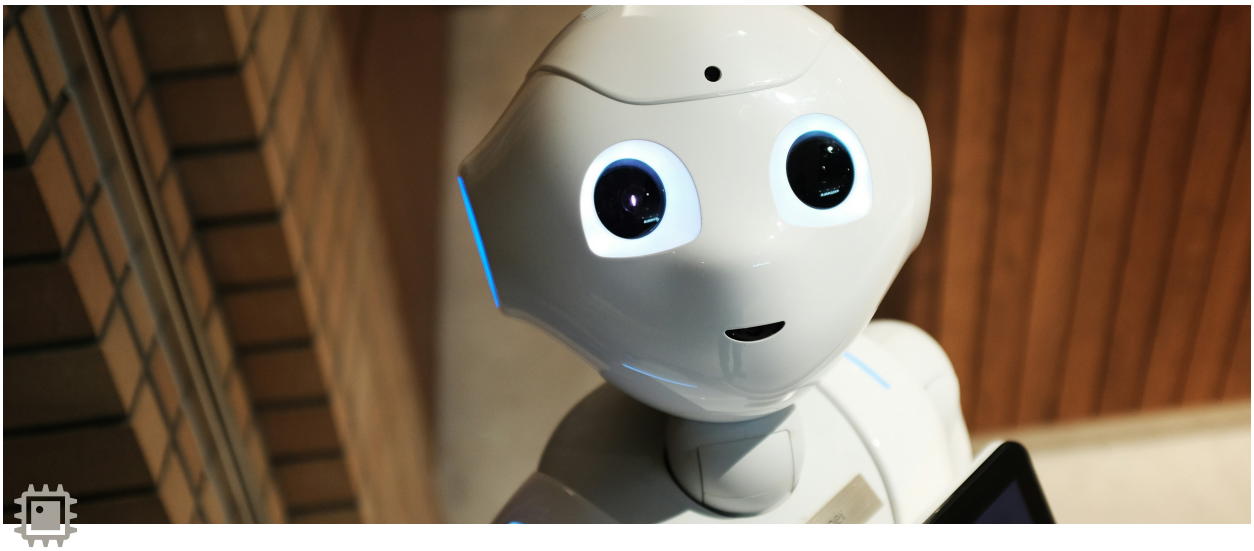


Reproduction blog post

This reproduction is done as part of the course on Deep Learning at TU Delft.
April 2024.
Authors: Sebastien van Tiggele (4705165), Luka Distelbrink (5274400),
Gleb Martjanovs (5295882), Daan Verburg (5605296)

Introduction

The reproducibility of scientific results is essential for the scientific community and science in general for several reasons. Experiments must be reproducible for conclusions to be considered valid. Without validation the reliability of conclusions becomes questionable (Yildiz et al, 2021).

In this project, the goal was to reproduce the work of Jaques *et al.* Their paper, titled *Physics-as-Inverse-Graphics: Unsupervised Physical Parameter Estimation from Video*, introduces a model capable of learning the physical parameters of a scene from video in an unsupervised manner, of which the differential equations of motion are known but the ground-truth appearance, position, or velocities of the object under investigation are not available. This novel approach aims to bridge the gap between unsupervised object discovery from video and learning the system's dynamics by learning unknown physical parameters and explicit trajectories.

The model created by Jaques *et al.* was originally written in Python version 3.3 using the Tensorflow version 1.12.0 library. The goal here was to recreate the model in Python version 3.11 using the Pytorch version 2.2.1 library. In order to do this, the original code was first updated to the current version of Tensorflow such that it could be executed in Python version 3.3. After this, subcomponents of the model code were rewritten in Pytorch version 2.2.1 one by one, ensuring that each modified component could be tested separately.

In this blog post, first, a section will be dedicated to the research by Jaques *et al.*, wherein the architecture of the model and the underlying theory are explained in-depth. Subsequently, the following section consists of our approach taken to reproduce the results of the paper. Thereafter, a section is dedicated to obtained results, followed by a conclusion section, and at last a task division section is presented to finalize this blog post.

