REPLICATING IMPULSE-BASED PHYSICS ENGINE USING CLASSIC NEURAL NETWORKS

Rareș-Cristian Ifrim[59]
Patricia Penariu[60]
Costin-Anton Boiangiu[61]

Abstract

The high costs for creating and using traditional simulators induced both by technical effort, which extends over a long period, but also by the need for permanent updating during this period, to improve accuracy by using a limited range of settings, bring to the fore an alternative, namely: data-based methods for physical simulation; they are a much more attractive option for interactive applications in terms of their ability to trade precomputation and memory footprint for better performance while running. Also trained physics engines might come to offer better simulations as the networks can be configured to take as input data from real-world measurements, thus it can combine the best from the real-time simulation engines that emphases on fast simulations but do not offer a real-world-like simulation, and high accuracy simulation engines and targets real-world simulations but require high computational resources. This paper aims to construct a neural network that learns how the impulses between two objects react when they make a contact, by using an already implemented physics engine for generating the training datasets and to compare the results of the trained engine versus the original one. Although this has been done successfully, the proposed neural network managing to score a prediction rate with values between 55 and 89% depending on the test "scenario", improvements can be made to increase performance and to obtain a suitable accuracy (over 90%, even 95%), thus achieving the goal of completely replacing the physics engine.

Keywords: Neural Networks, Physics Engine, Collision Optimizer.

1. Introduction

State-of-the-art physics engines [1]-[3] use different equations of motions, or a combination of them to simulate how objects with defined parameters like mass and volume behave, especially in contact with each other. Some of these equations are Lagrange multiplier [4],

[59] Eng., Computer Science and Engineering Department, Faculty of Automatic Control and Computers, Politehnica University of Bucharest, Splaiul Independentei 313, Bucharest 060042, Romania, rares.ifrim@stud.fils.upb.ro
[60] PhD Stud., Eng., Computer Science and Engineering Department, Faculty of Automatic Control and Computers, University Politehnica of Bucharest, patriciapenariu@gmail.com
[61] Professor, PhD, Eng., Computer Science and Engineering Department, Faculty of Automatic Control and Computers, University Politehnica of Bucharest, costin.boiangiu@cs.pub.ro

Jakobsen [5], Impulse dynamics [6], and so on. These engines can also be split in two types: high-accuracy physics engines, and real-time physics engines [7].

The first type of physics engines has more complex defined equations of motions to simulate an environment as best as possible, but with the cost of high performance, also needed from the systems where the simulations are running.

The latter type is much faster [7], usually being used in video games or movies where the frame rates per second matter for a better user experience, but they also come with more inaccurate simulations, as they make predictions instead of complete calculations.

The idea to use neuronal networks and machine learning comes quite naturally, as once the network is trained, the predicted result is made almost instantaneous, especially on a parallel system like a GPU [8].

As a physics engine, in this paper it is used a lighter version of the *Box2D* [9], which is a simple rigid body engine that works only with simple 2D objects like squares and circles and implements the impulse dynamics equations of motion. This engine consists of three modules, namely: Common, Collision and Dynamics. The first module refers to code for allocation, math, and settings; the second module defines shapes, a broad-phase and collision functions/queries; and the final module gives the simulation world, bodies, fixtures and joints [9]. The main loop of the *Box2D-Lite* physics engine is illustrated in Fig. 1.

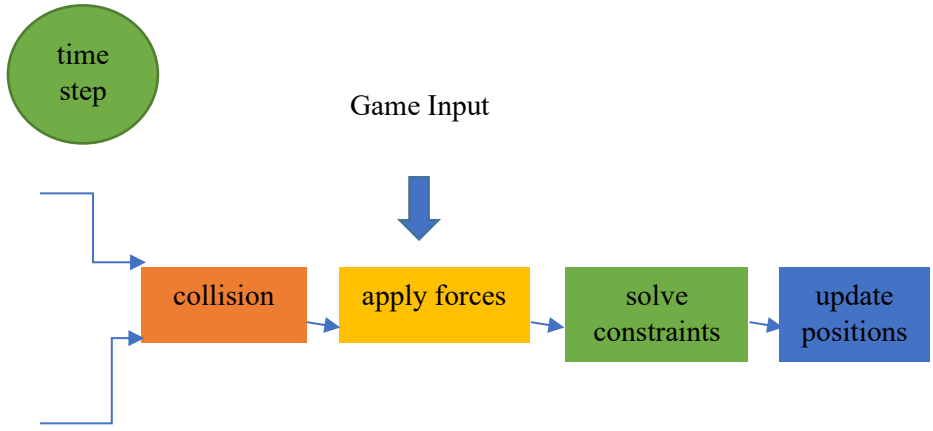

Fig.1. The main loop of the *Box2D-Lite* physics engine.

