions, which also simplifies a lot of the physics. With these assumptions out of the way, let us take a look at some of the physical quantities that we will be working with.

3.1.1 Mass

One of the most basic yet important attributes an object has is its *mass*. An objects mass depends on its *density* and its *volume*, or in our two-dimensional case its *area* as two-dimensional objects have no volume. The volume, or area, of an object tells us how much physical space the object occupies, and so depends on the geometry of the objects. For complex objects this might be very difficult to determine, where as for simple geometric shapes there are often simple formulas that can be used. As an example, the area of a circle is given by

$$A = \pi r^2 \tag{3.1}$$

with r being the radius of the circle.

The other factor that determined the mass of an object was the density, which unsurprisingly tells us something about how dense an object is, i.e. its mass per unit area in our two-dimensional case. The density depends on different factors such as what sort of material the object is made of, with some materials like metals being very dense and other materials like wood being less so. The density also depends on the temperature of the object, but we are going to ignore that part. An object could also have different densities for different parts of the object, making the calculation even more complicated, and so we are going to assume uniform density.

Knowing how much space an object occupies from having determined its area, and how much mass the object has per unit space from its density, we can then multiple the two to get the total mass of the object

$$m = A \cdot \rho \tag{3.2}$$

with m being the mass, A being the area and ρ being the density.

3.1.2 Position and Movement

Another important attribute of any object is its *position*. The position tells us where in the world the object is, and is typically represented as a set of coordinates in some form of *world coordinate system*, or WCS, which is just an inertial coordinate system with some fixed origin that we have chosen. If we call our position vector $\boldsymbol{r}$, where the letter is written in bold to indicate that it is a vector, and we remember that we are working with two dimensions, then we can write the position as

$$\boldsymbol{r}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \tag{3.3}$$

with x and y being the position of the object along the two axes as usual. The exact values of x and y of course depends on the position of the origin our world coordinate system. The position of an object is rarely a static thing though, often changing with time as the object moves around for one reason or another, and so we typically consider the position to be a function of time as indicated in equation 3.3.

This change in position as a function of time is another important attribute which should be familiar to most people. We call it *velocity*, with symbol $\boldsymbol{v}$, and write it as

$$\boldsymbol{v}(t) = \frac{d\boldsymbol{r}(t)}{dt} = \begin{bmatrix} \frac{dx(t)}{dt} \\ \frac{dy(t)}{dt} \end{bmatrix} = \begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix} \tag{3.4}$$

where similarly v_x and v_y are the velocities of the object along the two axes. The velocity is a vector, which has a direction that tells us something about where the object is moving, as well as a magnitude which tells us something about how quickly it is moving in said direction. The magnitude, or length, of the velocity vector, which is of course a scalar, is what we would typically call the *speed* of the object.

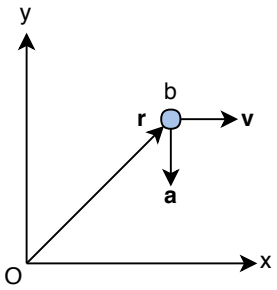


Figure 1: An illustration of a moving body b, with a position $\boldsymbol{r}$, a velocity $\boldsymbol{v}$ and an acceleration $\boldsymbol{a}$, in some WCS.

Velocity is not a static thing either, also often changing over time due to external influences, and this change in velocity as a function of time is what we call *acceleration*, with symbol $\boldsymbol{a}$, and define as

$$\boldsymbol{a}(t) = \frac{d\boldsymbol{v}(t)}{dt} = \begin{bmatrix} \frac{dv_x(t)}{dt} \\ \frac{dv_y(t)}{dt} \end{bmatrix} = \begin{bmatrix} a_x(t) \\ a_y(t) \end{bmatrix} \tag{3.5}$$

where again a_x and a_y are the accelerations of the object along our two axes. The acceleration is also a vector, and it has a direction which tells us something about what direction the velocity is changing towards, which in turn tells us something about what direction the object will move in the future. It also has a magnitude, which tells us something about how quickly the velocity is changing in said direction, and which in turn tells us something about how quickly the object will start moving in said direction.

An example of these three concepts is shown on Figure 1, with an object moving horizontally while accelerating downwards due to some external force such as gravity. This example could for instance represent a ball being thrown horizontally, or a bullet being fired from a gun, while being influenced by gravity. The next thing we need then, is a way to determine the acceleration of an object.

3.1.3 Momentum and Force

Putting aside acceleration for a second, we consider another interesting quantity called *momentum*, which we get from combining mass and velocity. The momentum of an object tells us something about how difficult it is to change the velocity of the object, i.e. how much force it takes. We typically assign

the letter p to the momentum, and define it as

$$\boldsymbol{p}(t) = m\boldsymbol{v}(t) = m \begin{bmatrix} v_x(t) \\ v_y(t) \end{bmatrix} = \begin{bmatrix} p_x(t) \\ p_y(t) \end{bmatrix} \tag{3.6}$$

where again p_x and p_y are the momentums of the object along our two axes. We note that the momentum is also a vector, which means that it has a direction which will be the same as for the velocity, as well as a magnitude which is simply the magnitude of the velocity vector scaled by the mass of the object.

How difficult it is to make the object move in another direction then depends on the difference between the new direction and the direction of p, as well as the magnitude of p, and changing an objects course will similarly change the momentum vector. In fact, the change in the momentum vector over time is one way that we can define the *force*, with symbol F, acting on an object

$$\boldsymbol{F}(t) = \frac{d\boldsymbol{p}(t)}{dt} = \begin{bmatrix} \frac{p_x(t)}{dt} \\ \frac{p_y(t)}{dt} \end{bmatrix} = \begin{bmatrix} F_x(t) \\ F_y(t) \end{bmatrix} \tag{3.7}$$

where again F_x and F_y are the forces acting on the object along our two axes. The force is another vector, whose direction tells us where we are trying to push things, and whose magnitude tells us how hard we are pushing.

Using our definition of momentum from equation 3.6 and acceleration from equation 3.5, we can also write this as

$$\boldsymbol{F}(t) = \frac{d\boldsymbol{p}(t)}{dt} = m\frac{d\boldsymbol{v}(t)}{dt} = m\boldsymbol{a}(t) \tag{3.8}$$

which tells us that applying a force to an object results in the object being accelerated in the direction of the force, with the acceleration having a magnitude that depends on the magnitude of the force and the mass of the object.

One might wonder then, how to determine the effect of multiple forces being applied to the same object. Luckily this is very simple to handle, as Newton tells us that we can simply sum up the different forces and treat them as one

$$\boldsymbol{F}_{tot}(t) = \boldsymbol{F}_1(t) + \boldsymbol{F}_2(t) + ... + \boldsymbol{F}_n(t) = m\boldsymbol{a}_{tot}(t) \tag{3.9}$$

with $\boldsymbol{a}_{tot}(t)$ then being the resulting acceleration of the object. This in turn also tells us that if we know all of the external forces acting on an object, we can determine the total acceleration of the object

$$\boldsymbol{a}_{tot}(t) = \frac{\boldsymbol{F}_{tot}(t)}{m} \tag{3.10}$$

From acceleration we can get velocity, and from velocity we can get positions, and so we are starting to collect some equations for our simulator to use. There is still an important aspect missing though, as everything we have done so far has been rather linear, and that is of course rotations.

3.1.4 Rotations

Similarly to how an object has mass, a position, a velocity, momentum and so on, it also has what is called rotational mass, an angle, angular velocity, angular momentum and so on. These quantities are generally a lot harder to determine and to work with, and also a lot harder to understand as they are generally far less intuitive. They are also not going to play too big of a role in the kind of simulations we will be running, and so we will go into less detail with these quantities. They are worth mentioning though, as they are a part of the physical world we are trying to simulate.

Before we actually get to rotations, we need to introduce a few more quantities. When considering the shape of an object, we can imagine finding a sort of geometric center of the object, by doing some sort of averaging of the distances and dimensions involved. For instance, in the case of a circle, the center would of course be at the actual center of the circle. Similarly, knowing the mass and shape of the object we can determine an important quantity called the *center of mass*, which plays a role in things like rotations and balance. If we consider splitting our object into a very large number of very small pieces, say n pieces, with each piece having a position $\boldsymbol{r}_i$ and a mass m_i, then the center of mass, $\boldsymbol{r}_{cm}$, is given by

$$\boldsymbol{r}_{cm} = \sum_{i=1}^{n} m_i \boldsymbol{r}_i \tag{3.11}$$

One might image that the $\boldsymbol{r}_i$ vectors could change over time, which in turn would mean that the center of mass would change over time, but this is not the case in our simulation, as we are working with rigid bodies. Once again taking a sphere as the example, depending on the mass distribution of the sphere the center of mass would then lie somewhere inside the sphere.

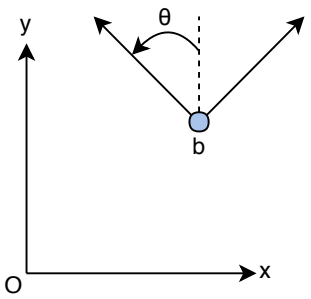


Figure 2: An illustration of a body b, including the body frame, with a positive orientation, or θ.

If the mass distribution was constant across all of the sphere, as we will be assuming, then the center of mass would be at the center of the sphere.

Another quantity that an object has is *orientation*. In order to define the orientation, we first need to assign each object their own, local coordinate system, sometimes called the *body frame*. This is a coordinate system with the same number of axes as the world coordinate system, but with the origin placed at the center of mass of the object. In this way, the body frame of an object moves with the body, and working with rigid bodies this further means that the body will never move or change in its own body frame.

The orientation of an object then says something about the angles between the axes of the world coordinate system and the body frame. In three dimensions we need tree different angles to describe the orientation, where as in two dimensions we only need one, which we shall call θ. If the body frame axes are parallel to their counterparts in the world coordinate system, we say that the θ is zero. Otherwise θ is equal to the angle by which the body frame is rotated compared to the WCS, with counterclockwise rotation considered positive and clockwise rotation considered negative. Of course, as is always the case when dealing with angles, one can just add or remove 2π to the angles without changing the situation, and so generally θ is chosen to be between 0 and 2π or between $-\pi$ and π, with the unit of course being radians.

Similarly to position, orientation is of course also something that changes with time, and this change in orientation over time is what we call *angular velocity*, symbol ω, defined as

$$\omega(t) = \frac{d\theta(t)}{dt} \tag{3.12}$$

The orientation was a scalar, and so the angular velocity will of course also be a scalar, which means that it has no direction. Again, a positive sign would mean that the orientation is increasing, i.e. the body frame is rotating counter-clockwise, and a negative sign means clockwise rotation. Similarly, a change in angular velocity over time is called *angular acceleration*, symbol α, defined as

$$\alpha(t) = \frac{d\omega(t)}{dt} \tag{3.13}$$

and it too is simply a scalar, with the same convention for the sign. Like the linear case we also have something called *angular momentum*, symbol L, defined as

$$L(t) = I\omega(t) \tag{3.14}$$

with I being the so-called *moment of inertia*, also sometimes called rotational mass as it plays a similar role to mass in the rotational equations. Similarly to how an objects mass says something about how much effort it takes to change the objects velocity, the moment of inertia says something about how much effort it takes to change the objects angular velocity. If we once again imagine cutting our object into n small pieces, the moment of inertia is given as

$$I = \sum_{i=1}^{n} m_i ||\boldsymbol{r}_i||^2 \tag{3.15}$$

and similarly to the mass, the fact that we are working with rigid bodies means that the moment of inertia does not change over time. Finally, similarly to how force was defined as a change in linear momentum over time, for rotations we have a quantity called *torque*, symbol τ, which is defined as a change in angular momentum over time

$$\tau(t) = \frac{dL(t)}{dt} \tag{3.16}$$

which again is a scalar in two dimensions, with the same convention for the sign of the quantity as before. Torque is something that occurs as a result of

one or more forces affecting the object, and so if we know the forces affecting an object, we can determine the resulting torque as

$$\tau(t) = \boldsymbol{r}(t) \times \boldsymbol{F}(t) = r_x F_y + r_y F_x \tag{3.17}$$

and this in turn allows us to determine the resulting angular acceleration as

$$\alpha(t) = \frac{\tau(t)}{I} \tag{3.18}$$

With this we should have most of what we need to get our simulator running.

3.1.5 Simulating Movement

In the real world, time is a continuous thing and movement happens nicely and smoothly. If we want to simulate movement and physics on a computer though, we will need to turn everything into discrete quantities due to the nature of how computers work. In particular, we will need to cut time up into discrete pieces, $t_1, t_2, ..., t_n$, with a set distance between each step, Δt. Our simulator will then do one time step at a time, by looking at what information is available at the current point in time and using this to determine how everything should look after the next time step.

Let us consider a single object at a time t_i, and assume that we know its position and orientation, its linear and angular velocity, as well as what forces are affecting it. Knowing the forces affecting our object we can then determine the total force acting on the object using equation 3.9, which in turns gives us the total acceleration of our object at time t_i using equation 3.10. Knowing the acceleration and velocity of our object at time t_i, we can then calculate what the velocity will be at time t_{i+1}

$$\boldsymbol{v}(t_{i+1}) = \boldsymbol{v}(t_i) + \boldsymbol{a}(t_i) \cdot \Delta t \tag{3.19}$$

and using our knowledge of its position and velocity at time t_i we can then determine its position at time t_{i+1}

$$\boldsymbol{r}(t_{i+1}) = \boldsymbol{r}(t_i) + \boldsymbol{v}(t_i) \cdot \Delta t \tag{3.20}$$

Similarly, knowing the total force we can determine the total torque using equation 3.17, which then gives us the total angular acceleration using equation 3.18. Knowing the angular velocity and acceleration at time t_i we can then determine the angular velocity at time t_{i+1} as

$$\omega(t_{i+1}) = \omega(t_i) + \alpha(t_i) \cdot \Delta t \tag{3.21}$$

and knowing the orientation and angular velocity at time t_i we can determine the orientation at time t_{i+1} as

$$\theta(t_{i+1}) = \theta(t_i) + \omega(t_i) \cdot \Delta t \tag{3.22}$$

We then have most of the quantities we need to continue with the next step of our simulation, the only thing we don't know is what forces are affecting the object at time t_{i+1}. Some of these will be easy to identify and calculate, such as gravity, while others like the forces involved in collisions between objects will be more difficult to determine.

3.1.6 Examples of Forces

One constant and hard to avoid force that should be familiar is *gravity*, $\boldsymbol{F}_g$. Assuming that we are working on a sufficiently small scale, we can approximate the surface of the earth as being flat. If we then choose our WCS such that the x-axis is pointing along the surface of the earth, and the y-axis directly away from the center of the earth, perpendicularly to the surface, we can approximate gravity as a constant force, pointing in the direction of negative y with a magnitude of roughly 10 $kg \cdot m/s^2$. This is a very simple and easy-to-understand force, which in simple simulations like the ones we will be doing will be the main, dominating force. As for the reason why gravity exists and behaves as it does; I will leave that up to the reader to look up at their own discretion.

Let us then consider the situation shown in Figure 3, where we have one object at rest on top of, and in contact with, another object. Let us imagine that it is a ball on top of a table, and let us focus on the ball. The ball will of course still be affected by gravity, which will attempt to pull it downwards. Due to more complex laws of physics though, solid object are generally not able to pass through one another, preventing the ball from moving. Instead, the force of gravity affecting the ball will be pushing against the table. Another thing that Newton told us, is that whenever one object

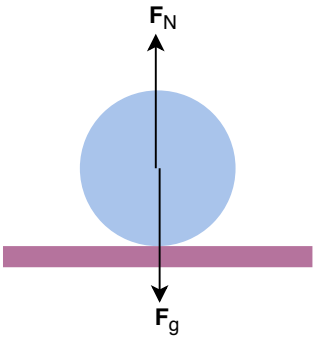


Figure 3: An illustration of one object resting atop another. The top most object is being affected by a force due to gravity, $\boldsymbol{F}_g$, and the bottom most object is pushing back with a normal force, $\boldsymbol{F}_N$. The vectors are slightly offset to make them easier to discern.

is affecting another with a force, the second object will be affecting the first with a force with similar magnitude and opposite direction. This tells us that the table will be pushing back on the ball with a similar force, only pointing upwards instead of downwards, and this force is called the *normal* force, $\boldsymbol{F}_n$. We can write this as

$$\boldsymbol{F}_N = -\boldsymbol{F}_g \tag{3.23}$$

This force will be important in the later stages of our simulation, when we have a lot of objects piling together, and it is also important for determining another important force, namely *friction*.

Friction is a force that is occurs whenever two objects are in contact, and that generally works towards preventing the objects from moving. Let us consider Figure 4, which is similar to the previous figure except the bottom most object, the table, now has a slope. In this case we might expect the ball to start sliding down the slope, but it is still at rest, and the reason is friction.