Physics-as-Inverse-Graphics: Unsupervised Physical Parameter Estimation from Video

In order to reproduce the work of Jaques *et al.*, it is important to grasp its relevance, the structure of the model, and how it is applied. The relevance becomes apparent when looking at physical scene understanding methods described in literature preceding the work of Jaques *et al.* These methods either require object state supervision, meaning they rely on manually labeling object states, or they were not integrated with differentiable physics to learn interpretable system parameters and states. The model proposed by Jaques *et al.* addresses both is these shortcomings. It first learns the states of the objects (such as positions and velocities) from video data and then it feeds these states to a physics engine equipped with the differential equations governing the scene to learn the corresponding physical parameters.

The architecture of the model can be divided into 4 modules: an encoder, a velocity estimator, a differentiable physics engine, and a graphics decoder. A global view of the architecture is depicted in Figure 1 below.

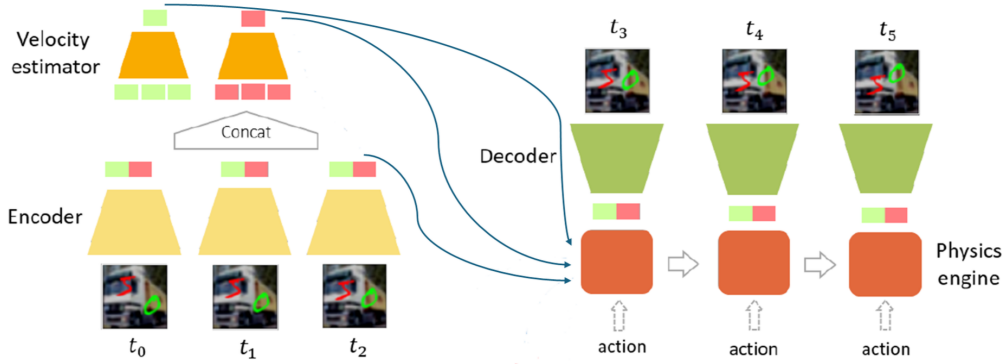


Figure 1: A schematic depiction of the architecture of the model by Jaques *et al.*

Encoder

The encoder net extracts the coordinates of each object, for each frame separately, using a localization approach consisting of two steps. First, the input frame I_t, corresponding to a certain timestamp t, is passed through a so-called U-Net producing N masks, where N is the predefined amount of objects in the scene. A U-Net is a type of convolutional neural network consisting of a contracting path (encoder), which gradually reduces the spatial dimensions of the input image while increasing the number of feature channels, and an expansive path (decoder) that upsamples feature maps from the contracting path and reconstructs segmentation masks with reduced spatial dimensions and feature channels, resulting in accurate pixel-level object classification (Ronneberger et al., 2015). A learnable background mask and the N masks from the U-Net are then stacked and passed through a softmax resulting in $N + 1$ masks, softly assigning each pixel to a mask. Each mask is then multiplied by the input image and a 2-layer localization network produces coordinate outputs for each masked input component. Thus, the input to the encoder net is a single frame I_t and the output is a vector $\mathbf{p}_t \in \mathbb{R}^{N \times D}$ corresponding to the D-dimensional coordinates of each of N objects. A schematic depiction of the encoder net is depicted in Figure 2 below.

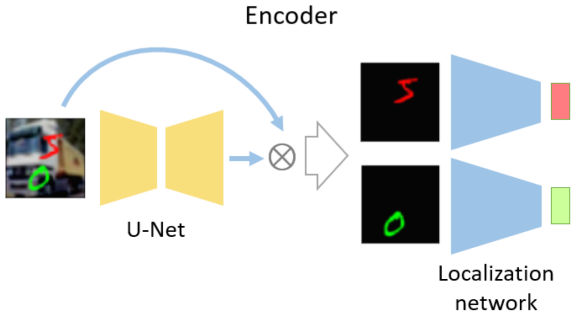


Figure 2: A schematic depiction of the encoder net is in the model by Jaques *et al.*

Velocity Estimator

As shown in Figure 1, the coordinates produced by the encoder net are passed through a velocity encoder consisting of a 3-hidden layer multilayer perceptron with 100 tanh-activated units. The velocity vector at the L-th input frame for each object $\mathbf{v}_L^n = f(\mathbf{p}_1^n, ..., \mathbf{p}_L^n)$ is estimated using the first L input frames.