The collision detector comes with two phases: a broad phase, and a narrow one [9]. The broad phase finds pairs of overlapping boxes and creates arbiters for every pair. Each created, or updated arbiter performs a narrow phase collision, where it searches for the normal vector with the minimum penetration, as can be observed in Fig. 2.

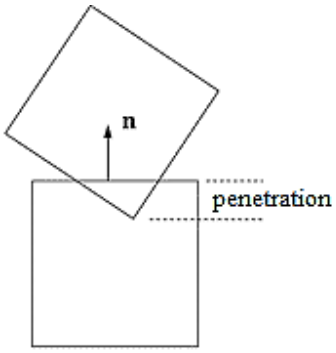

Fig.2. Narrow phase, box versus box collision.

The next step, after the collision detection, is applying the forces on the two collided bodies. In this step, the engine applies Newton's 2nd law for velocity and angular velocity on each object in the scene.

The third step is the "solve constraints" step, where the impulse between the two objects is calculated using Newton's 2nd and 3rd laws for the relative velocity at the contact point. From this step, the direction and magnitude of the impulse are obtained and thus the new instantly changed velocities of the objects after the contact can be calculated [10].

The initial velocity and angular velocity of the two objects that are about to collide are taken as inputs for the proposed network, and the "solve constraints" step that calculates the resulting velocity is replaced with the network that predicts these results. Several random tests are generated, with objects in different situations, for all these inputs and their corresponding *Box2D-Lite* engine outputs (the changed velocity and direction) a dataset is created which is then used to train the neural network.

2. Related works

In [11] data-driven methods simulate deformation effects, including external forces and collisions, allegedly 300× to 5000× faster than standard offline simulation, using the same approach for collecting training data through an offline engine and feeding it to the neuronal network.

Paper [12] describes a hybrid engine consisting of a standard physics engine for the new cases, and the "learned" engine for fast calculation of results for the "already met" cases. The standard physics engine takes new unseen inputs, calculates the output, and feeds the neuronal engine so that it can learn from the new experience. This process is illustrated in Fig. 3.

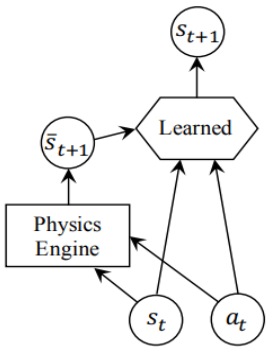

Fig.3. *SAIN* - A simulator-augmented interaction network, where s_0 is the initial state, a_t is the action at time t and $\underline{s}_t$ is the prediction by the physics engine at time t [12].

In [13] is trained an engine for fluid-based simulations using the authors' framework for neuronal networks called Graph Network-based Simulators (GNS). The aforementioned framework is used to perform particle simulations, by mapping every particle in a graph and using a messaging system in the graph between particles for exchanging energy and momentum among their neighbors. The accuracy of the computed predictions obtained by this engine, versus their ground truths, can be observed in Fig. 4.