The ball is of course still affected by gravity, but this time the force is not completely perpendicular to the table. For this reason, let us imagine that we split the force of gravity into two components, one that is perpendicular to the table, $\boldsymbol{F}_\perp$ and one that is parallel to the table, $\boldsymbol{F}_\parallel$. The perpendicular component will then be the force that is pushing against the table, resulting once again in a normal force with equal magnitude and opposite direction

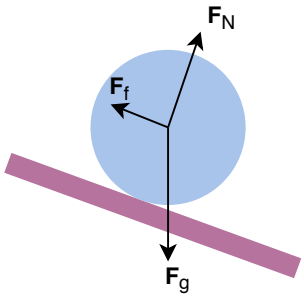


Figure 4: An illustration of one object resting atop another. The top most object is being affected by a force due to gravity, $\boldsymbol{F}_g$, part of which is pushing against the bottom object and being countered by the normal force $\boldsymbol{F}_N$, and part of which is pushing the object towards sliding down the slope and being countered by the friction force $\boldsymbol{F}_f$. The magnitude of the vectors are chosen to illustrate a point, and not to say anything about the actual magnitudes of the forces involved.

being exerted by the table

$$\boldsymbol{F}_N = -\boldsymbol{F}_\perp \tag{3.24}$$

The parallel component on the other hand will have a direction that points down the slope, and will be attempting to force the ball to accelerate in said direction, which in turn would cause the ball to move down the slope. In the case where the ball is at rest and not sliding down the slope, this tells us that

$$\boldsymbol{F}_f = -\boldsymbol{F}_\parallel \tag{3.25}$$

which means that the total force and equivalently total acceleration of the ball is zero.

It will of course not always the case that the ball is at rest and all forces equal, so how do we actually determine the friction force? For this we will need to briefly talk about one of the topics of the next section in advance, namely *contacts*. In simple cases like our example with the ball and the table, we will typically consider the contact between the two objects to be a single point. Of course, in reality that is not exactly the case, rather there will be some area where the two objects are in contact. Let us assume that this area is flat, and call it a contact *surface* or *plane*. In our example, this plane would be parallel with the surface of the table. In general, the friction

force will then be parallel to this contact plane, and point in the opposite direction of the component of the total external force affecting the ball that is parallel to this plane, the $\boldsymbol{F}_{\parallel}$ in our example. As for the magnitude of the friction force, we get an upper limit from the normal force

$$||\boldsymbol{F}_f|| \leq \mu ||\boldsymbol{F}_N|| \tag{3.26}$$

with μ being the so-called *coefficient of friction*. This is a coefficient that is dependent on a number of factors, such as what materials the objects in contact are made of, and which typically also has a different value when the objects are at rest, called static friction, compared to when they are moving relative to each other, called kinematic friction. In the case of no force component acting in the contact plane, the friction magnitude will be zero. In the case where there is some component in this plane, the magnitude of the friction force will grow to match this force if possible, maxing out at the value determined using 3.26.

Now, in reality calculating friction forces will often be a lot complicated than what we have described, but we only really need a basic understanding as the simulator will be doing the work for us. Having already taken a sneak peak at the topic of contacts then, let move on to this topic in earnest.

3.2 Contacts and Collisions

In the previous parts we talked about the basics of rigid bodies; some of the attributes they have, how they move around and what makes them move around. All that we are missing then is to talk about what happens when they collide.

In the real world, where time is continuous, it's actually a rather complex process, which depends a lot on the objects involved and their properties. For instance, when two objects come into contact, at the area of contact we will likely see both object being subjected to some level of deformation, i.e. their shapes will change. In some cases, we might even see one or both objects being destroyed and being shattered into pieces by the impact. We might also see one object stop moving after the impact, like a rock being dropped on a floor, or we might see the object bounce back after the collision, like a rubber ball hitting a floor.

When simulating collisions, we are going to have to simplify things a lot, at least for the level of simulator that we will be using. One assumption that we have already made that will help us is that we are dealing with rigid bodies. This tells us that the shape of our objects will stay constant,

i.e. that no deformation will take place. The collisions will then typically be resolved using a couple of laws focusing on conservation, such as conservation of energy, involving primarily kinetic and thermal energy, and preservation of momentum.

An interesting quantity is the so-called *relative* velocity of the two objects. If one object has velocity $\boldsymbol{v}$ and the other has velocity $\boldsymbol{w}$, then the relative velocity, $\boldsymbol{u}$, is simply defined as

$$\boldsymbol{u}(t) = \boldsymbol{v}() - \boldsymbol{w}(t) \tag{3.27}$$

In this way, if we know the velocities of the objects before the collision, we can determine the relative velocity before the collision, and similarly if we know the velocities of the object after the collision we can determine the relative velocity after the collision. If we then divide the relative velocity after the collision with the relative velocity before the collision, we get something called the coefficient of *restitution*. A collision with a coefficient of restitution of one is called a perfectly elastic collision, and means that the objects will have the same velocities after the collision as they had before, only the signs have changed. A collision with a coefficient of restitution of zero is called a perfectly elastic collision, and means that the objects will not move at all after the collision.

Now, typically the coefficient of restitution is something we will know beforehand, either based on the objects involved and what material they are made of, or simply by having set it ourselves when creating the object in our simulator. We can then determine what the velocities of the involved objects should be after a collision by using the coefficient together with the two preservation laws mentioned earlier. For our simulations, we will be setting the coefficient of restitution to zero in order to simply things and in order to prevent too much bouncing and instead encourage settling.

With all of this in mind, we can now take a look at what we would expect from a collision. Imagine that we have two objects that are approaching each other, with whatever velocities they have. They come into contact, their post-contact velocities are determined, and then they continue moving without any issues. If time was continuous it would be a smooth process, with the velocities being determined instantaneously. But again, time is not continuous in our simulator, and so it will not be quite as simple a process. In the simulator time is sampled into small steps, and the odds that two object come into contact exactly when the simulator has taken a step are quite low. What we will see instead is that two object will be approaching

each other, each with some velocity, then we simulate a step in time, and then we see the two objects not just being in contact but actually overlapping, or *interpenetrating*. This is a result of the objects coming into contact during the step forward in time, and so being allowed to continue moving despite it. Two or more objects occupying the same space is of course not very realistic, and so this is something we need to handle.

Let us imagine that we have taken a step, and done everything related to that in terms of forces, accelerations, velocities and positions, and we then have one or more cases of objects overlapping. We will handle this in two steps; first by fixing the velocities and secondly by fixing the positions.

3.2.1 Fixing Velocities

As we talked about previously, we know that in the real world changes in velocities are a result of accelerations, and similarly accelerations are a result of forces, and all of this happens over time. What we could do then would be to determine and apply some kind of contact force, which in turn would result in an acceleration that would then modify the velocities of the objects in the way that we wanted. This would require the simulator to simulate even more time though, and meanwhile we would have objects occupying the same space in a very unrealistic manner. What we instead need is a way to change the velocities instantaneously, and for this we will use something called *impulses*, symbol J, defined as

$$\boldsymbol{J} = \int \boldsymbol{F} dt \tag{3.28}$$

An impulse is equivalent to continually applying a force to an object over a time period, which would mean accelerating the object, only an impulse is applied instantaneously, which is exactly what we wanted. Another way of defining it is to say that applying an impulse results in an immediate change of momentum, which in turn means a corresponding immediate change in velocities as masses are constant, and this is what we will use to fix the velocities.

Now, our simulator is not actually going to be determining any forces when calculating impulses, instead it will look at relative velocities. If we consider two objects that are overlapping, we can determine their relative velocity as described previously. As the objects are in contact, we will also have a contact normal, and we can then split the relative velocity into two components; one that is parallel to this contact normal, the *normal* component, and one that is perpendicular to this normal, the *tangent* component.

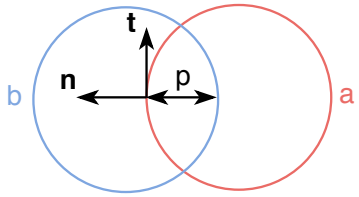


Figure 5: An illustration of two objects, a and b, overlapping, with n being the contact normal, t the contact tangent and p the interpenetration.

The most important thing then, to prevent the objects from colliding and overlapping again, is to change the velocities such that the normal component of the relative velocity becomes nonpositive. This is because a nonpositive normal component of the relative velocity tells us that the two objects are moving apart, in the case of a negative value, or moving in parallel, in the case of a zero. The actual value that we want of course depends on the relative velocity before the collision and the coefficient of restitution. The case of a coefficient of restitution of zero is particularly simple, as we can then simply determine the impulse as

$$\boldsymbol{J} = -(m_1 + m_2)\boldsymbol{u} \tag{3.29}$$

with m_1 being the mass of one object, m_2 being the mass of the other and $\boldsymbol{u}$ being the relative velocity. All we have to do then is apply the impulse to both objects, taking things like masses, radiuses and signs into consideration, and then the two objects should not collide during the next step, at least if we ignore outside forces acting on the bodies.

3.2.2 Fixing Positions

That brings us to the second part, which is fixing the positions such that the objects are no longer overlapping. The first thing we do is to determine the actual interpenetration, by considering the two objects and their respective shapes. This will be a vector, which we can then similarly split into a normal and a tangential part. All we need then in order to fix the interpenetration is to fix the normal part, assuming we are working with simple shapes like squares and circles.

This is actually very simple once we know the actual interpenetration values, since all we do then is to move the objects by simply changing the positions of the objects accordingly, taking into account that heavier objects should probably be moved less than lighter ones.

Now, there is one thing that we have been ignoring so far in our discussion of contacts and collisions, and that is once again rotations, i.e. the objects' orientations, rotational velocities etc. These are things that also needs to be taking into account when dealing with contacts and collisions, and as before there are similar equations and reasonings for how to handle this, including the use of the impulses we talked about earlier. Once again though, everything is a little more complicated when dealing with rotations, and in our case it will not play a very big role, and so we have chosen not to go into anymore details about it.

3.2.3 Multiple Contacts

So far we have only been concerned with a single collision and contact between a pair of objects, but in reality we will likely have many contacts and collisions between many pairs of objects simultaneously, and we might even have some objects that are in contacts with multiple objects at the same time. This will require us to be a little more careful with the techniques described above.

Let us imagine that we have an object, a, which is colliding with another object b to the left of a, and simultaneously colliding with a third object c to the right of a. Let us say that we solve the collision between a and b first, resulting in a being moved slightly to the right. Next thing we will do is then to solve the collision between a and c, which we might imagine results in a being moved slightly to the left. This is obviously an issue, since a will now likely be overlapping with b once again. If we then solve for the collision between a and b again, the result might similarly cause a and c to be overlapping again, and so on. A similar thing might also happen for the velocities when we are trying to solve the two collisions.

One solution might be to consider and solve both contacts simultaneously, but that seems a little complicated, and it quickly becomes very complicated when adding more objects. What we will do instead is to simply accept that we can not solve everything in one go, and then just do multiple passes over all contacts. What we mean by this is that we will solve every single contact once, one contact at a time, using the methods described above. We will then take a look at all of the contacts once again, to see if there are any issues left, and if so, we will solve all of the contacts once again. We then repeat this procedure, until there are no more issues left, or until the remaining issues are at least so small that we can safely ignore them. For positions, we will consider the interpenetration for each contact, and only stop iterating when it is below some chosen threshold, which might or might not be zero.

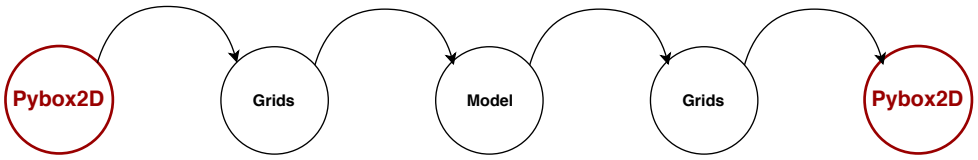
Similarly for the velocities we will keep iterating until the impulses we are calculating are below some chosen threshold.

The procedure above is of course a very simplified version of how to handle contacts and constraints, which relies on a lot of assumptions and simplifications. In most cases, the whole thing will be a lot more complicated, and as such one will have to use more complicated methods for solving the contacts, such as setting certain constraints for the relative velocity and the impulses and then solving the whole thing as a linear complimentary problem. These methods can be both clever and actually kind of nice, but they are also a lot more complicated. For this project we just need a basic idea of what a contact is, how to solve it and what role impulses play, in order to understand what we are trying to train our model to predict, and so for a more thorough discussion we refer the reader to the plethora of literature on the subject.

Now, having a basic understanding of the physics, let us move on to the simulator that we have chosen for simulating said physics.

4 The Pybox2D Simulator

The physics simulator that we have chosen to use is called `Pybox2D` [Lau08], and is basically just a `Python` wrapper around the `Box2D` simulator [Cat06] written in `C++`. We chose this simulator because it is small, lightweight and relatively simple. We chose the `Python` version because it makes it easier for us to connect it to our model, which is build with `TensorFlow` and also running in `Python`. Finally we chose this simulator because we have full access to the source code, allowing us to make any changes we see fit.



The simulator was originally developed at a game developer conference, called *GDC*, as a tutorial in how to do physics when developing games and engines for games. As such, its focus is on simplicity, performance and 'realism', and not so much on doing actual, accurate physics calculations. It does work with floating point numbers, but only 32 bits numbers as opposed to 64 bit or better. The simulator is built to work with meters, kilograms and seconds, which means that if the distance between two objects in the simulator is 1, it simulates a distance of 1 meter, if an object has a weight of 1 it actually means 1 kilogram and so on. As the name implies, `Pybox2D` only simulates physics in two dimensions.

4.1 Classes

The simulator works with five main classes; *bodies*, *fixtures*, *shapes*, *contacts* and *joints*. We will not be talking about, or working with, joints in this thesis. A *body* is an object with a position and a velocity, an angle and an angular velocity, a type, which we will return to later, as well as various coefficients and flags which specifies certain behavior. A body does not manage its own mass or shape, this is what fixtures and shapes are for. A *shape* is, as the name applies, an object which spans an area of space, with a specific shape. `Pybox2D` works with a few different shapes, the main ones being circles and polygons. Shapes are used to determine when bodies collide or even penetrate, as well as when drawing bodies. Shapes do not have any mass or density though, and do not know anything about bodies, this is

what fixtures are for. A *fixture* is what connects a shape to a body, and stores information about friction, restitution and density. A body can have multiple fixtures, each with their own shape, but for the sake of simplicity we will be working with bodies with a single fixture and a single shape. The mass of a body can then be determined using the density of its fixture and the surface area of its shape. From now on we will simply be referring to bodies as objects with all the expected attributes including mass and shape, rather than distinguish between bodies, fixtures and shapes. All of the mentioned attributes are represented as 32 bit floats, or vectors thereof, in the simulator.

The simulator works with three different types of bodies; *dynamic*, *kinematic* and *static*. A *dynamic* body is the standard body type, which moves, collides and generally behaves as you would expect from a physical body being affected by various forces and interacting with other bodies. This is the body type that will typically be used for most bodies in a simulation. A *static* body has no velocity and ignores any forces, and as such does not move during a simulation unless the user manually moves it. A static body does not collide with other static bodies or with kinematic bodies, only with dynamic bodies. For such collisions static bodies are treated as if they have infinite mass, which means that only the dynamic body will be affected by the collision. Static bodies are typically used to represent solid objects like a floor, a ceiling or a wall, which is expected to be unchanging and used to limit the simulation world in some way. Finally, a *kinematic* body is one that moves only according to its set velocity and thus ignores any and all forces. Kinematic bodies does not collide with other kinematic bodies, and neither do they collide with static bodies. They do collide with dynamic bodies though, in which case they are treated as having infinite mass as was the case with static bodies. We will not be using kinematic bodies in our simulations.

4.2 The Simulation Loop

In order to make the simulator actually simulate things, you have to tell it to take a step forward in time, telling it exactly how large of a time step to take. The simulator then takes a step forward by calculating forces and accelerations, which it uses to determine velocities and positions after the time step as described in section 3. The next thing it does then is to detect and handle contacts. The simulator does actually have built-in functionality for detecting and handling contacts when they actually happen during a timestep, which would prevent interpenetration, but we have disabled this functionality to keep things simpler.

When solving contacts, the simulator then separates the contacts into what it call *islands*, which are essentially just different connected graphs of contacts. What this means is, that if body A is in contact with body B, and body B is in contact with body C, then A, B and C make up an island. If we then had a body D in contact with body E, but with neither being in contact with any of the first three, then D and E would make up another island. When solving the contacts, it will then do it one island at a time. This is mainly done to save time, and really only needlessly complicates things, so we will just pretend like all contacts are handled together. We did actually try to change the source code to make it so, but it broke the simulator for reasons we were never able to determine, although our suspicion was that some or all of the clever tricks the simulator uses to reduce memory usage were to blame.

With the presumption mentioned above in mind, what the simulator is doing to solve contacts then is to pass over all contacts, determining and applying impulses and fixing interpenetrations. After each pass over the contacts, it then considers the relevant stopping conditions, and if they are not met, it does another iteration. This continues for as long as needed.

Going into a little more detail, the first thing the simulator does is actually to apply the warm start impulses, if any are provided. This is a way to kick-start the iteration process, providing the simulator with a starting impulse value for all contacts, which one would then hope would cause it to require fewer iterations to fulfill its stopping conditions. A minimum of one iteration is still required though, due to the way the code is written. The simulator then solves the velocity and the position parts, which it actually does separately, starting with the velocity part.