Differential Physics Engine

After all initial position and velocity vectors have been determined by the preceding steps, the trajectory for every object is determined by a physics engine. The physics engine contains the physical parameters that need to be learned (e.g. mass, gravity, spring constant, etc.) and the differential equations describing the system. To calculate the trajectories of the objects, $\mathbf{p}_t$ and $\mathbf{v}_t$ are calculated from $\mathbf{p}_{t-1}$ and $\mathbf{v}_{t-1}$ using Euler integration:

$$\mathbf{p}_{t+\frac{i}{M}} = \mathbf{p}_{t+\frac{i-1}{M}} + \frac{\Delta t}{M} \cdot \mathbf{v}_{t+\frac{i}{M}}; \qquad \mathbf{v}_{t+\frac{i}{M}} = \mathbf{v}_{t+\frac{i-1}{M}} + \frac{\Delta t}{M} \cdot \mathbf{F}(\mathbf{p}_{t+\frac{i-1}{M}}, \mathbf{v}_{t+\frac{i-1}{M}}; \theta),$$

which is repeated for $i \in [1...M]$ times, where Δt is the integration step, $\mathbf{F}(.)$ is the force exerted on each object defined by the equations of motion, and θ are the parameters that need to be learned. The model is able to determine the physical parameters of the following scenarios: a gravitational system with 3 colored balls where the balls' mass
m and the gravity constant g need to be learned, two colored balls connected to a spring where the equilibrium distance l and the spring constant k need to be learned, and a pendulum where the gravity g and actuation coefficient a need to be learned.

Coordinate-Consistent Decoder

The decoder is the most crucial part of the model, as precisely this component enables unsupervised training of the encoder, velocity estimator, and physics engine. It generates a predicted image $\tilde{I}_t$ using either the positions coming from the encoder or the physics engine. The key concept enabling unsupervised learning is that the decoder is coordinate-consistent, meaning that the relationship between the object's location in pixel space and the position vector is fixed such that correct future predictions can be made. A schematic depiction of the Coordinate-Consistent Decoder is shown in Figure 3 below.

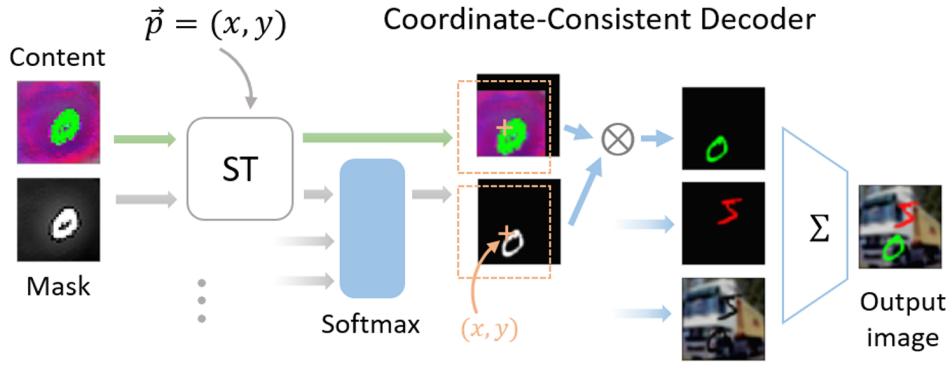


Figure 3: A schematic depiction of the Coordinate-Consistent Decoder used in the model by Jaques *et al.*

To ensure coordinate-consistency, spatial transformers (ST) are used in a particular way. In the original formulation of the ST, an affine transformation is applied to the output image to obtain the source image, meaning that the elements of the transformation matrix do not represent the translation, scale, or angle of the attention window. For a general affine transformation of a source image in 2D with translation (x, y), rotation θ and scale s, it modifies the coordinates as:

$$\begin{pmatrix} x_o \\ y_o \\ 1 \end{pmatrix} = \begin{pmatrix} s \cdot \cos\theta & s \cdot \sin\theta & x \\ -s \cdot \sin\theta & s \cdot \cos\theta & y \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_s \\ y_s \\ 1 \end{pmatrix}.$$