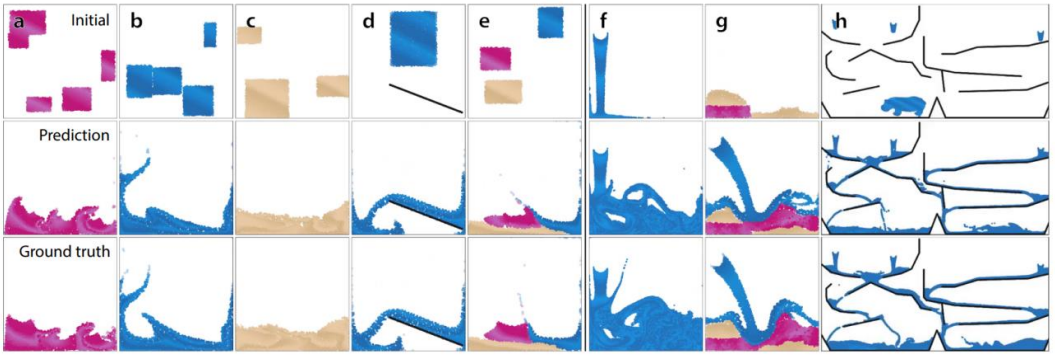

Fig.4. Prediction made by the trained engine versus the ground truth of the Navier-Stokes equations for fluids (**a.** "goop"- a viscous, plastically deformable material; **b.** water; **c.** sand; **d.** their interaction with rigid obstacles - *WaterRamps* domain; **e.** multiple materials and their interaction - *MultiMaterial* domain; **f.** high-res turbulence, trained on *WaterRamps*; **g.** multi-material interactions with unseen objects, trained on *MultiMaterial* and **h.** generalizing on larger domains, trained on *WaterRamps*); image taken from [13].

In association with [11], [12], [13], neuronal networks promise to deliver both accurate and fast simulation, which is a big improvement in many scientific domains. Also, the neuronal networks tend to be more prone to parallelization than some numerical methods that are used in standard simulations. They may use more naturally multi-core technologies like GPUs, which now come with special computing units for machine learning, an example, in

this case, being the latest generations of *NVIDIA* graphics cards equipped with tensor cores [14].

Considering the above, in this paper, it is shown that even simple engines like the proposed one, can be enhanced, as the network may receive training input from both the initial source and a more accurate one, so that simulations can surpass the original engine in both speed and accuracy.

3. Proposed solution

The proposed system uses a fully connected neural network fed with the raw velocities that the objects are having. As a proof of concept, a simple physics engine [15] that uses Newton's laws in applying impulses between colliding objects is employed. This engine is acting as a data generator for the neuronal network. Positions, velocities and angular velocities of the two objects before and after the collision are used, as these are the only values that the physics engine is manipulating when applying an impulse between them. After this, a fully connected neural network, with an input layer of 10 neurons (for the positions, velocities and angular velocities as all these values matter before a collision) and an output layer of 6 neurons (only for the resulting velocities and angular velocities of the two objects) is employed.

During the preliminary tests, there were analyzed and tested different configurations varying from one hidden layer and up to four hidden layers, with several neurons around the average between the input layer and the output layer. Once an accuracy of more than 90% is obtained, the network configuration is saved and loaded in the physics engine, thus replacing the traditional code where the relative velocity at contact is calculated.

A. Demonstrator application architecture details

For the network to be trained, it is necessary to input and output data that would correctly represent the logic used by the actual physics engine to calculate the collision between two objects and what the resulting velocities of these bodies are, after the contact.

Keeping this in mind, data was collected from the running physics engine under some demo tests ("scenarios") where objects were colliding randomly, and the positions of the two objects were extracted as input data, as well as their velocities and angular velocities between the impact, and as output data, the resulting velocity and angular velocity after the contact were extracted.

As this data is represented in the *XY* coordinates, the initial data was not very helpful for the neural network as the differences between two sets of inputs for example (and the same thing for the corresponding sets of outputs) were too big, and the neural network was failing to learn from them. So, another representation of the data was made by choosing to extract the difference between the positions of the bodies for input, and the relative velocity

resulted from the impact as output. Thus, the inputs and the outputs between two different collisions were a lot smaller and the neural network could learn from it.

The *Box2D* [15] physics engine was very helpful in generating the data because in a "scenario" where multiple objects may collide, the physics engine is splitting the whole scene into pairs of two objects colliding and updates for each pair the resulting velocities. An example for collecting data for training can be seen in Fig. 5.

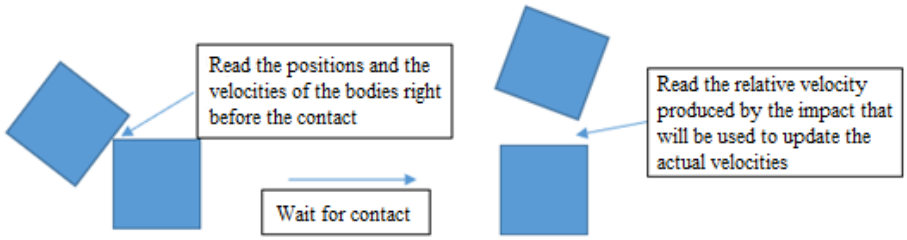

Fig.5. Generating training data from the physics engine.