For the velocity part, the simulator first calculates and applies the tangential part of the impulse needed to fix the velocity, and then afterwards it calculates the normal part of the impulse. After having done this for all contacts, it then simply does another iteration across all contacts, determining and applying new impulses as needed, then it does another iteration and so on. This process stops when it has done a fixed number of iterations, with the number being supplied by the user when asking the world to take a step. We note that as these impulses are being calculated and applied for each iteration, in order to actually know the total impulse for a given contact we would have to sum up all of the impulses calculated across all of the iterations of that step. In other words, for a given contact, if the total impulse that we have calculated and applied in $i-1$ steps is $\boldsymbol{J}_{i-1}$, then during the i'th iteration we then calculate and apply $\Delta\boldsymbol{J}_i$, bringing the total impulse

applied to

$$\boldsymbol{J}_i = \boldsymbol{J}_{i-1} + \Delta \boldsymbol{J}_i \tag{4.1}$$

In this notation, with $i = 0$ being the first iteration, $\boldsymbol{J}_{-1}$ would then be the warm start impulse provided by our model.

Next up the simulator then starts iterating over all contacts again, this time focusing on the positions. For the positions, it completely ignores the tangential part of the interpenetration and only solves for the normal part. When this has been done for all contacts, it then checks the stopping conditions, and if they are not met, the simulator does another iteration over all contacts. The stopping condition for the position, is either that the interpenetration has been reduced below a certain threshold, which is small but actually non-zero, for all contacts, or that a set number of iterations have been performed similarly to how it worked for the velocity.

Both the impulses and the corrections to the positions are referred to as $\boldsymbol{\lambda}$ or lambda in the source code, and so we have also used this name occasionally throughout our code and plots.

4.3 Modifications

We have made a number of changes to the `Pybox2D` simulator, two of which we will briefly go over.

The main change is to the contact solver, where we have changed the conditions for when the iterative solver stops, both for the velocity and the position parts. Starting with the velocity part, when doing a single iteration across the contacts we keep track of the largest impulse determined, and we then use this to add another stopping condition. We do this by comparing the largest calculated impulse for the iteration to a threshold value set by the user, and if the impulse is below the threshold we stop iterating. Now, the impulses calculated are actually vectors, and so in order to compare them to each other we need to reduce them to scalars. We have chosen to do this by taking the infinity norm of the vectors, defined as

$$|\boldsymbol{v}|_\infty = \max(v_0, v_1, .., v_n) \tag{4.2}$$

for a vector $\boldsymbol{v}$, and these are then the values that we compare.

We do a similar ting for the position part, keeping track of the largest positional change and comparing it to a threshold value. When solving the

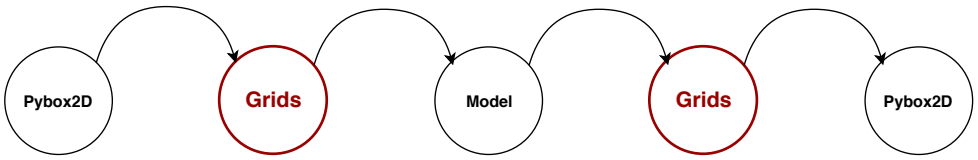
positional part of the contacts though, the velocity part has already been solved, whether in ten or a thousand iterations, and the simulator then only looks at the interpenetration when iterating. This means that the position part will produce very similar results for a given run no matter what warm start iterates we have provided, and so the position iteration process will be the same for all our models. For this reason, we will simply ignore the position part from now on, and focus only on the velocity part that we can actually influence.

The other big change is that we have added code for keeping track of, and exporting, loads of information and results pertaining to the iterations over the contacts during each step. We track the total number of contacts for each step as well as the number of iterations for the velocity part. We also keep track of the impulses being determined, in particular for each iteration across the contacts we keep track of the infinity norm of the largest impulse being calculated, the same value we use for the stopping condition. How these values change over iterations, steps and simulation runs, in particular as a result of different starting iterates provided by different models, is then what we will be interested in and what we will plot later on as various types of convergence rates.

With all of the physics and simulation out of the way, let us then take a look at how we are going to transfer all of the values associated with the bodies in our simulation to the grids that our neural network model takes as input.

5 Particles and Grids

As mentioned in the previous section, the simulator we use deals with discrete objects, and contacts between such objects, that can be positioned anywhere in a continuous 2D space. Each body as well as contact has a number of attributes, each with a specific value. Our model on the other hand will be a convolutional neural network, and requires the input data to be on a much more regular and ordered form, such as an image or a grid, where each node in the grid then has one or more values. For this reason we will need a way to go from a world of bodies and contacts to a collection of grids. Similarly, the output from our model will be a set of grids, where each node has a set of values, and we will then need a way to transfer these values back to the actual bodies and contact points.



We will start by talking a bit about grids in section 5.1, and then introduce the SPH method that we will use as the base for our particle-to-grid and grid-to-particle methods in section 5.2. In section 5.3 we will then introduce our full particle-to-grid method, and in section 5.4 our full grid-to-particle method. Finally in section 5.5 we will show and discuss the results of a few tests of these methods.

5.1 Grids

Let us start by defining what we mean by a *grid*. When we talk about a grid, we are talking about a two-dimensional collection of nodes, positioned in a rectangular shape, where all the grid nodes are placed in evenly spaced rows and columns in each of the two dimensions. The spacing between grid nodes in the x-dimension is what we call the x-resolution, and similarly the spacing in the y-dimension is the y-resolution. The x- and y-resolutions does not have to be the same, but typically will be, in which case we will simply refer to it as the grid-resolution. An illustration of a grid with different x- and y-resolutions is shown on Figure 6.

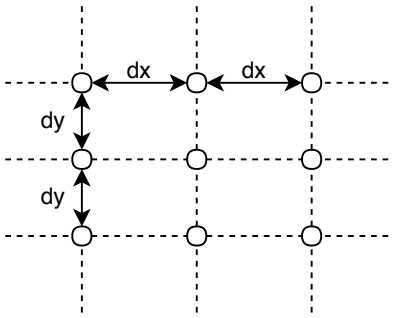


Figure 6: An illustration of a grid with nine nodes, where dx is the x-resolution and dy is the y-resolution.

An important features of the grid nodes is that they can have attributes, or put another way, they can store values. A grid node could for instance store a mass value, a velocity value and an impulse value. We will use this when we transfer values from bodies to grid nodes. In our implementation, a grid will typically be represented as a tree-dimensional matrix, or tensor, where the first two dimensions define the grid and the third dimension then holds the values.

Throughout this thesis we will sometimes be working with a single grid where each node has multiple values, and sometimes we will be working with multiple grids where each node has a single value. There is no deeper meaning behind why we use one or the other, we simply follow the way we have done it in our implementation, which uses one form or the other based on what is available and what is required. For instance, when generating our data we generate multiple grids, one for each body attribute, where each grid node has a single value, while our neural network model takes as input a single grid where each grid node can then have multiple values. From a theoretical standpoint, using one or the other representation should make no difference, although in hindsight it would have been nice with just a single representation.

Having defined what we mean by a grid, we now need a way to transfer data between bodies and grids. One obvious way to do this would be to create a grid spanning the simulation world, and then consider all grid nodes that are placed within the shapes of the bodies and assign each of them an equal amount of the attributes. It is not immediately clear though how much sense this makes for different kinds of attributes, and it also does not really tell us how to handle contact points as they are just that; points. This simple method also quickly runs into trouble in cases where the shape of the

object only encompasses a few or even no grid nodes. We will need a better method, that can handle all of these issues while not being overly complicated or time-consuming.

5.2 Smoothed Particle Hydrodynamics

The method we have chosen for this is inspired by the *Smoothed Particle Hydrodynamics* method, or **SPH**, which originally comes from the field of astrophysics [Mon92], and which has more recently seen use in fluid simulation [MCG03] among other things. Briefly put, the idea is to overlay the world with a regular grid, and then for each of our bodies, which each has a position and a set of attributes, use a so-called *smoothing kernel* to smoothly and symmetrically spread these values out over the grid nodes in the vicinity of the body. Something similar can then be done for the contact points.

A smoothing kernel, typically referred to with the letter W, is a kind of function that given a vector as well as other possible parameters depending on the specific kernel, returns a weight between 0 and 1. An important requirement for such a function to be considered a smoothing kernel is that the weights must sum to one, that is

$$\int W(\mathbf{r})\ \mathrm{d}\mathbf{r} = 1 \tag{5.1}$$

in order to ensure that value is neither gained nor lost when transferring between particles and grids.

Given a grid and a body with some attributes, we then identify all relevant nodes in the grid, where relevancy depends on the specific kernel, and then assigns each node a weight based on the distance to the body, as well as other possible metrics, using our smoothing kernel. When a node has been assigned a weight, the weight is then multiplied with each of the attributes of the body that we are interested in, and the results are stored in the grid node. All we need then, is to pick a good smoothing kernel.

5.2.1 Smoothing Kernels

The probably most familiar choice of smoothing kernel is the Gaussian kernel, which in two dimensions is defined as:

$$W_{Gauss}(\mathbf{r}, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{|\mathbf{r}|^2}{2\sigma^2}} \tag{5.2}$$

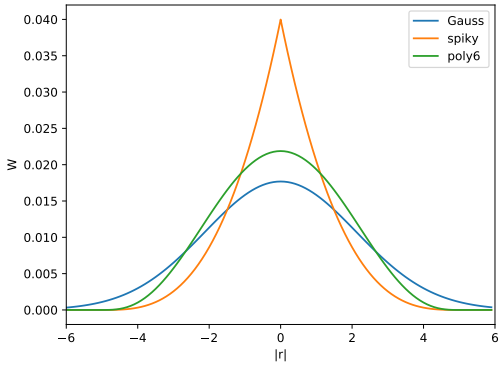


Figure 7: The weights calculated using our three different kernels, for different values of $|\mathbf{r}|$. σ is set to 3 for the Gaussian kernel, and h is set to 5 for both the poly6 and the spiky kernels.

where σ is the so-called *width* of the kernel and $\mathbf{r}$ is the vector from the body to the grid node. A profile of a Gaussian kernel is illustrated on Figure 7. This is the kernel that was originally used in SPH by physicists, and in fact, when working with physics, assuming that the kernel is some form of a Gaussian kernel is supposedly the "first golden rule of SPH" [Mon92].

That being said, the Gaussian kernel does have some issues that make it less appealing to us, mainly the fact that it has unlimited range. What we mean by this is that any node in our grid, no matter how far away from the body we are considering, will have to be taken into account and will be assigned a non-zero weight if using a Gaussian kernel. Typically there will be a lot of nodes in our grid, and a lot of these will then be assigned an effectively negligible weight, which takes precious time computing.

Another possible choice of kernel is the so-called *spiky* kernel, which is typically used in cases with lots of particles clustering together under high pressure, due to the desirable behavior of the gradient of the kernel [MCG03]. One definition of the spiky kernel is:

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - |\mathbf{r}|)^3 & 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where again $\mathbf{r}$ is the vector from the body to the grid node, and h is the so-called *support radius*. A profile of this kernel can also be seen on Figure 7. One very nice thing about the spiky kernel is that it has a very clear cutoff, in that any grid node at a distance of more than h gets a weight of exactly

0. This means that we can just ignore all grid nodes beyond distance h when calculating weights, which saves us a lot of time and effort.

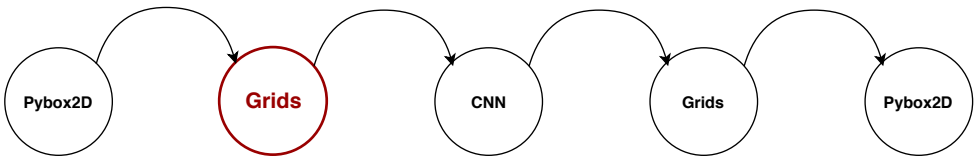
The spiky kernel is a little too spiky for our case though, and so we focus on a third kind of kernel, the so-called *poly6* kernel from [MCG03]. In two dimensions the poly6 kernel is defined as

$$W_{poly6}(\mathbf{r}, h) = \frac{4}{\pi h^8} \begin{cases} (h^2 - |\mathbf{r}|^2)^3 & 0 \leq |\mathbf{r}| \leq h \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

where again $\mathbf{r}$ is the distance vector and h is the support radius. Once again a profile of the kernel can be seen on Figure 7. Compared to the spiky kernel, the poly6 kernel has a more nice, wide profile similar to that of the Gaussian kernel, while at the same time also having the nice feature of limited range that we saw for the spiky kernel. This is the kernel that we will use throughout the project.

5.3 Particles to Grid

Having decided on a base method and an accompanying kernel, we return to the problem at hand. At any point during our simulation, we have a set of bodies, each with a set of attributes, as well as a set of contact points between bodies, which similarly each has a set of attributes. What we want, is to transfer these attributes onto a set of grids, one grid for each attribute, and we will do this using the SPH method described above.



5.3.1 SPH

The first step is, for each body, to determine which grid nodes are within range h, where h is the support radius of our poly6 smoothing kernel as described above. To do this, we will use a clever data type called a k-d tree, which is ideal for doing nearest neighbor searches. By using our grid as input we create a k-d tree, and then for each of the bodies we query the tree for all

neighbors within range h, giving us the relevant grid nodes. For each body, we then assign each of the neighboring grid nodes a weight, using our kernel W.

An important feature of the smoothing kernels is that across all space the weights sum to one, which is important for conserving the values of the attributes of the bodies. We are working with a grid with discrete nodes though, and so the weights that we calculate might not sum to one, in fact they very likely will not. We would still like to ensure that we do not lose or gain any value when transferring from bodies to grids though, and so after determining all the weights for a particular body and its neighbors, we make sure to normalize the weights such that they sum to one. In this way, all of the mass, velocity etc. of each body is transfered to the grid, without any gain or loss. When we have determined the weights for each neighbor of a body, we then multiply the weights with the values of the attributes of the body that we are interested in. In this way, each of the neighboring grid nodes should have a value for each of the chosen attributes, and we then store these values in the corresponding grids, having a separate grid for each attribute. We repeat this for all bodies, each time adding more value to particular grid nodes in the grids.

When all this is done, we repeat the process, this time considering the contact points and their attributes instead of the bodies, creating another set of grids. These grids together with the grids we created for the bodies then contain all the information our model needs to know about the world. Our algorithm for creating the grids is summed up in pseudo-code in Algorithm 1.

With this method, the values of the attributes of a body or contact can be spread out across zero or more grid nodes, and grid nodes can receive value from zero or more different bodies or contacts. We note that if a body or contact has only a single grid node within the range of the support radius, all of the attribute values of the body or contact will be transfered to this grid node, irregardless of the actual distance. This might seem a bit weird, but is a result of us normalizing the weights to ensure that value is neither gained nor lost. Another obvious issue is if a body has no neighboring grid nodes within range, in which case all of its value will be lost. The best way to prevent this, is to simply choose sensible parameters for you grid, in particular paying attention to the grid resolution versus the smoothing kernel support radius.

Algorithm 1 Transfer values from Particles to Grids

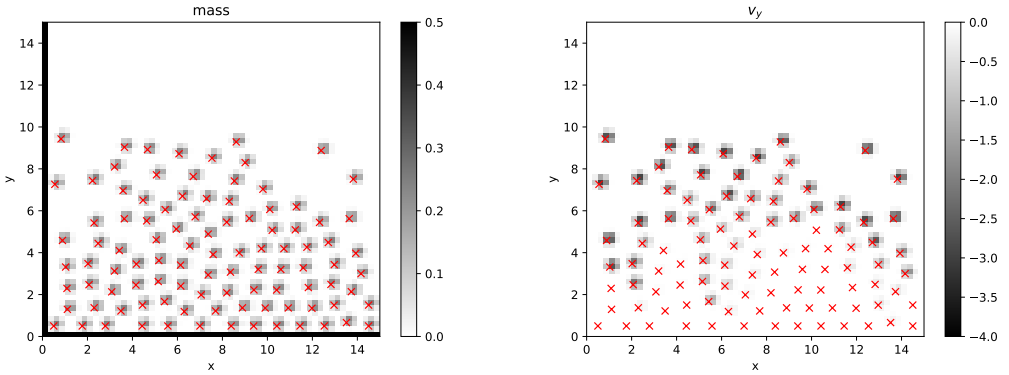
```
1:  function ParticlesToGrids(grid, bodies, contacts):
2:      tree ← KDTree(grid)
3:      neighbors ← tree.query(bodies)
4:
5:      create grids, one for each body attribute
6:      for each body do
7:          for each neighboring grid node do
8:              weight ← W(r_n − r_b, h)
9:          end for
10:         normalize weights
11:
12:         values ← body.attributes · weights
13:         store values in grids, adding new values to old
14:     end for
15:
16:     repeat for contacts
17:
18:     return grids
19: end function
```

5.3.2 Examples

A couple of examples of the kinds of grids that we have created using this method are shown on Figure 8 and 9. These are based on a simulation of a world with a 15 by 15 static box, containing 100 bodies all of which are circular with radius 0.5, and taken from a step about one second into the simulation. The plots show three of the corresponding grids, each of size 15 by 15 with a grid resolution of 0.25 and a support radius of 0.5. Further details on the kinds of simulations and grids we will use, and specifics on how they are generated are given in section 8. Red x's are used to mark the actual position of the bodies, and the bold, black lines on Figure 8a indicates the static world box, with the right wall just outside the plot..

Figure 8a shows the distribution of mass, which is nicely centered around the red x's as expected. We note that the bodies are fairly evenly spread across the world, with fewer at the top and more at the bottom as a result of the simulation having run for roughly a second. We also note that the mass distributions seem to be roughly the same for each body, as we would expect since all bodies have the same shape and mass.