To obtain transformer parameters such that a decoder input $\mathbf{p}_t^n = [x, y]_t^n$ places the attention window at (x, y) in the image, the transformation matrix above is inverted, resulting in:

$$\begin{pmatrix} x_s \\ y_s \\ 1 \end{pmatrix} = \frac{1}{s} \begin{pmatrix} \cos\theta & -\sin\theta & -x\cos\theta + y\sin\theta \\ \sin\theta & \cos\theta & -x\sin\theta - y\cos\theta \\ 0 & 0 & s \end{pmatrix} \begin{pmatrix} x_o \\ y_o \\ 1 \end{pmatrix}.$$

A ST with parameters $\omega_{\mathbf{p}_t^n}$ given by this inverted matrix is used to obtain a decoder with coordinate-consistent outputs.

As shown in figure 3, each object consists of a learnable content $\mathbf{c}^n \in [0, 1]^{H \times H \times C}$, and a mask tensor $\mathbf{m}^n \in \mathbb{R}^{H \times H \times 1}$, where $n = 1...N$, H is the image height/width and C is the number of channels in the image which for an RGB image would be 3 (one for every color). Both $\mathbf{c}^n$ and $\mathbf{m}^n$ are transformed by the ST as $[\hat{\mathbf{c}}_t^n, \hat{\mathbf{m}}_t^n] = \mathrm{ST}([\mathbf{c}^n, \mathbf{m}^n], \omega_{\mathbf{p}_t^n})$. In addition, background content $\mathbf{c}^{bkg} \in [0, 1]^{H \times H \times C}$ and background mask $\mathbf{m}^{bkg} \in \mathbb{R}^{H \times H \times 1}$ are learned as well that are not passed through the ST. As depicted in figure 3, the output masks are obtained by combining the logit masks using a softmax across channels, $[\tilde{\mathbf{m}}_t^1, ..., \tilde{\mathbf{m}}_t^N, \tilde{\mathbf{m}}_t^{bkg}] = \mathrm{softmax}(\hat{\mathbf{m}}_t^1, ..., \hat{\mathbf{m}}_t^N, \hat{\mathbf{m}}^{bkg})$. Then, in the last step, the contents are multiplied by the output masks resulting in the final output image:

$$\tilde{I}_t = \tilde{\mathbf{m}}_t^{bkg} \odot \mathbf{c}^{bkg} + \sum_{n=1}^{N} \tilde{\mathbf{m}}_t^n \odot \hat{\mathbf{c}}_t^n.$$

Using both ST's and masks as described above enables simulation depth ordering which allows the model to capture occlusions between objects.

Training

For training, L input frames are used to predict the next T_{pred} frames. The total loss used during training is defined as:

$$\mathcal{L}_{total} = \mathcal{L}_{\text{pred}} + \alpha \mathcal{L}_{\text{rec}} = \sum_{t=L+1}^{L+T_{pred}} \mathcal{L}\left(\tilde{I}_t^{\text{pred}}, I_t\right) + \alpha \sum_{t=1}^{L+T_{pred}} \mathcal{L}\left(\tilde{I}_t^{\text{ae}}, I_t\right),$$

where $\tilde{I}_t^{\text{pred}}$ are the frames produced by the decoder via the physics engine, $\tilde{I}_t^{\text{ae}}$ are the frames produced by the decoder using the output of the encoder and α is a hyperparameter. The loss $\mathcal{L}$ used throughout is the mean-squared error loss.

