Chris Overtveld, Manupriya Singh, Xiaoluo Gong, Xiaduo Zhao

PHYSICS-AS-INVERSE-GRAPHICS:

UNSUPERVISED PHYSICAL PARAMETER ESTIMATION FROM VIDEO

Chris Overtveld	4429621
Manu Singh	6050425
Xiaduo Zhao	6033024
Xiaoluo Gong	5923476

Table of Contents

Contribution Overview

Section	Name
Local Setup	Everyone
Hyperparameters Check	Xiaoluo Gong
Updating Codebase	C. Overtveld
Experimenting New Data	Xiaduo Zhao
First results with default hyperparameters	Manu Singh

Introduction: PHYSICS-AS-INVERSE-GRAPHICS

Physical parameter estimation is a critical component in understanding and modeling the dynamics of various systems through observational data. Traditionally, this task has relied heavily on labeled data or manual annotations, which are often expensive or impractical to obtain, especially in dynamic environments captured through video feeds.

The original paper, "Physics-as-Inverse-Graphics: Unsupervised Physical Parameter Estimation from Video" , presents a pioneering approach to address the challenges in unsupervised learning of physical parameters directly from video data. This method integrates vision-as-inverse-graphics with differentiable physics engines to estimate the state and velocity of objects in video sequences without explicit supervision.The key idea is to use an encoder-decoder system where the encoder identifies object positions and velocities from video frames, and these are then fed into a differentiable physics engine.

The physics engine uses these inputs to learn the underlying physical parameters of the scene (like spring constants or gravitational parameters) while maintaining a vision-based feedback loop. This allows for long-term video prediction and model-based control, leveraging the system's ability to predict future states based on learned dynamics.The model begins by a encoder estimating the positions (p_t) and velocities (v_t) of objects in a video frame at time t. The differentiable physics engine then takes these estimates and computes the future states of these objects using learned physical parameters θ (like mass, spring constant, etc.). This computation can typically

be modeled using differential equations like those from Newton's laws of motion:

$$d^2p/dt^2 = F(p, v, \theta)$$

By reproducing this study, we aim to validate the claims made by the authors, particularly the model's effectiveness in long-term future frame prediction and vision-based model-predictive control.

The positions and velocities are updated using numerical integration techniques (like Euler integration) to predict future states:

$$P_t + \Delta t = P_t + \Delta t \cdot V_t$$

$$V_t + \Delta t = V_t + \Delta t \cdot d^2p/dt^2$$

Where Δt is the small time step.

The decoder then uses these predicted positions and velocities to generate the next video frame.This step reconstructs or predicts the next frame $\hat{I}_{t+1}$ based on the updated states.

During training, the model optimizes a loss function that typically measures the difference between the predicted frames and the actual next frames from the video, encouraging the model to accurately predict future states that best describe the physical dynamics observed in the videos.

$$L_{total} = L_{pred} + \alpha L_{rec}$$

The α hyperparameter scales the contribution of the reconstruction loss when calculating the total loss. The choice of α affects how much emphasis is put on the reconstruction aspect of the training relative to the prediction aspect. By iteratively updating the physical parameters θ and refining the predictions,

the model learns to simulate and predict complex physical interactions in a visually interpretable manner, effectively learning the physics of the scene directly from video data.

By reproducing this study, we aim to validate the claims made by the authors, particularly the model's effectiveness in long-term future frame prediction and vision-based model-predictive control.

First set of Training losses with running the code with default hyperparameters

Here, we will discuss the training progress when we ran the results with default hyperparameters. While tracking the progress, we had to check the log.txt file. The value of eval_recons_loss should be below 1.5 ,indicating that the encoder and decoder have correctly discovered the objects in the scene. The value of eval_pred_loss should be less than 3.0 and 30 (for balls and mnist data respectively), indicating that the velocity estimator and physical parameters have been learned correctly.

bouncing_balls

On running the default code we got final eval_pred_loss as 15.897555 and eval_recons_loss as 14.472928.

spring_color

The final eval_pred_loss is 12.574614 and eval_recons_loss=7.6714106. Here, losses are high in comparison to required limits.

spring_color_half

The final eval_pred_loss is 12.681991 and eval_recons_loss is 11.18976 which are higher than required.

3bp_color

The final eval_pred_loss is 25.79133 and eval_recons_loss is 25.811243 which are way higher than the required limits of 1.5 and 3.0.

minist_spring_color

This one was not able to converge for a long time and start giving nan values. The last values of losses before nan are eval_pred_loss=125.05992 and eval_recons_loss=124.84918. They are far from convergence.

Hence, it was required to do hyperparameter checks and tune them to converge the model to get most learned parameters.

Hyperparams check

According to the training instruction on GitHub of the model (https://github.com/seuqaj114/paig) , there are mainly those hyperparameters that influence the model training result:

Epochs (default 500): This hyperparameter determines the number of times the entire dataset is passed forward and backward through the neural network. Increasing the number of epochs can lead to better training results if the model has not yet converged, but it also increases the risk of overfitting and the computational time required for training.

batch_size(default 100) : The batch size dictates how many examples the model is exposed to before updating the internal model parameters. A larger batch size can lead to more stable and accurate gradient estimates but may require more memory and can sometimes lead to poorer generalization.