Figure 8b shows the velocity in the y-direction for each of the bodies. We



(a) Body mass (b) Body y-velocity

Figure 8: Color plot of an example of a body mass grid and a body y-velocity grid. Red x's mark the position of the bodies.

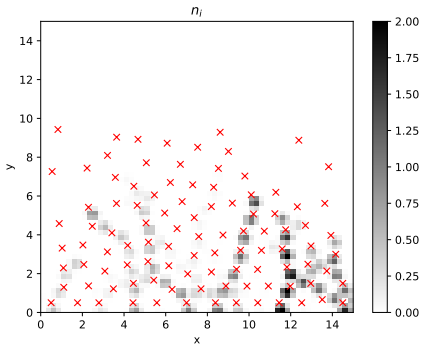


Figure 9: Color plot of an example of a grid of contact point normal impulses. Red x's mark the position of the bodies.

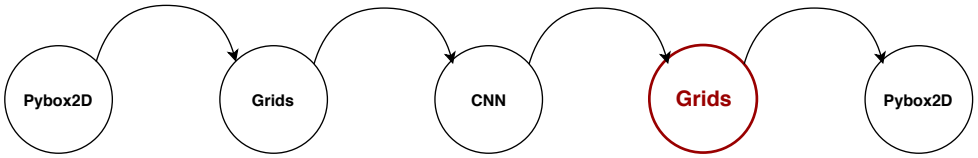
note that most of the bodies at the top have similar, negative y-velocities, which makes sense as they are probably still falling, while most of the bodies at the bottom have zero y-velocity, which again makes sense as they have probably hit the static world box and as such are no longer falling.

Similarly, Figure 9 shows the impulse in the direction of the contact normal for each of our contact points. We note that these are mainly found in the bottom of the graph, which makes sense as this is where most of the collisions are happening at this stage in the simulation. These values are calculated by the simulator based on all the other body and contact attributes, and is what we hope to teach our model to calculate for us. As such, this is an example of what we would use as a *label* when training our model, where

as the other graphs would be used as input or *features*.

5.4 Grid to Particles

Having created the input grids, fed them to the model, and allowed it to crunch the numbers, our model then returns a pair of grids as the result. What we want then, is to transfer the values from these grids back to our particles, in this case specifically the contact points, and then feed the results back into the simulation. For this we need a way to go from grids back to particles.



5.4.1 Bilinear Interpolation

A very simple way to do this would be to use a method called *bilinear interpolation*. The idea is that to determine what value to assign to a particle at a given position, we consider the four nearest grid nodes, which form a rectangle, and use the values of these to determine the value of the particle. This is done by doing linear interpolation first in one of the two directions, and then in the other direction afterwards.

Figure 10 shows an example, where we have a particle at (x, y) and four grid nodes at (x_0, y_0), (x_1, y_0), (x_0, y_1) and (x_1, y_1). Since the grid is regular we know that $x_1 - x_0 = y_1 - y_0$. The first step is to do linear interpolation in one dimension, for instance in the x-direction. We do this between p_0 and p_1, and between p_2 and p_3, by considering the ratios between the various distances. What we get is

$$v(x, y_0) = \frac{x - x_0}{x_1 - x_0} v(x_1, y_0) + \frac{x_1 - x}{x_1 - x_0} v(x_0, y_0) \tag{5.5}$$

$$v(x, y_1) = \frac{x - x_0}{x_1 - x_0} v(x_1, y_1) + \frac{x_1 - x}{x_1 - x_0} v(x_0, y_1) \tag{5.6}$$

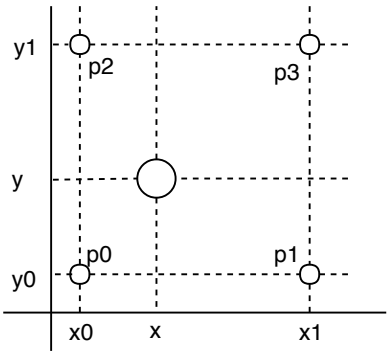


Figure 10: An illustration of a particle and its four nearest grid nodes, with various numbers highlighted.

where v is the value at a certain set of coordinates. Next up we then need to do linear interpolation in the y-direction, while keeping x fixed. The result is

$$v(x,y) = \frac{y - y_0}{y_1 - y_0} v(x, y_1) + \frac{y_1 - y}{y_1 - y_0} v(x, y_0) \tag{5.7}$$

The final thing we have to do then is to insert 5.5 and 5.6 into 5.7, use the values of the grid nodes that we know beforehand, and we can then determine the values for our particle. We do this for all the particles, which again are the contact points, and then we can feed the values back into the simulator.

This is a fast and relatively simple method, but it does have a weakness in that it only considers the four closest grid nodes, which might not always be ideal. When going from particles to grids, with each grid having some fixed resolution, we might have used a kernel with a support radius h such that the values of the various body and contact attributes are spread out over many grid nodes, i.e. more than four. In such a case, bilinear interpolation would not be able to recapture all of the value transferred to the grid, and so we need a better method.

5.4.2 SPH

An obvious thing to do would be to once again use our modified SPH method, only this time we would need to modify it to work in the reverse direction. This is actually pretty straightforward to do, requiring only a few simple changes to Algorithm 1 that we used before.

Algorithm 2 Transfer values from Grid to Particles

```
1:  function GridToParticles(grid, particles):
2:      tree ← KDTree(particles)
3:
4:      create list of grid nodes
5:      neighbors ← tree.query(gridnodes)
6:
7:      for each grid node do
8:          for each neighboring particle do
9:              weight ← W(r_n − r_p, h)
10:         end for
11:         normalize weights
12:
13:         for each neighboring particle do
14:             values ← gridnode.attributes · weight
15:             store values in particle attributes, adding new values to old
16:         end for
17:     end for
18:
19:     return particles
20: end function
```

Briefly described, our method was looking at each body, determining the relevant neighboring grid nodes, determining the weights for the grid nodes and finally transferring the values of the body attributes to these grid nodes. This time we will instead look at each grid node and determine relevant neighboring particles, which can be either bodies or contacts. We will then determine a weight for each such particle once again using our `poly6` kernel, and then normalize the weights to ensure they sum to one. The values stored in the grid nodes will then be distributed among the particles according to these weights. A pseudo-code version of this method is presented in Algorithm 2.

Using this method, a grid node can spread its values out to zero or more particles, and a particle can receive value from zero or more grid nodes. We again note that if a grid node has only a single particle within the range of the support radius, all of the value stored in the grid node will be transfered to this particle irregardless of the actual distance between the two. This is once again a result of us normalizing the weights to ensure value is not gained or lost.

The fact that we do not gain or loose value also does not mean that every

particle necessarily gets their correct share. Image that we first transfer the values of two particles to a grid node, one particle being close with a low value and one being far away with a large value. When we then transfer the values from the grid node and back to the particles, the closest one will then be the one with a large value, and the one further away will be given a small value, which is the opposite of the original situation.

Another important thing to note is also that this only works because we have access to all of the particles at once. If we were instead given a stream of particles, one by one, and asked to assign them values based on a grid, this method would not work, and we would have to choose another method, like bilinear interpolation. Despite all of this, we will indeed use the SPH methods outlined in Algorithms 1 and 2 for our project.

5.5 Tests

Finally, we have done some simple tests of the importance of the grid related parameters, namely the grid resolution and the SPH support radius. For this we created twentyfive worlds similar to those that we will use for testing the performance of our model later on. The details of these worlds and their creation are described in detail in section 8.1, but in short they are 15 by 15 worlds, with 100 bodies that are all circles with radius 0.5. Each world was simulated for two hundred steps of 0.01 seconds each, such that all the bodies were in a big pile. For each world we consider either each body or each contact in the world, and note down the original value of one of their parameters. We then transfer these values for all of the particles, again either bodies or contacts, onto a grid and immediately back to the particles, and we then note down the new values and compare them to the originals and get a difference. We do this for bodies and their mass, and we do it for contacts and their normal impulses.

For the test we first used five different resolutions and five different support radiuses, namely 0.125, 0.25, 0.50, 0.75 and 1.00 for both, which gives us twentyfive different pairs of parameters. For each pair we then performed the test described above using the twentyfive worlds, and determined the average differences. We quickly learned that the high values of these parameters led to bad results, and so we cut out the 0.75 and 1.00 values, and the remaining results can then be seen in Table 1.

First up, let us look at the result for the bodies in Table 1a. All bodies are the same shape and size and as such also the same mass. The first thing we note is that choosing a support radius that is smaller than the grid resolution seems to lead to bad results. For comparison, the mass of each

res	h	Avg body diff
0.125	0.125	0.0000
0.125	0.250	0.0000
0.125	0.500	0.0000
0.250	0.125	0.1137
0.250	0.250	0.0000
0.250	0.500	0.0000
0.500	0.125	0.5177
0.500	0.250	0.1074
0.500	0.500	0.0000

(a)

res	h	Avg contact diff
0.125	0.125	0.0000
0.125	0.250	0.0000
0.125	0.500	0.0191
0.250	0.125	0.0480
0.250	0.250	0.0000
0.250	0.500	0.0195
0.500	0.125	0.2349
0.500	0.250	0.0589
0.500	0.500	0.0237

(b)

Table 1: Average difference between original value and value after particle-to-grid-to-particle transfer, for body mass (a) and contact point normal impulses (b), for different values of grid resolution and support radius.

body is roughly 0.7854, so an average difference of 0.5177 which is the case for one of the pairs is pretty bad. That fact that choosing a support radius that is smaller than the grid resolution is bad makes sense as then there might be cases where there are no grid nodes within the support radius for some bodies and all their value would then be lost.

Now, there are several pairs of parameters that leads to an average difference less than 10^{-4}, so we probably want to pick one of those. Seeing as our bodies are circles with radius 0.5, and that contacts between bodies will happen at the edges of bodies, it it probably not a good idea to pick a support radius of less than 0.5, as that would mean that in most cases all of the body values would be zero at the grid nodes where the contact point values are found. This then leaves us with a grid resolution of either 0.125 or 0.25.

It is worth noting that as we are dealing with a two-dimensional grid, every time we cut the grid resolution in half, we quadruple the number of grid points, which increases the run time and memory usage of every piece of code using the grids including our neural network model.

Next up then, we look at the results for the contacts in Table 1b. In this case, all contacts will have different original values, which means that the average difference is not quite as simple to interpret as it might not give as full a picture as for the bodies. We note though that once again choosing a support radius that is smaller than the grid resolution is a bad idea, which is for the same reasons as for the bodies. Other than that, it seems that for

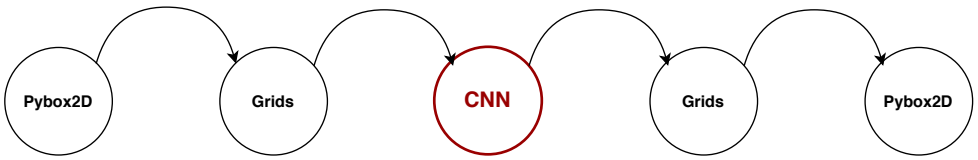
the contacts the result are in favor of choosing as small a support radius as possible. Keeping in mind that we really want a support radius of at least 0.50 for the reasons mentioned above, we are left with a grid resolution of 0.125 or 0.25, which have roughly the same loss. These are the same values as we settled on for the bodies, and keeping in mind that smaller resolutions are worse computationally, our final choice is a grid resolution of 0.25 and a support radius of 0.50.

Now, having sorted out the physics and the simulator, and designed a method for transferring data between the simulator and the grids that we will need, it is then finally time to move on the next part, starting with neural networks.

6 Neural Networks

Before actually getting to the model that we have designed and implemented, we first need to talk a little about the type of model that we will be building.

After extracting all of the physics data we need about the current state of the world from the simulator, and splattering it onto grids using the methods in the previous section, we will then use these grids as the input to our model. The model should then crunch the numbers, and output another set of grids representing the impulses, and the ideas behind how the model is built and how it crunches the numbers is what we are going to take a look at in this section. Our model will be built as a so-called *neural network*, which is a type of model that has been very popular lately, and which is good at modeling a large range of different things and phenomenons. Our hope is of course that it will also be good at modeling physics.



This section is intended as a quick brush up for fellow computer science students who might have already heard a bit about neural networks, or as a high level introduction for the interested numerical analyst. The focus will be on introducing and explaining the different kinds of pieces used, such as neurons, layers, convolutions etc., and some of the different ways to improve neural networks, such as dropout, regularization and normalization, but without going too deep into all the complicated math behind. For a more rigorous and in-depth explanation of the different concepts, we once again recommend looking at literature written for that purpose, such as [Nie15] which we used as a reference.

The general ideas and principles behind neural networks will be presented in section 6.1, where we talk about neurons and activations functions, and in section 6.2, where we talk about layers of neurons including convolutions. In section 6.3 we will then talk a little about how to train a neural network, and finally in 6.4 we will talk about a few tips and tricks that we will use to improve our neural network models.

6.1 Neurons

The basic idea behind neural networks is loosely based on the neural network of the human brain. The basic building blocks are objects known as *neurons*, which takes one or more numbers as input, and in return outputs another number. Each of the input numbers are assigned a weight, multiplied with this weight, and then all added together. Typically a bias is also added, and then the result of this is the number that the neuron outputs. A simple example of a neuron is shown on Figure 11. If we imagine a neuron getting the numbers $x_1, x_2, ..., x_m$ as input, the output y is then

$$y = b + \sum_{i=1}^{n} x_i \cdot w_i \tag{6.1}$$

with $w_1, w_2, ..., w_n$ being the assigned weights and b being the bias. This is also often written using vectors as

$$y = b + \boldsymbol{x} \cdot \boldsymbol{w} = b + \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \\ b \end{bmatrix} \tag{6.2}$$

where in the last step we have simply included the bias in the vectors.

6.1.1 Activation functions

Sometimes the result of the sum of weights in equation 6.1 is what we actually want as output, but often a neuron will do one more thing before providing an output, which is pass this sum through a so-called *activation* function, a. There are several reasons why one might use an activation function. One reason is that rather than the neuron outputting some completely unbounded number, we might want to limit it to output a number in a certain range. Another reason for using an activation function might be that we want to modify how changing the weights and bias of a neuron impacts the output.

A very simple example of an activation function is the *step* function, which simply is defined as

$$a_{step}(z) = \begin{cases} 0 & \text{if } z < t \\ 1 & \text{otherwise} \end{cases} \tag{6.3}$$

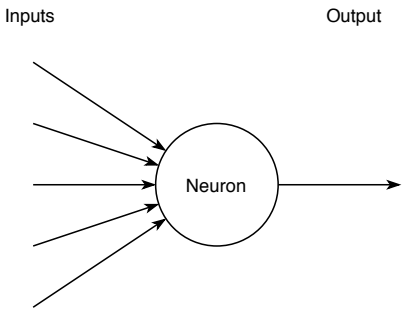


Figure 11: An example of a neuron with five inputs.

where t is some threshold we choose. The input, z, would then be the result from equation 6.1. This function can for instance be used when dealing with some kind of binary problem, where the output is expected to be yes or no. The function is typically a little too simple for most use cases though, and also has the often undesirable property that even a small change in the weights might cause a major change in the output, i.e. causing it to flip from zero to one or vice versa.

Another very common choice of activation function is the *sigmoid* function, σ, also sometimes known as the *logistic* function, which is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.4}$$

On Figure 12 a few activation functions are shown, and from this we see that the output of the sigmoid function is a number between zero and one. This is similar to the output from the step function, except this time the output can be any number between zero and one rather than just zero or one. In this way, rather than simply predicting yes or no, the neuron can now make a prediction of what the chance is of yes and what it is of no. We also note that if z is roughly five or larger, the output will approximately be one, and similarly if z is minus five or less the output will be approximately zero.

A third and final type of activation function is the so-called *rectifier*, also commonly referred to as *rectified linear unit* or `ReLU`, despite this technically being the name for a unit, such as a neuron, using such an activation function. The function is defined as

$$a_{ReLU}(z) = max(0, z) \tag{6.5}$$

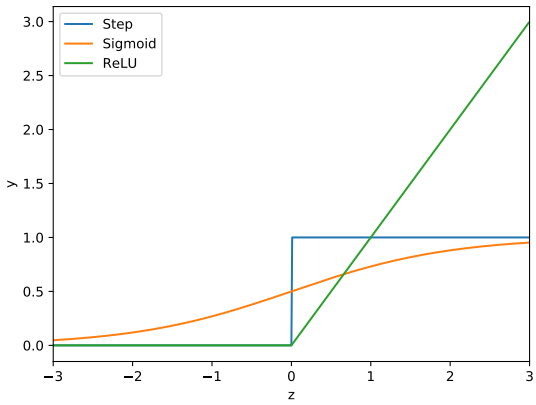


Figure 12: An example of three different activation functions. The threshold for the step function is zero.

which has the very simple effect of returning zero if the input is negative, and simply returning the input otherwise. This might seem like a very different function compared to the two previous ones, but it turns out that it actually has most of the same properties that made the other two useful for neural networks. Also the ReLU has shown very strong results when used in practice in neural networks, which is the reason that it is probably the most commonly used type of activation function, and also the one that we will use.