The collected data is then divided by the *80/20* rule for training/testing of the neural network. From training it was discovered that the data could not be used in its raw format as it was again, not good enough for the neural network to learn. In its raw format, it was possible to obtain a maximum of 60% accuracy, even with different learning rates, activation functions used, or a different number of hidden layers and neurons per hidden layer.

One problem that needs to be taken into account is the presence of different intervals between the generated outputs, and the outputs the activation function was able to produce. The outputs generated from the "scenarios" used were somewhere between *[-300, 300]* range while classic activation functions such as *sigmoid* or *tanh* or *ReLU* (Rectified Linear Unit) were producing outputs that could not map the generated one. Although these activation functions may not produce the desired output on the last layer, they still can be used in the hidden layers. One approach for resolving this interval mapping would be to standardize the data [15], by calculating the mean value and standard deviation of the output and rescaling the distribution of values so that the mean of observed values is *0* and the standard deviation is *1*.

After the network predicts an output, one can reverse the data standardization process to bring it to the original output generated by the physics engine. Another step used to increase the network accuracy and minimize the loss function was to use the linear activation function on the output layer as this can map the desired interval.

This increased the accuracy to almost 90% without using any optimizers for the neural network hyperparameters. The next step to increase the accuracy as most as possible is adding optimizers on the neural network to help it converge to the global minimum for the loss function.

B. *Preliminary performance measurements*

Currently, without any optimizations on the neural network learning, the accuracy is under 90%. This may sound enough, but it is not, as the loss function for this accuracy is above 0.1 and the predicted relative velocity is still not close to the real one.

There was applied the same optimization techniques were applied in [12] and managed to minimize the loss function below *0.05* after only 1000 training iterations. Here, hyper-parameters like the number of neurons and the number of hidden layers could be adapted as the target is to go under a *0.001* loss function.

It was chosen a one hidden layer network of 10 neurons, while the input layer is also 10 neurons because of all the used input data, and the output layer is two neurons because of the velocity derivative used to calculate the final velocities of two objects colliding (the velocity is expressed in bot *Ox* and *Oy* axis). Both hidden and output layers use the classic sigmoid activation function.

Of course, one problem arises and that is the fact that the output generated by the physics engine can be in any floating interval of values (in the presented "scenarios" going from *-300* to *300* in the *(x, y)* coordinates), while the sigmoid function can output data in the *[0, 1]* interval. For resolving this problem, the output data is normalized before feeding it to the neural network for training from the minimum and maximum available values to the *[0, 1]* interval so that this new output can be compared to the output of a sigmoid function [15]:

$$y = \frac{x - min}{max - min} \tag{1}$$

In equation (1), the *x* value represents the initial output obtained from the generated data set, and *y* represents the normalized output that fits in the *[0, 1]* interval.

C. *Fine-tuning the neural network*

For achieving a smaller loss function some fine-tuning was needed to be applied to the training data set as this is the biggest factor that influenced the accuracy of the neural network. It has passed from using randomly generated data sets to well-defined generated data. This was done by selecting an interval of points where the launched object (called the "bomb") was generated. In the first iteration, this set of *(x, y)* points was generated randomly in the interval *[15, 15]* for the *Ox* axis. The *y* point was maintained fixed at the value of *15*. The angular velocity and rotation were also generated randomly.

With this approach, there was obtained loss functions close to the intended one, but in a realistic simulation of the neural network inside the engine (the logic that was used to calculate the derivative velocity at contact was replaced by the neural network), there were still cases where the network did not perform as intended. This is because of the random

nature of the generated data set that might have been generated more around some cases and less around others.

Keeping this in mind, randomly generated data sets were left behind and an iterative method that would cover a set of cases equally was chosen. This was done by iterating through the initial *Ox* interval of *[-15, 15]* with a small step of *0.05*. To help the network even more, this interval was reduced in *[-5, 5]* and incorporate two different "scenarios": one where a single box was hit by the “bomb” box, and one where a stack of 10 boxes was hit by the same “bomb” box (this meant that the “bomb” was launched from the same position for both “scenarios”).

Another constraint applied to the training phase was to go with a fixed angular velocity and rotation because the random nature of the generated ones produced the same outcome, where the network performed well for some cases and bad for others.

After applying these constraints, generating the training data upon them and training the neural network with the data set there were obtained loss functions in the range of *10-4* after 5000 iterations of training. The table below (TABLE 1) indicates the loss function resulted in a training cycle of 2000 iterations.