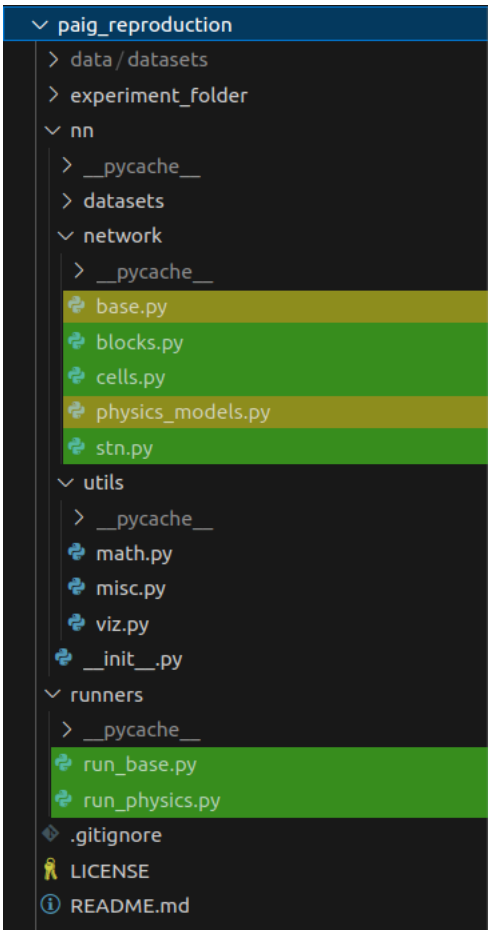
Our approach to reproduction

As mentioned in the introduction, the code of the original model by Jaques *et al.* was written in Python version 3.3, using the Tensorflow version 1.12.0 library. The plan was to recreate the model in Python version 3.11 using the Pytorch version 2.2.1 library since this is a framework that we are familiar with.

To achieve this, the original code was first updated to the current version of Tensorflow. This was necessary since Tensorflow 1.12.0 is not compatible with Python 3.11 and Pytorch 2.2.1 is not compatible with Python 3.3. Updating to the current Tensorflow version allowed us to work with Tensorflow and Pytorch at the same time. This enabled us to rewrite subcomponents in Pytorch separately while being able to compare intermediate results such as the shapes of the input and output tensors.

The subcomponents of the model were re-implemented in Pytorch according to the following order: encoder → velocity estimator → decoder → cells. The encoder, velocity estimator, and decoder are explained in the section about the original paper, and cells contains the functions used in the physics engine such as Euler integration and the equations of motion for each system.

Additionally, the base classes, containing the training and evaluation loops, were adjusted to better align with the workflow of the Pytorch library. The overview of modified files is presented on the right.



Project file tree. Fully reproduced files are marked in green and partially reproduced in yellow.

In order to still make use of the general utility functions of the original model, such as the data visualization code, the way the data is managed and stored is kept the same. This did lead to some redundant or slightly convoluted ways to process or reassign the data due to the differing ways Pytorch and Tensorflow work. Nevertheless, it allowed us to repurpose some of the original code's architecture.

Results

As mentioned above, we recreated the model using PyTorch 2.2.1. While we were able to recreate and train the model, we were not able to reproduce the results obtained in the original paper. The original authors trained their models for 500 epochs. We trained our PyTorch implementation of the model for the "spring-color" task for 200 and 500 epochs using the recommended hyperparameters from the original paper. The loss is shown in Figure 4 and Table 1 below. Here the "pred loss" refers to the loss when predicting future frames based on a few input frames, "recons loss" refers to the loss of feeding a frame into the encoder and decoder and comparing it to the original, "extrap loss" is the loss for the prediction of future frames for a longer timescale than encountered during training, and "train loss" is a weighted sum of prediction and reconstruction loss.

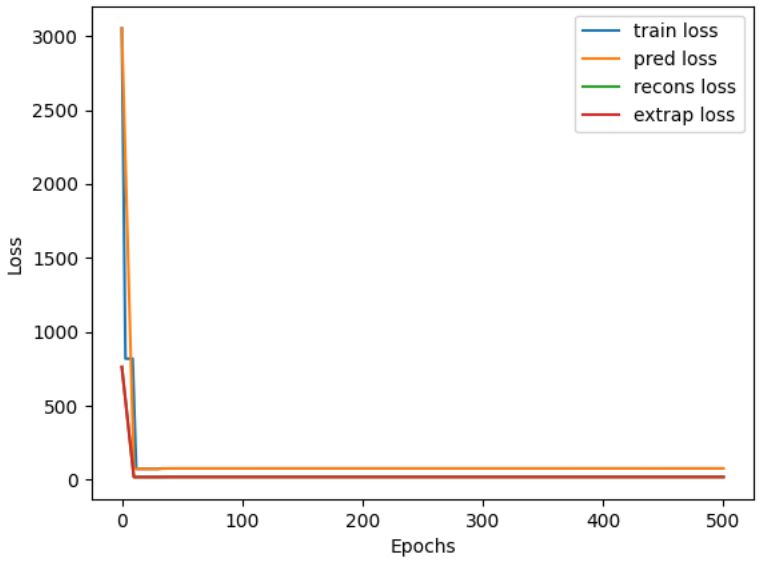


Figure 4: Loss during training and evaluation.