6.2 Layers

Having decided upon a set of neurons and activation functions, multiple neurons are then typically grouped together into what are called *layers*. A very common and simple example is the so-called *dense* layer, in which multiple neurons are placed in a column, with all of the neurons receiving the same inputs, and each then providing an output. A simple example of a dense layer is shown on Figure 13.

We can then consider a layer of neurons to be an entity in itself, which takes a number of inputs, processes them, and provides a number of outputs, and a neural network can then be built by connecting a series of such layers. A simple example would be to take a number of dense layers, and then connect them in a serial fashion, with each layers taking as input the output from the previous layer, and providing its output as input to the next layer. A simple example of such a network is shown on Figure 14.

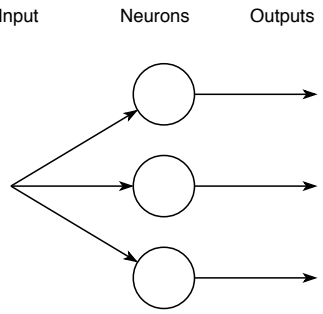


Figure 13: An example of a dense layer with one input, three neurons and as a result, three outputs.

Often, the inputs to a neural network is represented as being a dense layer in itself, with each input then being represented as a neuron which outputs the respective input number. Another way to think of it is that each of the input neurons takes one the the input numbers as input, and then uses a weight of one, a bias of zero and no activation function. This layer is typically called the *input* layer. Similarly, the last layer is often referred to as the *output* layer, as each neuron takes a number of inputs and then outputs a number which is not passed to any other neurons. Instead these numbers are grouped together and represent the output of the neural network. The neurons in the output layer also typically does not use any activation function, but instead outputs unbounded numbers, which one can then process in whatever way wanted. Finally the layers between the input and output layers are often referred to as *hidden* layers, as we do not typically see the output of these layers directly.

There are of course many different ways of combining neurons into layers besides dense layers, and there are many different things that one can do with such layers besides just passing the input through neurons. Two particularly common and clever layers that we will use are *convolution* and *pooling* layers.

6.2.1 Convolutions

In traditional dense layers, all neurons have access to all of the output from the previous layer, and provide their output to all of the neurons in the next layer. This means that every neuron has access to all of the input information, which seems like a good thing, but which also means that each neuron will have a lot of weights associated with it, which might not be ideal. The idea behind convolution layers is that rather than giving each neuron access to all of the inputs, we only give them access to part of the information, with each

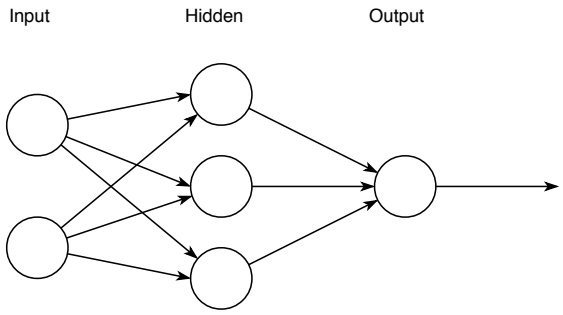


Figure 14: An example of a neural network with two inputs, a hidden layer with three neurons and a single output.

neuron then having access to different, but potentially overlapping, parts of the input. A good way to think of it that instead of each neuron knowing about everything that happens in the world, it will only know about what is happening in its own, local part of it.

A convolution layer expects the input to be in the shape of a grid, as opposed to a column like a dense layer. Similarly, the neurons in a convolutional layer, and hence the output, is also in the shape of a grid. The output will then be created by considering different patches, or groups, of neurons in this input grid, with the patch typically being called the *kernel*. An example of this is shown on Figure 15, where a kernel of size of 3 by 3 is used, which means that the output will be created by collecting the neurons into, potentially overlapping, groups of nine. When we are considering a particular neuron and its group of inputs, the output of the neuron is then created by multiplying the value from each of the neurons in the group by a weight, and then adding a bias, similarly to what we did before.

Another property of a convolutional layer is the *stride*. This is the number of neurons in each direction that the kernel is moved when doing the convolution. The convolution on Figure 15 uses a stride of 1 by 1, which in turn means that each input neuron in the convolution layer will be a part of up to nine groups. If we had for instance instead used a stride of 3 by 3, each neuron would then only be a part of a single group. Typically a stride of 1 by 1 is chosen, so as to create as many groups as possible, i.e. as many different combinations of the input neurons as possible.

A final important property of a convolutional layer is the *padding*. In our example shown in Figure 15, no padding is used, which means that we go from having a total of sixteen input neurons in the convolution layer to only having four output for the next layer. Often, one might want to keep the number of inputs and outputs the same though, which is where padding

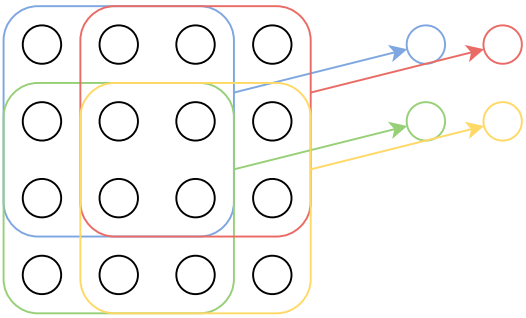


Figure 15: An example of a convolution layer with a kernel size of 3 by 3 and a stride of 1 by 1, where the nine inputs in the blue square are passed to the blue neuron, the nine inputs in the green square are passed to the green neuron, and so on.

comes in. The padding is done by adding extra input neurons to all four edges of the grid, in a number sufficient to ensure that the number of outputs is as desired, taking into consideration the kernel size and the stride of the convolution. Each of these added input neurons then simply has a value of zero, in order to ensure that the numbers remain consistent.

The input grid is then split into as many groups as possible, as determined by the kernel size, stride and padding. A convolution layer is then typically set up such that the upper, left neuron gets access to the upper left group of inputs, the neuron to the right of this neuron gets access to the group of inputs to the right of the others group of inputs, etc.

It is also worth noting that the input grid is allowed to have multiple values per grid node, which would be equivalent to having multiple input grids of the same size, and the output grid can also have multiple values per node. In fact, this is very common, and in our model we will almost always be working with grids with multiple values per grid node. How many values each grid node in the input grid has is typically referred to as the number of *channels*, and the number of values per grid node in the output is typically something that one can set by specifying the number of *filters*. In the case of multiple values per grid node in the output, the values are determined in the same way as before, only using a different set of weights and biases for each value for each node.

6.2.2 Pooling

The other new type of layer is the *pooling* layer. Similarly to the convolution layer, pooling layers expects the input to be in the shape of a grid, and

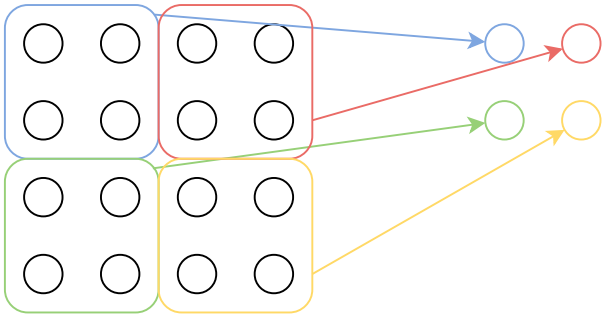


Figure 16: An example of a pooling layer with a kernel of 2 by 2 and a stride of 2 by 2, where the values from the four neurons in the blue square are passed to the blue neuron, the values from the four neurons in the green square are passed to the green neuron, and so on. If it was a max-pooling layer, the inputs to each neuron would then simply be put though the `max` function, and the result would be the output.

then handles the neurons in smaller groups at a time. In a pooling layer, the output of each neuron is determined by putting the input values to the neuron through some kind of function which accepts multiple inputs and produces a single output. Two very common functions to use are the `min` and `max` functions, which can take any number of inputs and then simply outputs the respectively smallest or largest value. Another common choice is to use the `average` function, which can also take any number of inputs. Typically the name of the function being used is included in the name of the layer, so for instance a max-pooling layer is one that uses the `max` function. An example of a max-pooling layer is shown on Figure 16.

Another similarity between convolution and pooling layers is the parameters involved. Pooling layers typically also have parameters for kernel size, stride and padding, with these having the same effect as before. For pooling layers though, these are often set in such a way that the number of outputs is actually lower than the number of inputs. As a simple example, on Figure 16 we have a max-pooling layer with a kernel size of 2 by 2 and a stride of 2 by 2. This results in the dimensions of the grid being cut in half, which in turn means that the number of outputs is smaller than the number of inputs by a factor of four.

The reason for choosing parameters that reduce the size of the grid is first of all that it reduces the number of outputs, which makes the following layers smaller. Another common reason is that reducing the size, or resolution, of the grid, allows one to sort of 'zoom out' a bit, and consider more of the data at once. In particular, it might be possible to detect various large-scale

features that might be present in the data, and which are difficult to see when focusing on just a few neurons and their local area.

Neural networks that use convolutions, and pooling, are typically referred to as *convolutional* neural networks, and are typically used when working with data that is already grid based, such as images.

6.3 Training

Now that we have the tools for building a neural network, we need a way to actually train it to provide the correct output given a set of inputs. The idea behind training is that once we have built a model in the form of a network with layers of neurons, we want to train the model by providing it with examples of input and output pairs, which tells the model what the output should be given a certain input. The model can then make the required changes to the weights and biases of all the neurons in order to achieve this. This is repeated for a number of input and output pairs, and in the end we hope that the model is then capable of making good predictions of the expected output given a certain input.

Let us consider a single neuron. It takes a set of inputs, the vector $\boldsymbol{x}$, multiplies it with a set of weights, the vector $\boldsymbol{w}$, and passes the result through an activation function, a. For simplicity we have simply included the bias in the vectors as in equation 6.2. The result of $a(\boldsymbol{x})$ is then the neurons' predicted output. Let us now imagine that we know what the actual output of the neuron should be, given $\boldsymbol{x}$, and call it $y(\boldsymbol{x})$. The difference between these two numbers, $|y(\boldsymbol{x}) - a(\boldsymbol{x})|$ then tells us something about how well, or badly, the neuron is performing.

Let us then consider a layer of neurons. Once again we have a set of inputs, $\boldsymbol{x}$, and all of the neurons will then compute $a(\boldsymbol{x}_i)$ as before, with $\boldsymbol{x}_i$ being the part of $\boldsymbol{x}$ that the i'th neuron has access to, which we can collect in a vector $\boldsymbol{a}(\boldsymbol{x})$. If we then once again assume that we know what the output of the layer should be, $\boldsymbol{y}(\boldsymbol{x})$, we can compute the difference between the two, $||\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{a}(\boldsymbol{x})||$.

Similarly, if we consider an entire network with multiple layers, we can still focus on the output provided by the network given a set of inputs compared to the correct output. Of course there are many different ways of computing the difference between the correct output and the predicted output, using more or less complicated functions. These are typically called *cost* functions,

or *loss* functions, and one simple example would be to take the squared error

$$C(\boldsymbol{x}, \boldsymbol{w}) = ||\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{a}(\boldsymbol{x})||^2 \tag{6.6}$$

where again $\boldsymbol{w}$ represents all of the weights, and biases, in the network, C is the loss function, and $C(\boldsymbol{x}, \boldsymbol{w})$ is our loss given $\boldsymbol{x}$. Of course, simply training with a single pair of input-output data is probably a bad idea, and so we will want to train on multiple pairs, as many as we can get and have time for really. For each input-output pair we will get a loss, and so we need a new loss function. One that is commonly used for cases like the one we will be working with is the *mean squared error* function, or `MSE`, defined as

$$C(\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{n} \sum_{i=0}^{n} ||\boldsymbol{y}(\boldsymbol{x}_i) - \boldsymbol{a}(\boldsymbol{x}_i)||^2 \tag{6.7}$$

where $\boldsymbol{x}$ is now a vector of input sets, and n is the number of such sets. What we want to know then, is how to modify the weights in order to make this loss as small as possible. If we have a fixed set of input-output pairs, then we can ignore the dependence on $\boldsymbol{x}$, and C is then simply a function of $\boldsymbol{w}$. So what we have is $C(\boldsymbol{w})$, a function of many parameters in the form of weights, and we want to minimize this function.

The solution is of course to use the tried-and-true method of *gradient descent*. Without going into too much detail with how gradient descent works, the idea is to determine the gradient of our cost function, $\nabla C(\boldsymbol{w})$, which is a vector consisting of all the partial derivatives of C with respect to our weights. This gradient is determined using a method known as *backpropagation*, which is a very clever way of determining the gradient by starting at the final layer and then working backwards, hence the name. Backpropagation is very much the core of training neural networks, but it is also a fairly complicated and mathematically heavy method, and so we will not go into further details about it.

Once we have calculated the gradient, we will then take a step in the direction of $-\nabla C$, hoping that this reduces our loss, and then we will repeat the whole process until we decide that it is time to stop, for one reason or another. What we mean by this is that given $\boldsymbol{w}_i$ and $C(\boldsymbol{w}_i)$ at some step i, we will determine ∇C_i and then use this to update our weights according to

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \nabla C_i \tag{6.8}$$

where η determines the size of the step we take, and is typically called the *learning rate*. We will then determine ∇C_{i+1}, take another step, and so on. This continues until we decide to stop, which might be due to various reasons such as having reached a minimum, seeing so small changes for each iteration that continuing does not seem worthwhile, having hit the maximum number of iterations intended, etc.

Knowing when to stop iterating is actually difficult, but is also quite important, due to something called *overfitting*, which can be a big issue when training neural networks. The problem is, that if you train for a very long time on the same training set, particularly if the training set is not too large, you might end up modifying the weights such that the model can perfectly predict the output for each input set in the training data, only then to discover that given a set of input data that the model has not seen before, the models performs poorly. To avoid this, a so-called *validation* set of data is often used, where occasionally while training the model will be given the input data from the validation set and asked to predict the output, and the prediction will then be compared to the correct validation output. The performance on the validation set is then tracked, and if it starts to decrease while the loss for the training set is still seemingly improving, this might be a sign of overfitting and tell you that it is time to stop iterating.

Another choice we have to make is what weights to use initially when we start iterating. A simple choice would be to simply set all weights equal to zero initially, but this turns out to be bad for a number of reasons. The standard choice is instead to initialize the weights using a normal distribution, with mean zero and a standard deviation of ones choosing. For instance, a common choice is one divided by the number of weights in the network. We also need to initialize the biases, which are normally either simply set to zero or similarly picked from a normal distribution.

Now, if we want to do gradient descent and backpropagation while taking our entire training set, which might include anything from thousands to millions of test sets, into consideration for each iteration, it could potentially take a monumental amount of time. For this reason, a variation of gradient descent called *stochastic* gradient descent is often used when training neural networks. The idea is that rather than considering the entire training set for every iteration of training, we will instead consider smaller subsets of the training set, typically called *batches*. For each iteration we then choose a batch b, and uses this to calculate ∇C_b, which can be done significantly faster, hoping that it is not too different from the actual gradient.

In total then, by iterating multiple times over batches of our training set,

we are able to modify the weights such that the loss of our model is reduced, which in turn should improve the ability of our model to predict the correct output given a set of inputs. In our model we will actually be using a more complicated training method known as `ADAM` [KB14], which generally seems to perform better than stochastic gradient descent, but is also a lot more complicated, despite the basic idea being the same.

6.4 Optimizations

Finally there are various tips and tricks people have come up with over the years, which can optimize your neural network in different ways such as reducing training time or overfitting. We are going to go through three of these that we will use for our own neural network later on.

The first technique is called *dropout*, and is typically represented as another type of layer. The dropout layer will have the same shape as the previous layer, and each neuron in the dropout layer will then receive just a single input, namely the output from the corresponding neuron in the previous layer. Each neuron in the dropout layer then either ignores its input and outputs zero, or it outputs a scaled version of the input. The fraction, or rate, of neurons that output zero is a parameter of the dropout layer that you choose, and the output of the remaining neurons is then scaled based on this rate such as to keep the total sum of the output of the dropout layer the same as the total sum of the output from the previous layer. When training your neural network, the rate for each dropout layer will be fixed, but what neurons outputs zero and what neurons doesn't will change on every training run. In this way, the model can never be too dependent on specific neurons, or groups of neurons, as they might sometimes simply output zero. This helps with avoiding overfitting, and so is a very common thing to use, especially in combination with dense layers.

The second technique is called *batch normalization* [IS15], and is something that takes place between the layers of a network. The idea is to take the output from one layer and normalize it before passing it on to the next. When training the network with a batch of training data, the idea is to consider the output of a specific neuron in the layer being normalized, and collect the output of this neuron for each of the training sets in the batch. The mean and standard deviation is then determined for this set of outputs, and all of the outputs of the specific neuron are then normalized by subtracting the mean and dividing by the standard deviation. The same is then done for each of the neurons in the layer. During training, a kind of moving mean

and standard deviation is also determined and continually updated, and this is then used for normalization when the network is used to predict the output of inputs sets not seen before. This might seem like a very rigid thing, and so in order to give the optimizer training the network more control over the normalization, including the option to basically disable it, two new parameters are introduced, typically called β and γ, which control the process, and which the optimizer can then add to the set of parameters to be trained and learned.