Iteration	Test Score	Train Score
0	0.102988	0.103005
10	0.051049	0.051049
50	0.017851	0.017834
100	0.008923	0.008907
1000	0.001305	0.001304
2000	0.000803	0.000800

TABLE 1. Loss function resulted in a training cycle of 2000 iterations with the described constraints for the training data.

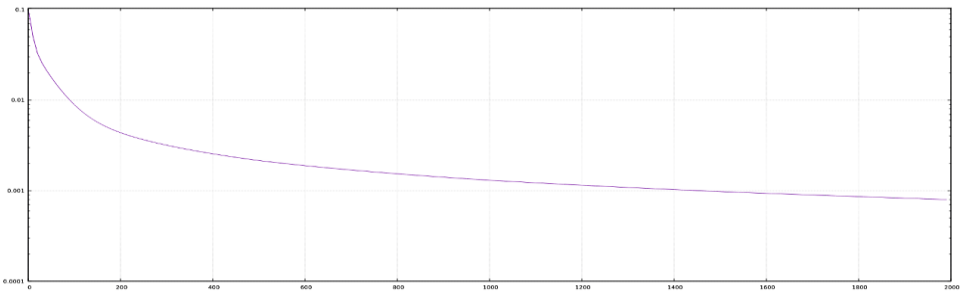

Fig.6. Evolution of loss function over the 2000 iterations with dynamic learning rate. On *Oy* axis is the loss function and on the *Ox* axis is the number of training iterations.

4. Results

In Fig. 7 it can be seen the two "scenarios" used for generating training data for the neural network (single object – first row and stack - second row). Along with these two, is a more complex "scenario" (pyramid – third row), all of these being later used as a reference for checking how well the network behaves after it has been trained.