Loss \ Model	Jaques *et al.*	PyTorch 200 epochs	PyTorch 500 epochs
L-pred	1.39	71.9	76.8
L-rec	0.63	14.5	19.4
L-extrap	-	39.3	18.7

As Table 1 shows, the original paper shows a much lower loss than was finally obtained with the reimplementation. Figure 4 shows that the loss already plateaus long before the 500 epochs, after just 10 epochs there is not much improvement.

Figure 5 shows the graphical output sequence of the model trained for 200 epochs. The middle row shows the true sequence of the two balls connected by a spring, moving in time from left to right. The first four frames in the top row are the input to the model, after which the predicted sequence is shown towards the right. The bottom row shows the reconstructed sequence. This is a sequence that is fed into the encoder, and directly out of the decoder without any further physics involved. The predicted sequence does not show two balls, but rather fragmented dots of various colours. This seems to suggest that the encoder did not successfully learn masks to differentiate the two objects and background. This does not mean that the result is necessarily in the encoder, as a disruption downstream from the encoder could prevent it from training correctly. In addition to this the fact that the reconstruction looks significantly better than the prediction suggests that there is an issue in another part of the network. The reconstructed sequence shows a slightly more coherent green ball and a vague blue dot that seems to be absent in some frames.

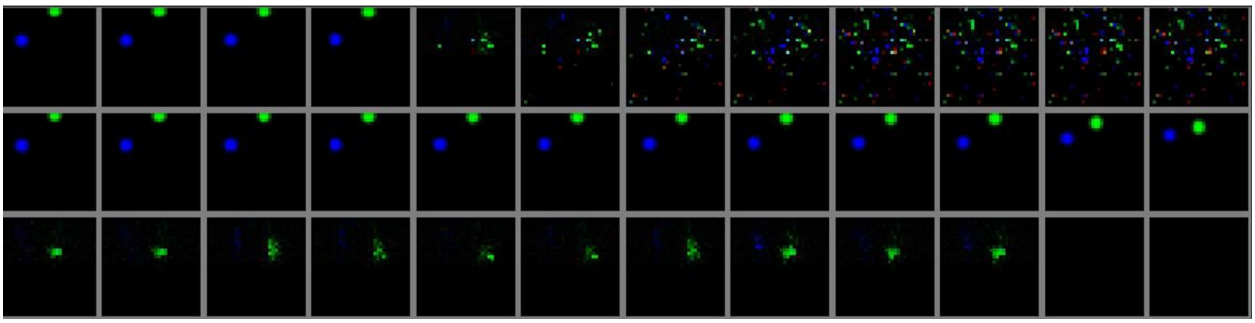


Figure 5: The results from the PyTorch model trained for 200 epochs on the “spring balls” task

Figure 6 shows the output sequence of the model trained for 500 epochs. Looking at the predictions in the top row, it looks as if it predicts nothing at all, but on close inspection, the blue dot can be seen in the middle of most (but not all) frames. The reconstructed sequence shows similar behaviour to the predicted sequence, where most but not all frames have a blue dot in the center.

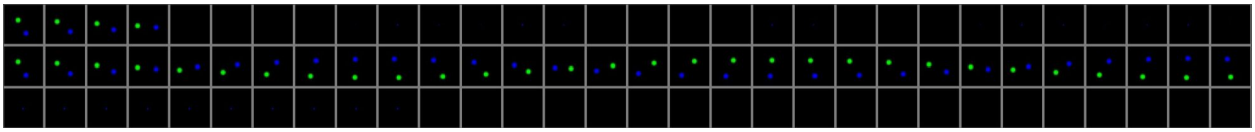


Figure 6: The results from the PyTorch model trained for 500 epochs on the “spring balls” task

While the loss did not change much between the 200 epoch and 500 epoch models, these trained models exhibit quite different behaviours. However, neither result is remotely close to predicting the sequences correctly. Since this prediction task involves predicting small moving balls on a black background, simply predicting an empty black background will yield a fairly low loss. Training on data with a more complicated background may alleviate this issue somewhat, but the model can still learn the static background, and this output will always have a relatively low loss compared to the true sequence, as the moving balls are rather small. However, it is unlikely that this is simply a local minima that can be escaped through hyperparameter tuning.