The third technique is called *regularization*, and has to do with controlling the size of the weights in the network. If left uncontrolled, the weights in the network might grow to become very large, in fact as large as the models wants. For instance, you could imagine all the weights in one layer growing very large, and then the weights in the next growing proportionally small to compensate. These uncontrolled weights leads to weird behavior and makes the model more unstable, and so we might want to prevent this. One way is to use regularization, which is a way of punishing the model for using large weights. The way it works is that we add an additional term to our loss function, depending in some way on the size of the weights, and during training the model will then tend towards choosing smaller weights in order to minimize the loss. Two of the most common types of regularization are *L1* and *L2* regularization. L1 regularization adds the term

$$\sum_i |\boldsymbol{w}_i| \tag{6.9}$$

to the loss function, which encourages the model to reduce the size of all weights as much as possible. This in turn can lead to some weights being reduced all the way to zero, if the numbers they are multiplied with are not important for the performance of the model. L2 regularization adds the term

$$\sum_i \boldsymbol{w}_i^2 \tag{6.10}$$

to the loss function, which encourages the model to reduce the size of the largest weights more so than the size of the smaller weights. Using this form of regularization it is less likely that any weights will be set to zero, and so all inputs and numbers should still have an impact.

One thing that is commonly done for both of these types of regularization, and also others, is that the term is multiplied with a chosen scalar when added to the loss function, with the scalar typically called λ and not to be confused

with the lambda we used for impulses. This λ can then be used to control how much effort the model should put into reducing the regularization term, with a value of zero telling the model to ignore it, and a large value telling the model that it is very important, perhaps even more important than making good predictions. In fact, one has to be careful when choosing λ, or risk ending up with a model with very low weights but poor results.

Now, having talked about what neural networks are, how they are built, how they work and how they are trained, let us move on to the specific one that we built.

7 The Models

Throughout our work on this project we have built and tested a lot of different neural network models, starting with very simple, linear networks with just a few convolution, pooling and dense layers, and then adding more and more layers, connections, optimizations and so on. Going over all of these would take a long time, and a lot of them did not perform very well, so instead we have just picked out the most successful of the neural networks that we built.

In an article we stumbled upon late in the process, [TSSP16] which is also mentioned in section 2, the authors create a model which deals with simulating fluids using grids and neural networks among other things. Their neural network is actually fairly similar to the one that we had created ourselves, with some interesting differences that actually seems rather clever and makes a lot of sense. For this reason, we have implemented a version of this model as well, and later on we will then test not only the performance of our own neural network model but also that of a model based on their neural network, in order to have something to compare our own model with.

Our implementations of both models use some or all of the optimization techniques described in section 6.4. That means doing dropout before dense layers, doing batch normalization after every convolution and dense layer, apart from the output layer, and using L2 regularization in all layers that use weights. All of our neural networks have been built in `Python` using the `TensorFlow` machine learning framework.

7.1 The `Peak` model

The first model that we will take a look at is the best of the models that we created on our own, which we will call the `Peak` model as it is peak of our design process. Figure 17 shows an overview of the model, which at first can seem a little complicated.

The spine of the model is a linear set of alternating convolution and average pooling layers, which is similar to how many of our first models were built. There are four convolution layers, all with a kernel of 4 by 4, a stride of 1 by 1 and using padding to maintain the number of elements, with an increasing number of filters in each layer. In between these are three average pooling layers, all with a kernel of 2 by 2 and a stride of 2 by 2, each of which reduces the dimensions of the data by a factor of two, which in turn means reducing the number of nodes by a factor of four. The idea is to use convolutions to let each grid node know about the values of nearby grid nodes besides just its own, and to use pooling to reduce the resolution, allowing each convolution layer to look at larger and larger parts of the data.

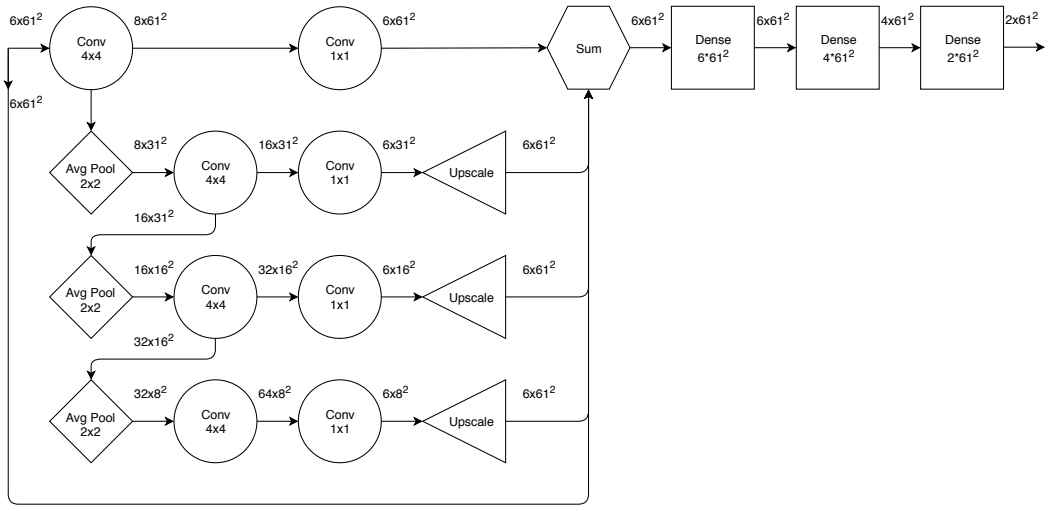


Figure 17: An overview of the `Peak` model. The numbers outside the shapes refer to the size of the data, or tensors, being passed around, and the numbers within refers to the kernel sizes, or to the number of elements for the dense layers.

Rather than then simply passing along the output of the final layer to the dense layers, as we did in most of our early models, we forward the output of every one of the four convolution layers, as well as the raw input, to the dense layers. In this way, we have data for the grid nodes and their neighborhood on a number of different scales. Originally we combined all of these sets of data by passing each of them through a small dense layer and then concatenating all of the results, but this was both overly complicated and very slow. Taking an idea from the `Pressure` model, we now instead use an additional convolution with a 1 by 1 kernel to reduce the number of filters to be equal for all five sets of data, and then use a layer of upscaling, based on bilinear interpolation, to modify all the data to have the same dimensions. Finally we then added it all together.

This sum is then used as input to a line of dense layers, with each dense layer having fewer elements than the previous, until we reach the output layer. The idea is to give each grid node access to all of the information we generated, and then give them a few layers to add and multiply things together as needed. Using dense layers means that all grid nodes have access to all the information about all the other grid nodes, which might be useful, but the downside is as mentioned that it means that we have a lot of weights, in fact our models has tens of millions of them.

The model uses dropout with a rate of 0.4, batch normalization and L2

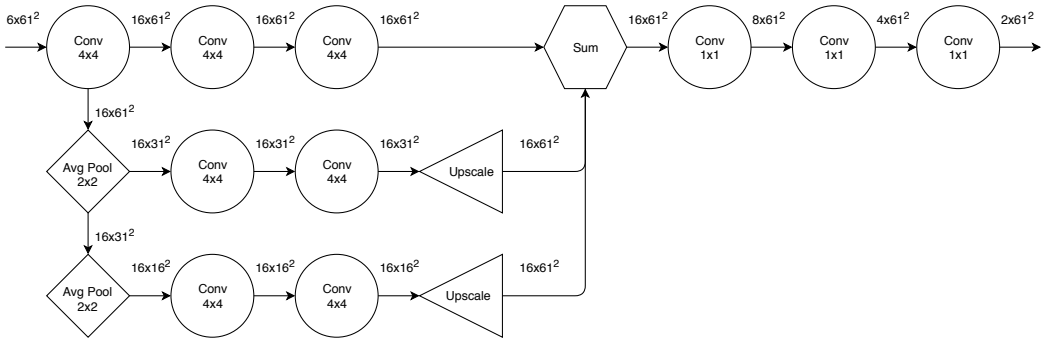


Figure 18: An overview of the `Pressure` model. The numbers outside the shapes refer to the size of the data, or tensors, being passed around, and the numbers within refers to the kernel sizes.

regularization with a λ value of 0.01. All weights were initialized using a normal distribution with a mean of zero and a standard deviation of 0.1. The model was trained with a learning rate of 0.001.

7.2 The `Pressure` model

As mentioned earlier, the second neural network model that we will consider is based on the neural network from [TSSP16], and we will call it `Pressure`. We note that though the overall setup of the neural network is unchanged from what they designed in the referenced paper, we have made changes to the number of layers, number of filters, kernel sizes etc., in order to improve the performance of the model on our data and setup. An overview of the model setup is shown on Figure 18, which shows that the model is actually fairly similar to the `Peak` model from before, with a few main differences.

The first part is once again made up of convolutions and poolings, only this time the raw input is put through a single covolution layer before being sent through a line of average pooling layers. The output from the initial convolution layer, as well as the output from each of the pooling layers, is then sent through lines of convolution layers, before being upscaled, if needed, and then all added together. This is very similar to how we built or own model, with just a few differences in the order of layers, and the idea seems to similarly be to collect information about the grid nodes on a number of different scales. All convolution layers use a kernel size of 4 by 4, a stride of 1 by 1 and uses padding to maintain the number of elements. All pooling layers use a kernel of 2 by 2 and a stride of 2 by 2.

The second part of the model is also fairly similar to ours, although it uses

a different approach in that it does not use dense layers but instead simply uses more convolution layers. Each convolution layer uses a kernel size of 1 by 1, which means that each grid node does not have access to all of the data, as they did in our `Peak` model where we used dense layers, but instead only has access to the information gathered about that specific node, which does include some information about the neighborhood from the convolution and pooling layers in the first part.

Having less information might seem like a bad thing, but the upside is that it reduces the number weights for each layer from millions to thousands, which significantly reduces the complexity. One might also ask why the grid node in the bottom right corner should really care about knowing all of the details for the grid node in the upper left corner, and removing the access to this information might even help the model focus on the data that matters more. Another benefit is in training time, which is something that we have not really focused on but is worth mentioning, as this model is around fifty to a hundred times faster to train compared to our `Peak` model with its dense layers. For this part, besides using a kernel of 1 by 1, the convolution layers all use a stride of 1 by 1 and padding.

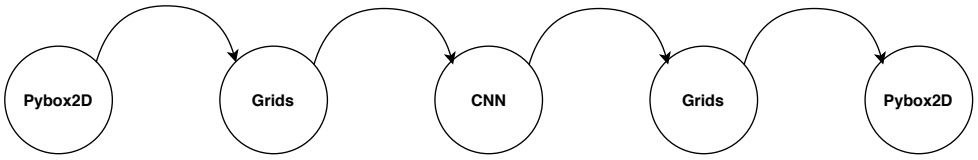
Our implementation of the model uses batch normalization as well as L2 regularization with a λ value of 0.01, but it does not use dropout. This is another one of the differences that actually seems make a lot of sense, since the point of dropout is to limit the information that the model has access to when training so as to not make the model overly dependent on certain features. Most neural networks deal with classification, and in that case having or not having access to a particular piece of information among thousands might cause slight differences in the predicted percentages for the different classes, but is unlikely to actually change much. In our case though, we are working with regression, and so every single little bit of information is important to us, as we are trying to calculate numbers that are as accurate as possible. An oversimplified example might be imagining that due to dropout we suddenly did not know the mass of a certain grid node, which would likely make it very difficult to determine the impulses for that grid node.

Similar to before, all weights were initialized using a normal distribution with a mean of zero and a standard deviation of 0.1, and the model was trained with a learning rate of 0.001.

Now, having described our two models in details, it is time to put the models to the test.

8 Results and Discussion

Having gone over the physics and the simulator simulating it, the models attempting to emulate it, and the way to transfer data between the two, we are now ready to actually try and run the whole thing and see what kinds of results we get.

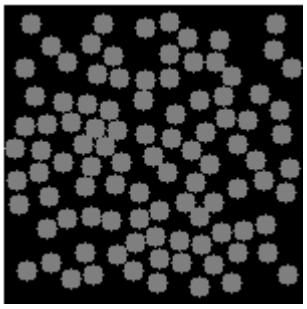


We will start off by talking about the data that we have generated, and how we have done so, in section 8.1. In section 8.2 we will then introduce a few, simple, additional models that we have created, which are not based on neural networks but only rely on the simulator itself. We will use these, and the results of testing their performance, partly as baselines for comparison with our neural network models and partly to get an idea of what we can expect from the simulator performance-wise. We will also use this section to introduce a few of the types of plots we have created. In section 8.3 we will present the results of testing the performance of our neural network models, and finally in section 8.4 we will then briefly discuss these results.

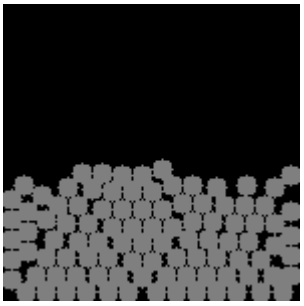
8.1 Data Generation

First things first, let us talk about data generation. We have two different sets of data; one that we use to train our neural network models and one we use to measure their performance. Both sets consist of similar simulation runs, with the simulation runs for training the neural network model being generated in advance and the data then transfered onto grids, and the simulation runs used for performance measurements being generated as needed.

Each simulation run is based on a world which consists of a static box, and a fixed number of dynamic bodies inside this box. Before the simulation begins, the box is created and the bodies are placed randomly inside the box using a random number generator with a seed of our choosing. This seed allows us to easily recreate a world setup when needed. After setting up the world, we then ask it to take a specific number of steps, and for each step we record the values of various body and contact attributes as well as impulse results from contact handling.



(a) Step 0



(b) Step 250

Figure 19: A visualization of step 0 of a simulation from our data set, and step 250 from the same simulation. The static box is positioned at the edges of the plots and not visible. The bodies do not seem entirely circular, but this is only because of the low resolution of the plot.

During these simulation runs, what will happen is that the bodies fall under the influence of gravity and create a pile at the bottom of the static box. Examples of the initial world setup as well as the world after two hundred and fifty steps are shown on Figure 19. While a simulation is running, we then collect various information about the bodies and about all of the contacts for each step. For the bodies we store information about mass, position, velocity and angular velocity. For contacts we store information about position, what bodies the contact involves, contact normals and contact impulses.

In order to create training data for our neural network models, we then first take all of the information that we collected during the simulation run and store it as xml. We then load the xml and transfer the data onto a set of grids with the kind of properties that we want, such as resolution and support radius, using the methods described previously, and we then store these grids. Originally we decided on storing the data as xml, as we then expected that we could simply load the data and transfer it to grids when needed. The reason for doing the transfer to grids as part of the data generation instead, is the time it takes and the fact that we might want to use the same grid for training more than one model, and doing the transfer repeatedly would be a waste.

When we are measuring the performance of our models on the other hand, we will create a world and run it similarly to how we did before. The difference is that for each step, all of the information that we collect will be passed to whatever model we are currently using, and the model will then be expected to provide a starting iterate for each contact. Our neural network

models will do this by transferring all of the data onto a set of grids, use these to predict a set of impulse grids, and then transfer the impulses from these grids and back to the contacts as starting iterates. During performance testing we will also be collecting all of the performance related information that we have modified the `Pybox2D` simulator to provide, such as impulses and iteration numbers.

Now for the specifics. For each world, the static box has dimensions 15 by 15, and consists of a bottom and two sides, with no top as it is not needed. Inside this box we randomly distribute 100 bodies, each with a circle shape, a density of 1 and a radius of 0.5. The coefficient of friction is set to 0.5, the coefficient of restitution is set to 0.0 and the gravitational acceleration is set to 9.8. Static bodies in `Pybox2D` does not have any mass, as they are effectively treated as having infinite mass, so when transferring the mass of all of the bodies to a grid, we assign the grid nodes marking the edge of the grid where the static box is placed a mass of 100, which is more than one hundred times the mass of the bodies.

When running the simulation, each step represents 0.01 seconds and a total of 250 steps are taken, which means that each simulation simulates 2.5 seconds. The maximum number of iterations for each step is set to 1000, and the impulse threshold is set to 10^{-4}. For each model we want to test, we run 10 such simulations, making sure to use specific seeds such that all models are tested on the same set of worlds.

All grids are created with a resolution of 0.25, both in the x- and y-dimension, and a support radius of 0.5 is used for the SPH method when transferring to and from the grids. Both the start and the end points of the worlds are included, which means that the grids will be 61 by 61.

As input to our model we use six attributes, each with their own grid. From the bodies we get four attributes; mass, velocity in the x-direction, velocity in the y-direction and angular velocity. These four quantities tells our model more or less all that it needs to know about the bodies and their movement. For contacts we use two attributes; the x-component of the contact normal and the y-component of the contact normal. These values first of all tells the model where the contacts are, and it also tells it something about the likely direction of the impulse.

As output from our model we get two attributes, which are of course the normal and tangent components of the impulse, each with their own grid. When training the model, we will then use similar grids created with the information we gathered when creating the simulation training data.

In total then, the input to our model is 6 by 61 by 61, or 6 by 61^2, and the output is 2 by 61^2.

8.2 Simple Models

Now, in order to understand the performance of our model, we first need something to compare the performance too. We might also want to do some tests to get an idea of what the limits are, both upper and lower, for how well our model could perform. For this we have made a number of fairly simple models, that do not use neural networks, which we will go over first, while also taking the opportunity to introduce some of the types of graphs we have created.