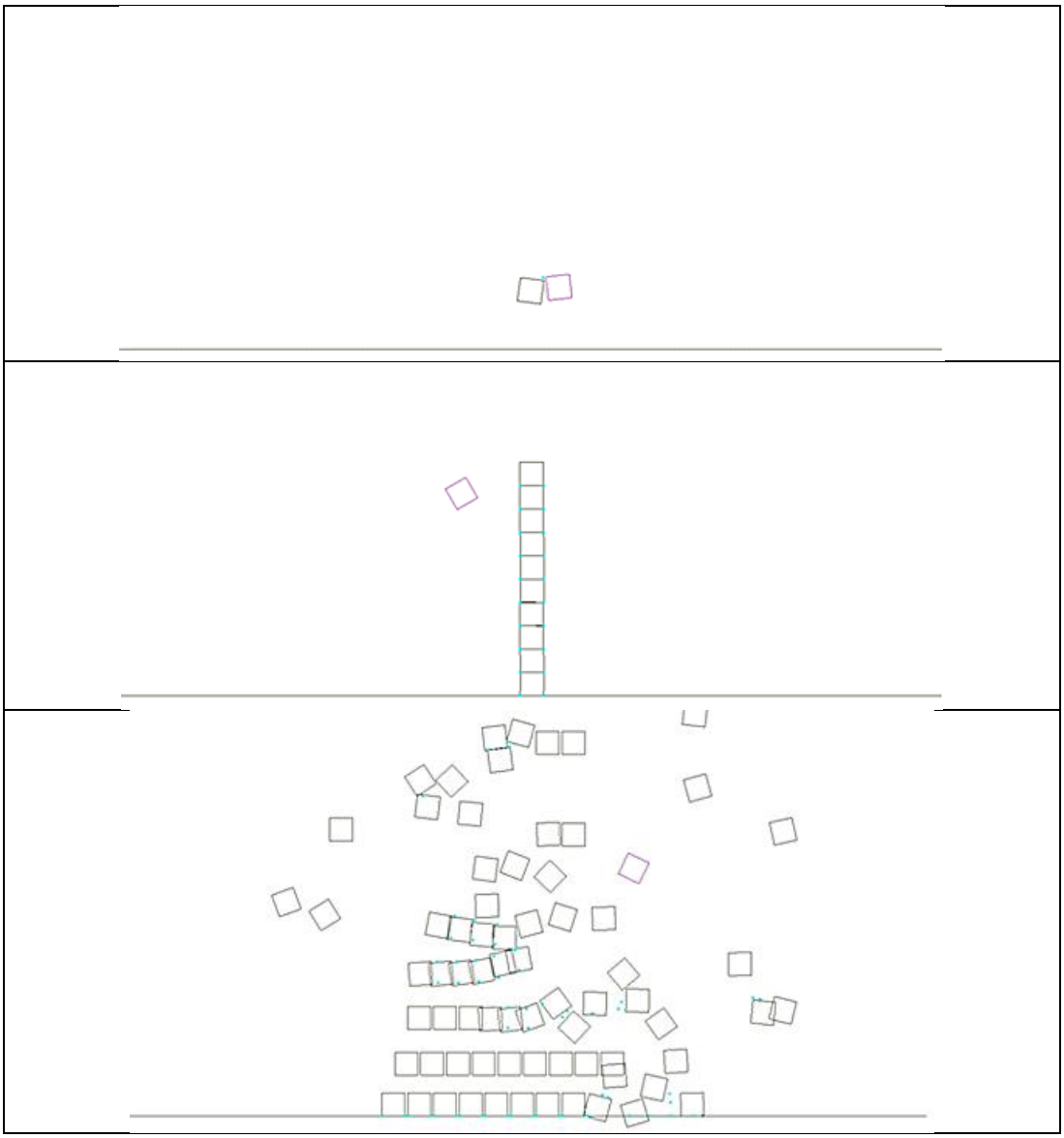

Fig.7. "Scenarios" denoted 1 (first row) and 2 (second row), that generate training data by simulating contacts between two objects ("scenario" 1) and an object and a stack of other ten objects ("scenario" 2); "scenario" denoted 3 (third row) consisting of a pyramid of 66 objects.

The generation of training data consists of launching an object (the purple square called the "bomb") into another object, or a stack of ten objects. The bomb starts from the position *(-5, 15)* and iterates through the final position *(5, 15)* with a step of *0.05*, getting data from both "scenarios" with the same position of the "bomb" object. The input data collected represents the positions of two objects that are about to collide (in Cartesian coordinates), and their velocities, angular velocities, and rotations. The output data represents the derivative (in Cartesian coordinates) that are then applied to the initial velocities before the contact resulting in the final velocities (and directions) of the objects after the contact.

The loss function is the MSE (Mean Square Error) function, and it is about *10-4* for the network. With this loss function value, still there are errors as it can be expected, because this value does not reflect an accuracy of 100% which would mean a perfect replica of the physics engine used for training.

The network also adds a performance impact on the physics engine. This is expected, as the neural network presents a higher level of computation than the original physics engine that used a simple Newtonian equation to calculate the derivative of the velocity. For simple "scenarios" like "scenario" 1 and "scenario" 2, this impact is not observed as both the physics engine and neural network obtain the same performance, but for a more complex "scenario" with a big number of objects interacting ("scenario" 3), the computation necessary for the neural network prediction is way bigger than what the traditional engine would calculate, and this has an observable impact on the graphical performance.

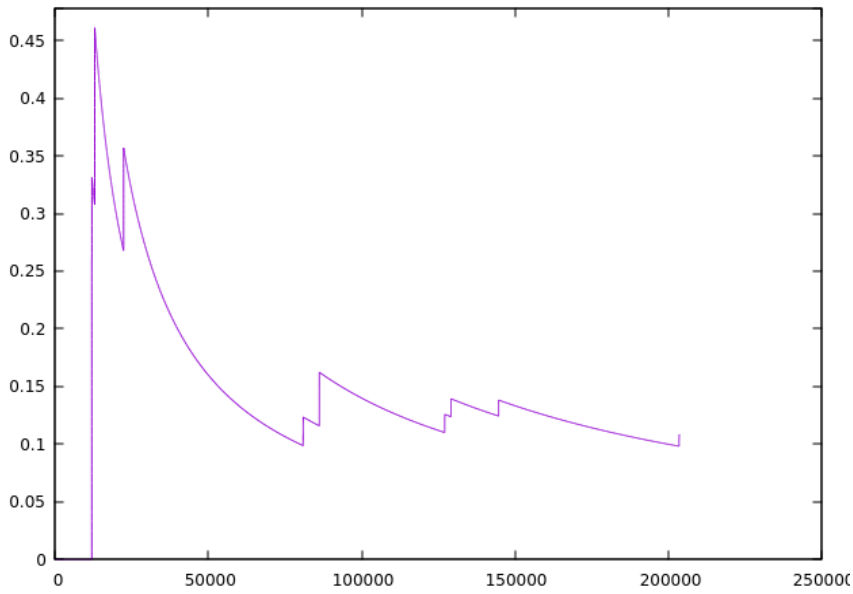

Fig.8. Evolution of prediction fail percentage over the total number of randomly generated simulation tests. On the *Oy* axis is the failure percentage of the neural network prediction and on the *Ox* axis is the total number of simulation tests.