The fact that neither trained model was able to reconstruct frames that were fed into the encoder and decoder, indicates that there is a problem in this part of the model, most likely the issue is in the decoder, since the encoder is simply a convolutional neural network, while the decoder is more complex.

Tensorflow 2.0 results

To further investigate our findings, we decided to run the same “spring-color” task using the updated Tensorflow 2 model, which was directly upgraded from the GitHub of the authors of the original paper. This allowed us to directly use the code provided by the original authors and compare the results obtained with both our PyTorch implementation and the reported results in the paper. Again, this model was trained using the recommended hyperparameters from the original paper.

The results of the losses are shown in Table 2 in the same manner as in the previous section.

Loss \ Model	Jaques *et al.*	Tensorflow 200 epochs	Tensorflow 500 epochs
L-pred	1.39	14.3	13.8
L-rec	0.63	1.2	0.7
L-extrap	-	25.1	23.1

Interestingly enough, the losses seem to be much closer to the values as reported by Jaques *et al.*, but taking a look at the results in Figure 7 and Figure 8 shows that the predicted frames do not closely match the true sequence and fade out in later frames.

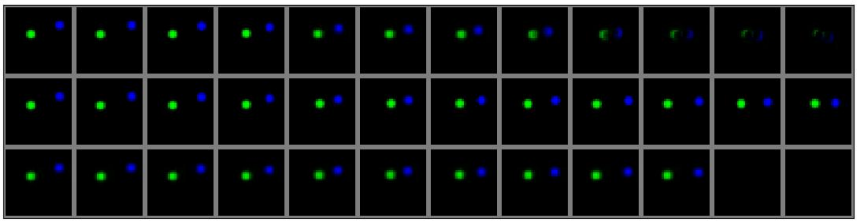


Figure 7: The results from the Tensorflow 2 model trained for 200 epochs on the “spring balls” task

The bottom row represents the reconstructed frames, which after both 200 and 500 epochs match the true sequence very well as Table 2 suggests. The top row shows the predicted frames, and the model trained for 500 epochs shows that the balls start to fade out later but rather quickly.

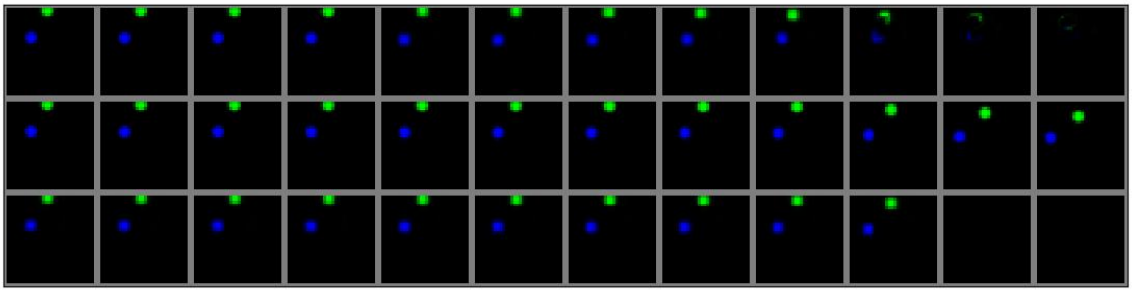


Figure 8: The results from the Tensorflow 2 model trained for 500 epochs on the “spring balls” task

It is quite curious to see the reconstructed frames behave so poorly compared to the performance as described in the paper, as shown in Figure 9. What causes these discrepancies is hard to detect, but does seem to indicate that something is not working correctly.