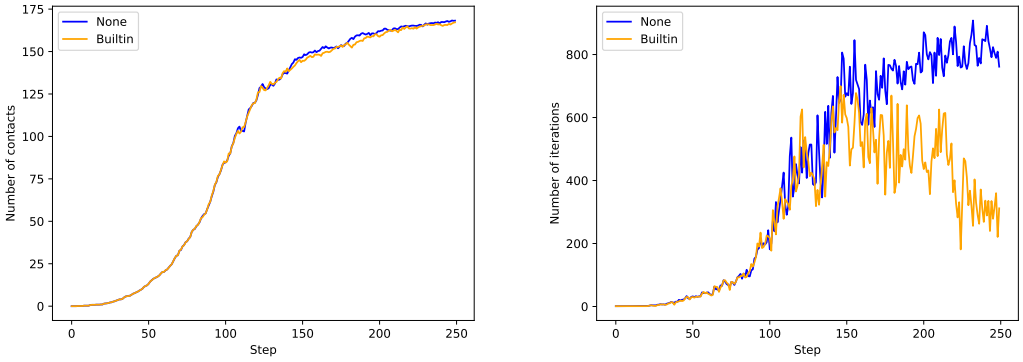
8.2.1 `None` and `Builtin`

The simplest baseline to compare our model to would be one using no warm start at all, i.e. using a warm starting iterate of zero for all contacts and all steps. This is what we call the `None` model. Another simple model to use for comparison, is to simply use the built-in warm start, which for each contact uses the result from the previous step, if any, as a starting iterate. We call this model the `Builtin` model. Let's start by looking at the results of testing these two models.

Figure 20a shows the average number of contacts for each step over the ten simulations. This is the actual number of pairs of bodies in contact with each other being solved during the step, as opposed to the number of pairs of bodies whose axis-aligned bounding boxes are in contact, which is all that is needed for `Pybox2D` to call something a contact. As we expect the number of contacts goes up as the simulation is running, corresponding with the bodies falling, hitting the box and piling up. The numbers tells us that towards the end of the simulation, each body is in contact with roughly three and a half other bodies, as each contact involves two bodies, which makes sense for a pile of bodies.

We note that the number of contacts is actually not entirely the same for the two models. It is not surprising that using different starting iterates actually leads to slightly different end results, but it seems a little surprising that these small differences actually adds up across multiple entire simulation runs to result in different contact counts. The differences are small though, and so we will not worry too much about them. The number of contacts will also be very similar for all the other types of models we will introduce and use, and so we will not bother with repeating this plot each time.

On Figure 20b we can see the average number of iterations spent on determining the impulses for the velocity part, for each step over the ten simulations. For the `None` model, we see that the number of iterations increases steadily, quite similarly to what we saw for the number of contacts.

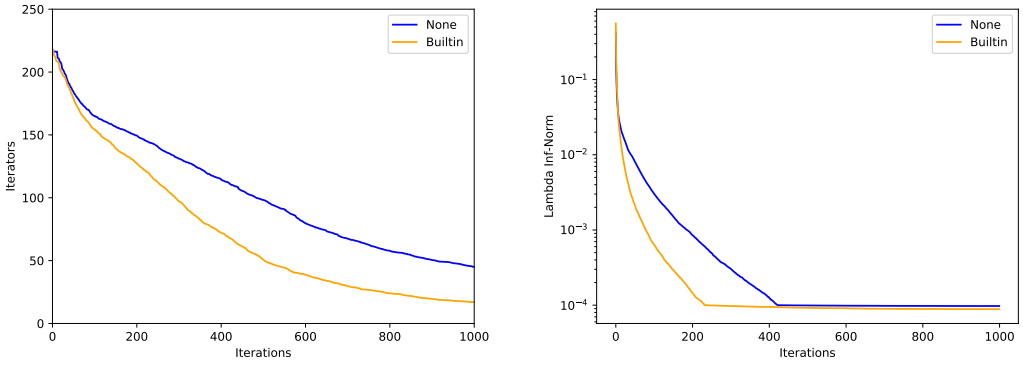


(a) Contacts (b) Iterations

Figure 20: (a) Average number of contacts for each step and model. (b) Average number of iterations required to determine impulses for each step and model.

This makes sense, as the increasing number of contacts during the pile-up of the bodies means that it becomes harder to solve each contact, as there are more and more other contacts to take into account. Meanwhile, for the `Builtin` model we see similar behavior for the first part of the graph, but then for the second half the model performs a lot better, needing only around a third of the iterations towards the end compared to the `None` model. This also makes good sense, as towards the end of the simulations the bodies will be more or less at rest in a big pile, which means that the results from solving all the contacts for one step should be very similar to the results of the next step, which makes it a good warm start iterate. This type of plot is good exactly for that reason, in that it allows us to easily see at what parts of the simulation a particular model is strong, and at what parts it is weak, which might help us know what to focus on to improve the model.

Next up then is another set of plots. Figure 21a shows a plot of the average number of iterators left for each iteration over the ten simulations. This is the based on the same numbers as Figure 20b, only the numbers are used differently. Each simulation consists of 250 steps, and each step takes a certain amount of iterations to determine the impulses. What we are showing on Figure 21a is then how many steps used at least x number of iterations, where x then ranges from zero to the maximum allowed number of iterations, which we set to one thousand. We note that the numbers are generally falling for both models, while being lower for the `Builtin` model



(a) Iterators (b) Convergence Rates

Figure 21: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

than for the `None` model, which makes sense as we expect it to perform better, i.e. finish iterating faster. We note that despite each simulation consisting of 250 steps, there are never 250 iterators, which is due to the fact that some of the steps at the beginning of a simulation will simply have all the bodies falling, with no contacts yet occurring, and so there is nothing to solve. We will generally be using this type of plot rather than the iteration counter plot from before, as this type is more closely connected to the next type of plot we will look at.

Figure 21b shows the convergence rates for the infinity norm of the impulses, or lambdas. Once again the data comes from running ten simulations for each model, with each simulation containing 250 steps. Each of these steps then results in a convergence rate, which details what the infinity norm of the highest change in impulse among all contacts was, for each iteration. The convergence rate shown on Figure 21b is then produced by taking the median, and not the mean, for each iteration among the $250 \cdot 10 = 2500$ steps. Now, some of the steps will of course finish iterating in less than the limit of one thousand iterations, in fact most will, as we saw on Figure 21a. We could then simply exclude any steps finished iterating from the calculation of the median in the later steps, but this leads to fairly strange plots as it quickly leads to the bad convergence rates dominating the plot, and so for any step that finish iterating we simply repeat the final value for the remaining iterations. This is the reason why the curves flatten out when they get below the threshold of 10^{-4}, since it means that more than 50% of the steps have finished iterating. This is also the connection with the iterator plot on

Figure 21a, as the curves on Figure 21b flatten out at the same iteration as the corresponding curve on Figure 21a goes below $250 \cdot 0.5 = 125$.

From the plot on 21b we see a similar story to what we did before, with the `Builtin` model having a convergence rate where the median reaches below the threshold in a little over two hundred iterations, while the `None` model takes almost twice as many iterations. As the plot gets rather boring when the curves flatten out, we might cut off part of the plot from now on, and for instance only include the first 500 iterations.

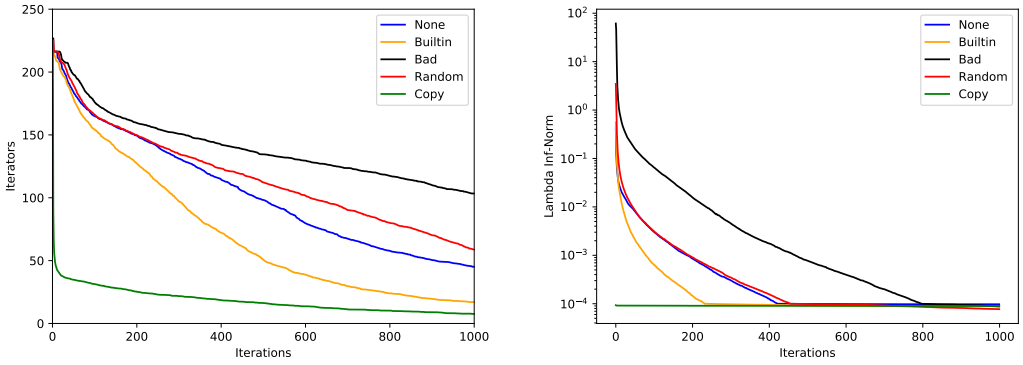
8.2.2 `Random`, `Bad` and `Copy`

Having considered two simple baselines for warm starting in the form of the `None` and `Builtin` models, we have seen that using good warm start iterates does in fact improve the convergence rates for the simulator. Next up we will then look at three different cases; one where we pick random starting iterates to see if that is better than using zeros, one where we try to pick bad starting iterates to see if that is possible, and one where we try to pick as close to optimal starting iterates as we can reasonably get with a simple model.

First off, we will try picking random, reasonable starting iterates, in order to see whether using any starting iterate other than zero improves the performance. This is the `Random` model, which for any contact randomly predicts a tangent impulse between -2 and 2 and a normal impulse between 0 and 5. These ranges of values were picked by very quickly looking through the data and the impulses being generated. Currently the numbers are chosen randomly from within the intervals, a better solution would probably have been to use some form of distribution, for instance a normal distribution with a set mean and standard deviation, since the impulses are often quite small and only rarely actually go above 1 or below -1. This model is interesting because if it performs well, then it tells us that even if our neural network model performs well later on, it might just be picking and predicting random numbers, which would probably not be what we intended.

The second thing we want to check, is whether it is actually possible to pick starting iterates that are worse than simply using zeros, and if so how much it impacts the performance. For this we have created the `Bad` model, which for any contact simply predicts a normal impulse of 50 and a tangent impulse of 50. This model is interesting because it shows whether or not we will actually be able to tell if our model is predicting very bad starting iterates, or whether it will simply look like the model is predicting zeros.

Finally we want to see how much we can improve the performance by

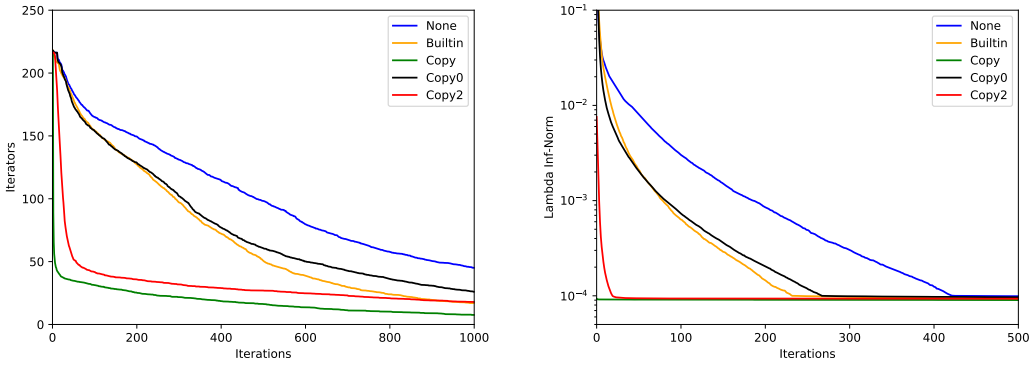


(a) Iterators (b) Convergence Rates

Figure 22: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

providing the best starting iterates we can get with a simple model, in order to see how much room there is for improvement compared to predicting zeros or using the results from previous steps. For this we have created the `Copy` model, which for each step creates a separate copy of the world, tells the copy to take a step and records all of the resulting impulses determined during that step. It then uses these results as warm start iterates when taking a step with the original world. We note that the way the simulator is set up, no matter what starting iterates it gets it still has to do at least one iteration across all contacts, and this will sometimes result in changes to one or more impulses that are larger than the threshold, which means that the simulator is forced to do further iterations, although this is a rare thing and most of the time it will simply do a single iteration and then still be below the threshold and stop iterating.

The results for these three, new models can be seen on Figure 22. The first thing we note is that the `Bad` model is indeed performing very poorly, with less than half the steps finishing iterating with the limit of one thousand iterations on average, and the convergence rate being an order of magnitude worse than what we saw previously. The second thing we note is that the `Copy` model is significantly better than all the others, with roughly 80% of the steps finishing iterating in just a few iterations and a convergence rate that is completely flat and below the threshold right from the start. Finally we note that the `Random` model is performing about as well as the `None` model, with numbers being slightly worse.



(a) Iterators (b) Convergence Rates

Figure 23: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

In total then, from this we gather that it is actually possible to increase the performance significantly using good starting iterates, which is of course what we would aim for with our neural network model. At the same time, we also see that using bad starting iterates leads to very poor results, and so we would hope to avoid this. Lastly we know that if our neural network model performs roughly equally to the `None` model, then it is not only no better than picking zeros, it might in fact just be making predictions using a random number generator.

8.2.3 `Copy2` and `Copy0`

Having determined that good starting iterates can make a big difference, another thing that we would like to know then is how the quality of our starting iterate, i.e. how close it is to the result, affects the performance. To do a quick check of this we have created two models; `Copy2` and `Copy0`. Both models work in the same way as the `Copy` world, i.e. by creating a copy of the world, asking the copy to take a step and then using the results as starting iterates in the original world. The difference is that these models modify the results from the copy before using them as starting iterates. As a reminder, in our simulation runs we use a threshold of 10^{-4} to decide when to stop iterating, which means we focus on changes in the first four decimals. The `Copy2` model rounds all starting iterates up to having only two decimals, and the `Copy0` model rounds them up further to having zero, effectively turning them into integers. In this way, the models use starting iterates that are somewhat close to the ones used by the `Copy` model, being off by only a set

number of decimals.

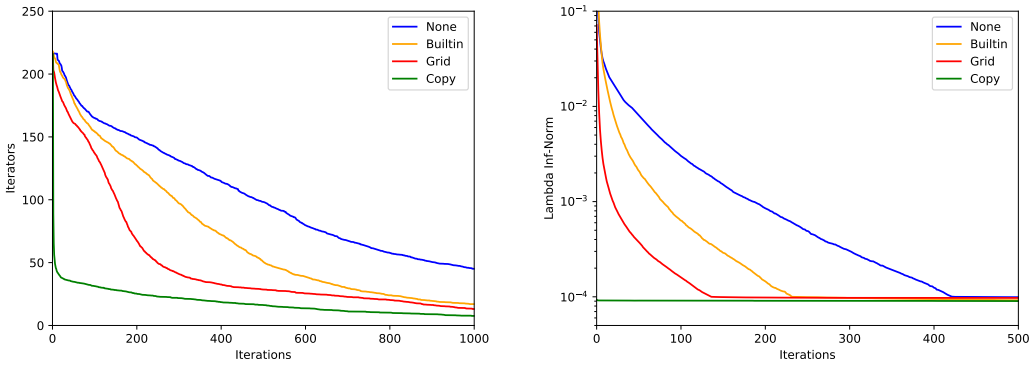
On Figure 23 we can see the results for these two models, together with a few of the old. From this we see that the `Copy2` model actually performs almost as well as the `Copy` model, while still significantly outperforming the `None` and `Builtin` models. About 80% of the steps are done iterating after roughly fifty iterations, as opposed to roughly ten iterations for the `Copy` model. Meanwhile the `Copy0` model looks to be performing about as well as the `Builtin` model, which is rather surprising considering it is simply predicting integer starting iterates for a set of simulation where the impulses are smaller than one in most cases.

In total then, these plots seem to tell us that our neural network model does not need to be extremely precise in its predictions, as long as they are fairly accurate. This is good news, as it should make it easier for us to come up with a model which performs decently, even if it can not compete with the `Copy` model.

8.2.4 `Grid`

A final thing we want to check is how transferring data between the simulator and our neural network by the use of grids affects the performance. This is of course a fairly difficult thing to check thoroughly, but just to get an idea we have created the `Grid` model. The model creates a copy of the world and uses it to get the impulse results similarly to how the `Copy` model worked, and then it transfers those results on to two grids, one for the normal impulse and one for the tangent impulse, using our SPH method. When generating warm start iterates for the original world it then transfers the impulse values back from the grid to the particles again. In this way, the difference between the performance of the `Grid` model and the `Copy` model should give us at least an idea of how much we lose by transferring to and from grids.

On Figure 24 we can see the results for the `Grid` model, which are slightly worrying. Where as for the `Copy` world we saw that about 80% of the steps finished iterating in around 10 iterations on average, for the `Grid` model it takes closer to 300 iterations. Similarly we see that the convergence rate is significantly worse. Although just a very simple test, it does seem to suggest that we will indeed lose performance due to the transfer to and from grids. On the other hand, the `Grid` model still performs markedly better than the `Builtin` model, which gives us hope that our neural network model can compete with the built-in warm starting procedure if nothing else.