As it can be seen in Fig. 8, the neural network achieves a successful prediction rate between 55% and 89% depending on the test "scenario". Unfortunately, this is not a desirable prediction rate, especially because there is not a constant percentage for all test "scenarios". This shows that classic neural networks are not suitable for this kind of application as it needs an accuracy of over 90% (even 95%) to be able to accurately replace the physics engine.

5. Conclusion

It is possible to use a neural network for replicating a physical engine in certain scenarios at an acceptable accuracy (the one obtained here could be improved by generating a bigger data set, by using a longer training time, and by tuning hyper-parameters, but the classic approach (especially the classic activation functions) cannot fully replace the physics engine. Also using a neural network just for predicting the collisions of objects for this case is not favorable as the performance is impacted and for a simple engine such as the presented one, it is better to use the traditional ways of expressing the physics equations.

Acknowledgement

This work was supported by a grant of the Romanian Ministry of Research and Innovation, CCCDI - UEFISCDI, project number PN-III-P1-1.2-PCCDI-2017-0689/„Lib2Life– Revitalizarea bibliotecilor și a patrimoniului cultural prin tehnologii avansate"/"Revitalizing Libraries and Cultural Heritage through Advanced Technologies", within PNCDI III.

References

[1] I. Millington, *Game physics engine development*, in CRC Press, 2007.
[2] E. Todorov, T. Erez and Y. Tassa, *MuJoCo: A physics engine for model-based control*, in: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp. 5026-5033, October 2012.
[3] J. Tompson, K. Schlachter, P. Sprechmann, K. Perlin, *Accelerating Eulerian Fluid Simulation With Convolutional Networks*, in ICML'17: Proceedings of the 34th International Conference on Machine Learning, volume 70, pp. 3424–3433, August 2017.
[4] D.P. Bertsekas, *Constrained optimization and Lagrange multiplier methods*, Academic press, 2014.
[5] J. Solsvik, H.A. Jakobsen, *The foundation of the population balance equation: a review*, in Journal of Dispersion Science and Technology, volume 36, issue 4, pp. 510-520, 2015.
[6] J. Bender, *Impulse-based dynamic simulation in linear time*, in Computer Animation and Virtual Worlds, volume 18, issue 4-5, pp. 225-233, 2007.

[7] Wikipedia, *Physics engine*, URL: https://en.wikipedia.org/wiki/Physics_engine, Accessed: 12 April 2021.

[8] A. Gajurel, S.J. Louis, F.C. Harris, *GPU Acceleration of Sparse Neural Networks*, arXiv preprint arXiv:2005.04347, 2020.

[9] E. Catto, *Box2D, a 2D physics engine for games*, URL: https://box2d.org/documentation, Accessed: 12 April 2021.

[10] E. Catto, *Modeling and solving constraints*, in Game Developers Conference, 2009.

[11] D. Holden, B.C. Duong, S. Datta, D. Nowrouzezahrai, *Subspace Neural Physics: Fast Data-Driven Interactive Simulation*, in Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 1-12, 2019.

[12] A. Ajay, M. Bauza, J. Wu, N. Fazeli, *Combining Physical Simulators and Object-Based Networks for Control*, in 2019 International Conference on Robotics and Automation (ICRA), pp. 3217-3223, IEEE, 2019.

[13] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, P.W. Battaglia, *Learning to Simulate Complex Physics with Graph Networks*, in International Conference on Machine Learning, pp. 8459-8468, PMLR, 2020.

[14] NVIDIA Cloud & Data Center, NVIDIA V100 Tensor Core GPU, URL: https://www.nvidia.com/en-us/data-center/v100/, Accessed: 12 April 2021

[15] E. Catto, *Fast and Simple Physics using Sequential impulses*, in Proceedings of game developer conference, 2006.

[16] *How to use Data Scaling Improve Deep Learning Model Stability and Performance*, URL: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/, Accessed: 12 April 2021.

[17] J. Brownlee, *How to use Data Scaling Improve Deep Learning Model Stability and Performance*, Machine Learning Mastery: Vermont, Australia, 2019.