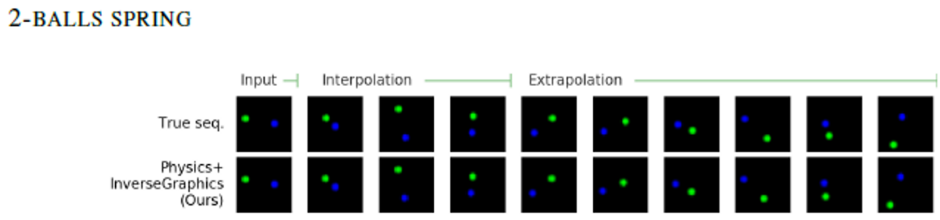


Figure 9: The proposed results in the appendix of Jaques *et al.* for the “spring ball” task

The balls are being reconstructed very similarly to the ground-truth sequence, and taking a look at the learned masks in Figure 10 shows that the U-Net does correctly classify the different objects from the input.

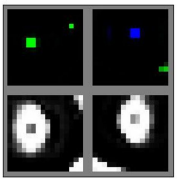


Figure 10: The objects and the corresponding learned masks of our Tensorflow model

The model not only puts out these predicted and reconstructed frames but also tries to learn the underlying constants of these physics models. In the case of the “spring balls” task, the paper shows the ground-truth values versus the learned value of the model. Table 3 compares these numbers to our reproduced results, where k is the spring constant and l is the equilibrium distance.

Parameters	(k, l)

Learned Value Jaques *et al.*	(4.26, 6.17)
Ground-Truth Value	(4.0, 6.0)
Learned Value (reproduced)	(0.15, 1.02)

The results from Table 3 show that the physical parameters are learned incorrectly, which could be the cause of the erroneously predicted frames.

Conclusion

In a rapidly evolving field of machine learning things age as quickly as appear. The aging of Python libraries caused significant challenges in the reproduction of the work done by Jaques *et al.* Reproduction required a deep understanding of the theoretical foundation but also involved challenges associated with the implementation of the theory.

The transition between the two Python frameworks was not as smooth as expected, Tensorflow and PyTorch proved to have distinct features and conventions that required significant restructuring of code at some points. It also caused some difficulties when testing the intermediate progress of the reproduction, since Tensorflow's symbolic tensors could not be interchanged with PyTorch tensor to verify functions' implementation, due to some settings required to run TF1 compatible code in TF2.

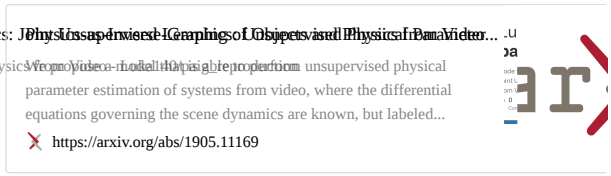
Of course, having source code to check whether inputs and outputs are the same, line by line is not necessarily required to reimplement a network, it is more of a luxury. However, due to the size and complexity of the program implementing it purely from the brief descriptions would be very difficult, and while there are a few comments in the original code, many operations are unexplained, and it can be hard to understand why they are performed.

Due to the complexity of the model, some deviation from the original results presented by Jaques *et al.* can be expected. However, as discussed in the results section, we were unable to reproduce results even remotely close to the results shown in the paper. To draw a more definitive conclusion on the reproducibility of this paper, a detailed analysis of the rewritten code would have to be conducted, to check for the correctness of the implementation, in addition to further testing and verification. This is therefore not a full reproduction.

A preliminary conclusion can be drawn, however, regarding how easily this paper can be reproduced, highlighting aspects that facilitate the reproduction process and areas that require improvement. Even though the authors provided their code it lacked comments and contained unfinished parts or unused code that made it more difficult to interpret how it was supposed to work. A more detailed flowchart of the code would be highly appreciated as well as a more detailed description of the individual components of the model instead of only a high-level description.

Useful links and references:

Link to the reproduction repository



Link to the original paper

Link to the original source code

Task division:

- Sebastien van Tiggele (4705165):
  - Rewrote for PyTorch
    - Parts of physicsmodels.py
    - Parts of Base.py
    - Runner files
  - Blog post writing
- Luka Distelbrink (5274400):

    - Updated original code to TF2
    - Rewrote for PyTorch:
        - The majority of physicsmodels.py
        - blocks.py
        - Alterations to base.py for a PyTorch-suitable training loop
    - Blog post writing
- Gleb Martjanovs (5295882):
    - Rewrote cells.py
    - Poster design
    - Blog post writing
- Daan Verburg (5605296):
    - Blog post writing
    - Implementation of the Spatial Transformer Network.