(a) Iterators (b) Convergence Rates

Figure 24: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

8.3 CNN Models

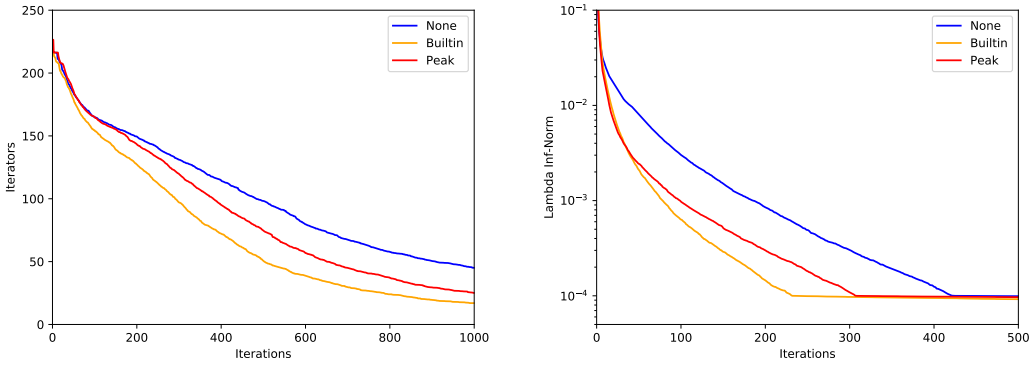
Finally we get to the convolutional neural network models. These have been tested similarly to the simple models we looked at above, and the results can be seen below.

8.3.1 `Peak`

We start with our own `Peak` model. On Figure 25 we see that our model unfortunately seems to perform slightly worse than the `Builtin` model, although still better than the `None` model. It takes roughly eight hundred iterations before 80% of the iterators are done, and roughly three hundred before 50% are done. The convergence rate seems to follow that of the `Builtin` model initially, but then starts to fall behind.

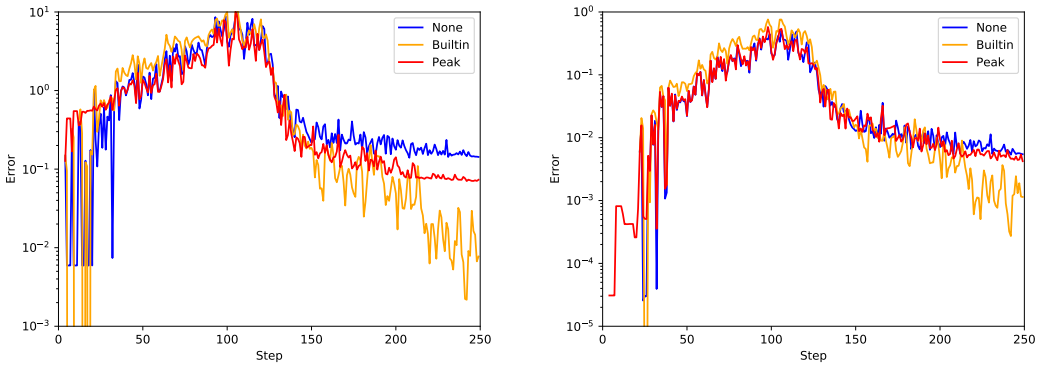
We have then also created yet another type of plot, which shows the difference between the warm start iterate and the final impulse value when the iteration process is done, for each contact for each step. Specifically we measure the mean squared difference between each of the components of these two impulses. The results can be seen on Figure 26.

On Figure 26a we see that the normal component of the differences between the starting iterates and the final results are large for our model, relative to the other two models, for the first number of steps. This seems a bit weird, as we would have expected the model to perform better when there are fewer contacts. On the other hand, for the later part of the simulation, when



(a) Iterators (b) Convergence Rates

Figure 25: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

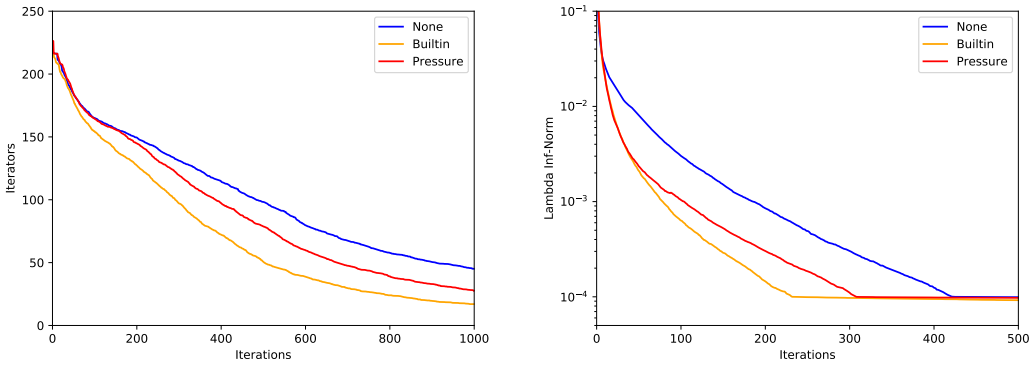


(a) Normal (b) Tangent

Figure 26: MSE between starting iterate and final result, with (a) showing the normal component and (b) showing the tangent component.

all the bodies are piled up, the difference seems lower for our model compared to the `None` model, although not quite as low as the `Builtin` model, which makes sense as we already noticed earlier that the `Builtin` model is particularly good at this stage.

On Figure 26b we see that our model seems to be very bad at predicting the tangent component, having more or less the same values as the `None` model, which likely indicates that our model is simply just predicting something very close to zero for all contacts. The tangent component of the impulses are generally one or more orders of magnitude lower than the nor-



(a) Iterators (b) Convergence Rates

Figure 27: (a) Average number of iterators left after each iteration. (b) Impulse infinity norm convergence rate.

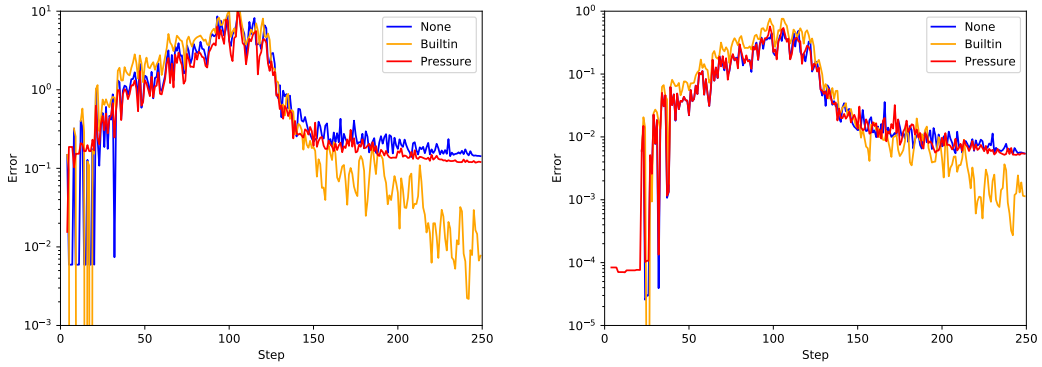
mal components though, so this is not quite as critical, although it is still concerning, and is probably because the model prioritizes it less for the same reason.

8.3.2 `Pressure`

Next up we will look at the performance of our modified `Pressure` model. We have measured the performance of the model similarly to how we did for the others, and the results can be seen on Figure 27 and Figure 28.

The first of these is slightly surprising in that the performance in terms of number of iterators and convergence rate is very similar to that of our `Peak` model, once again ranking somewhere between the `Builtin` and the `None` models. The convergence rate once again seems to follow that of the `Builtin` model initially, before falling off later on.

On Figure 28 we see that for the tangent component of the difference the results are also very similar to what we saw for the `Peak` model, with this model also doing very poorly. The reasons for this are probably the same as before. For the normal component the results seems to be similar for the first part of the simulation, while it seems significantly worse for the second part, with the difference being more or less the same as that for the `None` model.



(a) Normal (b) Tangent

Figure 28: Mean squared difference between starting iterate and final result, with (a) showing the normal component and (b) showing the tangent component.

8.4 Discussion

Finally, let us take a look at the overall results.

The results for all of the simple models are encouraging and seems to indicate that it is indeed possible to make the contact solving iteration process significantly faster by providing good starting iterates, and also that good starting iterates can make a difference even if they are not too close to the final solution. They also showed that transferring data to and from grids are likely to have a negative impact, but in total we would expect that our neural network models should be able to significantly improve the iteration process, as long as they can predict sufficiently good starting iterates of course. For the two neural network models though, we get very similar performance that places them somewhere between the `Builtin` and the `None` models, which is fairly disappointing. The models does not seem to be able to produce good enough starting iterates to really make a big difference.

As for why the performance of our two models is so similar, and not very good, there are a few reasons that we can think of. First off, we would like to note that we have done a few limited tests of the performance impact of changing various parameters for the neural network models, such as changing the number of layers, filters, elements, kernel sizes etc. The results of these test show that changing these number do indeed change the performance, as one would expect, and so the models tested above use the best values of these parameters that we have found. This also means that although the

performance of the two models seems very similar, we could easily change some of these parameters in order to make one or both of the models perform worse. For instance, for the `Pressure` model, something as simple as adding an extra round of convolutions to the first part of the model, bringing the total up to three, and changing the kernel sizes for the model to five leads to an instance of the model which performs more or less as bad as the `None` model.

The results might not be very good then simply because our models aren't very good. There are a near infinite number of ways to build a neural network, and even when you have decided upon a design there are typically still a lot of different parameters that can be set and fine-tuned to further improve the model. Building a neural network then is very much a process of trial and error, and having some previous experience from working with neural networks is very much an advantage that can help you reach your goal faster. We have no such experience, in fact personally this is the first time that I have ever had hands-on experience with working with neural networks. As such, it might be that we needed to spend more time on designing and improving our own `Peak` model. Similarly, the `Pressure` was originally designed for a different situation, and although that situation might seem similar to our, there are also significant differences in how it was originally used.

As for the similarity of the results, it might simply be because our models are so similar. They do both use the same idea of having a number of parallel layers of convolutions and poolings for creating and gathering data, followed by a linear set of layers for processing the information and creating the output. Perhaps we needed a completely different design of our neural network in order to get good results. It might also be that we have simply hit the limit, or a limit of a kind, for how well a neural network can mimic physics, at least when combined with all the transferring of data to and from grids and the `Pybox2D` simulator. The `Grid` model certainly seemed to indicate that this is a limiting factor to some degree. It might also be that the simulator and the way it simulates physics has part of the blame, since it is a fairly simple simulator which was not originally made for the purpose of fast and accurate iterating.

Another thing to keep in mind is that the model is actually not exactly trained to produce good starting iterates for the simulator; it is trained to produce grids that are similar to those that it was trained with. This means that even though training the model does lead to the value of the loss function that we using declining, this only means that the model gets better at predicting such grids, but it does not directly mean that it gets better at predicting good starting iterates for the simulator, as we still have the whole grid-to-particle transfer between the two. Also, the grids that the model is

trained on are themselves introducing another source of errors, as they are produced by transferring the actual impulse results that we would have liked to use for training, from the contacts points and to a set of grids.

Of course the reason for our poor results might also be a combination of all of these reasons, and possibly more that we have not thought of, which seems like the most reasonable explanation.

One thing that certainly sets the models apart is the training process. Training time of course depends on the hardware you are using, whether you are using CPUs or GPUs etc., and so actual training time numbers are not necessarily very helpful or telling. We did most of our training on a server with 32 CPUs and 128 GBs of RAM. On an average computer, running on a CPU, the `Peak` model can probably be trained in something like ten to twenty hours. If you use a computer with a better CPU, or more CPUs, this number might improve, and if you move towards using GPUs you might be able to improve it significantly. In comparison to this, training the `Pressure` model was generally something like one hundred time faster, which as mentioned previously is because it only has to train a few thousand weights, as opposed to the `Peak` model which has to train tens of millions of weights. Another reason for the big difference is also that the `Pressure` model could generally be trained using something like twenty sets of data, where as the `Peak` model typically needed around one hundred to one hundred and fifty sets.

Another issue regarding run times is the actual simulations. The kind of simulations we used above simulated two and a half seconds each, which the simulator can more or less do in two and a half seconds. This means that the simulation can be run in real time, which is of course desirable. Using either the `None` model or the `Builtin` model does not affect this, whereas the `Copy` models do take longer as they are effectively running two simulations simultaneously, and the `Grid` model takes even longer as it has to transfer data to and from grids. Our two neural network models not only have to transfer data to and from grids, they also have to actually run the data through the neural networks and produce a prediction, and they need to do this once for each step, which means two hundred and fifty times per simulation. For this reason, a two and a half second simulation takes around one minute using the `Pressure` model and around two minutes using the `Peak` model, which is of course far from ideal. This is something that would need to be improved if the models were to actually be used in a real simulator.

9 Conclusion

Firstly, we have implemented a pair of methods; one for transferring data from a set of objects to a set of grids, and one for transferring data from a set of grids and back to a set of objects. Our limited testing of these methods show that they work as intended, as long as one is careful about picking suitable values for the involved parameters such as the grid resolution and the support radius. We have not directly tested the efficiency of our implementation, but from our experience with using the two methods in combination with the simulator and our neural network model, we can conclude that the process of transferring to and from grids is fairly slow, and would have to be significantly sped up if it was to be used for any realistic purpose.

Secondly, we have designed and implemented a convolutional neural network, which we have then trained on sets of simulation data to be able to predict the kind of starting iterates we want, given information about the state of the simulation world. Our testing of the model shows that it is indeed able to predict such values, and using them in the simulator does indeed improve the iteration process, but unfortunately not as much as we had hoped. In particular, the values that our model predicts are not as good starting iterates as those determined using the method already built into the simulator that we have used.

The results of the tests of our model does not seem to favor the idea of using neural networks for generating starting iterates as we had hoped, although we can by no means definitively rebuff the idea. The poor results could be a result of us having designed a bad model, perhaps as a result of not having any prior experience in designing neural networks; it could be because the specific simulator that we have used was not ideal for this kind of project; it could be that transferring data to and from grids is a bad idea and causes us to loose too much information, and it could be a combination of all of these and more.

References

[BPL+16] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *CoRR*, abs/1612.00222, 2016.

[Cat06] Erin Catto. Box2d - a 2d physics engine for games. `https://github.com/erincatto/Box2D`, 2006.

[ESHD05] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-Based Animation*. Charles River Media, Hingham, Massachusetts, 1st edition, 2005.

[GTH98] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 9–20, New York, NY, USA, 1998. ACM.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[JL18] Yifeng Jiang and C. Karen Liu. Data-augmented contact model for rigid body simulation. *CoRR*, abs/1803.04019, 2018.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[Lau08] Ken Lauer. Pybox2d - 2d game physics for python. `https://github.com/pybox2d/pybox2d`, 2008.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, 2003.

[Mon92] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574, 1992.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[TSSP16] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. *CoRR*, abs/1607.03597, 2016.

[WTW+17] Nicholas Watters, Andrea Tacchetti, Theophane Weber, Razvan Pascanu, Peter Battaglia, and Daniel Zoran. Visual interaction networks. *CoRR*, abs/1706.01433, 2017.

A The Code

The code can be found on GitHub at `https://github.com/LukasEngedal/Deep-Contact/releases/tag/v1.0`, which is forked from the GitHub repository where Jian Wu, Lucian Tirca and I originally shared the work we did on the initial parts of the project. I created a separate fork due to the significant number of changes I made to a lot of the code in the last months of the project, which I decided not to upload to the shared repository in order to avoid causing trouble for the other two participants and any code they may have written and not uploaded themselves.

The main folder contains two subfolders, `pybox2d` and `src`. The `pybox2d` folder contains our modified version of the `Pybox2D` simulator, which can be installed by running the `setup.py` twice, first with the flag `build` which will build the code, and secondly with the flag `install` which will install the simulator in Python. The `Pybox2D` code is written mainly in `C++`.

The `src` contains all of the code that we have created, which is organized rather haphazardly into another set of folders. The code is structured as a number of separate files containing the various functions that we have created, and another set of files intended to be used and run as scripts. The intended use of these files is to pick the one you want to run, open the file, edit the various parameters defined therein to fit your particular needs, and then run the file in a terminal. All of our code is written in Python 3.6, and requires the latest versions of Numpy, Scipy, OpenCV, Pandas, TensorFlow, TensorBoard and MatPlotLib, as well as having `Pybox2D` installed.

Let us assume that the intention is to test the full functionality of the code; to train and run a neural network. Currently all of the relevant script files are set up so as to create training data, create a neural network, test the neural network and plot the results, in the exact way that we used to create the `Peak` model and the results for this model.

The first file to run is `generate_xml.py`, found in the `gen_data` folder. This file will generate 110 sets of training data as xml files, 100 for training the network and 10 for validation. The next file to run is `generate_grids`, in the same folder, which will load the 110 xml files, transfer all of the data to sets of grids and then store the grids. In total this should take around an hour or so for the average pc.

The next step is then to run the `run_training` file found in the `TensorFlow` folder, which will create and train the neural network. Note that this will potentially take a long time, something like 10-20 hours on an average pc running the code on a CPU. Currently TensorFlow is told to train the neu-

ral network using the CPU, this can be changed by outcommenting line 83 in `peak.py`, causing it to run on the GPU instead. This of course requires that TensorFlow has been installed correctly to run on the GPU, which is not a trivial thing, and even then might not work if the GPU does not have sufficient memory for the large tensors that we are using for our dense layers. When training the neural network, the process can be monitored using TensorBoard.

After training the neural network, one then needs to run the performance tests with the model. This is done by running the `run_model` file in the `performance` folder. The results can then be plotted similarly to how we did in the results section by running the `plot_results` file in the same folder. In order to compare the results to that of other models, particularly the various simple models that we used, one has to first open the `run_model` file and change what model is used, by outcommenting the line setting the model to be the `Peak` model and then using one of the other outcommented lines instead. When all the desired models have been run, one similarly then has to open the `plot_results` file and specify what models to include in the plot.