Autoencoder_loss (recommended 3.0): This is the weight of the reconstruction loss in the total loss function (the α mentioned above).

Increasing it places more emphasis on accurately reconstructing the input frames, while decreasing it puts more emphasis on the predictive aspect of the model. This can be tuned to balance the performance of the model on reconstruction versus prediction.

Base_lr (recommended 3e-4): The learning rate is one of the most critical hyperparameters and affects how much the model's weights are updated during training. A higher learning rate can cause the model to converge faster but can overshoot the optimal solution. A lower learning rate ensures more stable convergence but at the risk of getting stuck in local minima or taking too long to train.

In our training, we adjusted the **Epoch** and **Base_lr** to make the code more feasible for running and we checked the influence of those two hyperparameters.

As we are not able to adjust the code to run on GPU, we trained with CPU and the training can be very slow. We finally managed to train for 230 epochs. The reconstruction loss is converged to 0.69 and the prediction loss is converged to 12.66 (See figure 1.). The threshold mentioned by the author is below 1.5 for reconstruction loss to indicate that the encoder and decoder have correctly discovered the objects in the scene and the prediction loss and below 3.0 in case of ball simulation for prediction loss. Our reproduction for 230 epochs, has successfully discovered the objects but not making the sufficient prediction. Thus, the author’s recommendation for 500 to 1000 epochs is reasonable.

Since the training with the recommended initial learning rate of 3e-4 is very slow, we adjusted the learning rate to 1e-3 (See figure 2). The session with a

base learning rate of 1e-3 shows a more rapid decrease in training loss initially compared to the session with a learning rate of 3e-4. This is typical as a higher learning rate can allow for faster convergence initially. However, despite the rapid decrease, the training loss for the learning rate of 1e-3 stabilizes around a higher value compared to 3e-4. Also, reduction in training loss for the lower learning rate appears smoother, which indicates a more stable learning process, reducing the likelihood of diverging suboptimal local minima.

Figure 1. Training losses for different epochs of 2-balls spring data

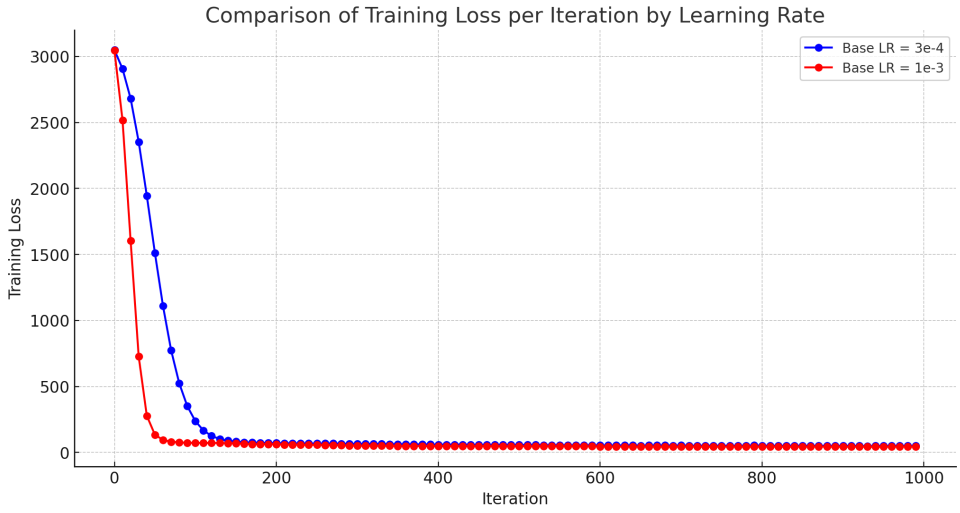


Figure 2. Training losses for 2-balls spring data under different base learning rates

- **New data**

  After training our model, to evaluate its generalization capability, we created a new dataset using data augmentation techniques. The dataset was constructed by applying random horizontal or vertical flips, as well as random translations in the x or y directions for each batch of test data from the original dataset. Here are the detailed steps and methods used:

  Step 1: Random Flipping

  For each frame within every video batch, a random choice is made between a horizontal or vertical flip:
  Horizontal Flip: The image is flipped from left to right.
  Vertical Flip: The image is flipped from top to bottom.

  Step 2: Random Translation

  Following this, each frame undergoes a random translation. The extent of the translation is determined by a random value ranging from 0 to 5 pixels, and can be either to the left or right (for the x-axis) or up or down (for the y-axis).

  Step 3: Zoom factor

  Added random zoom function. Each video frame is zoomed in or out according to a randomly selected zoom factor. Then reshape the output to the original shape by filling or clipping.

This metric measures the model's performance on data it has never seen before. The slight reduction in extrapolation loss for the new test set compared to the original test set suggests that data augmentation may have

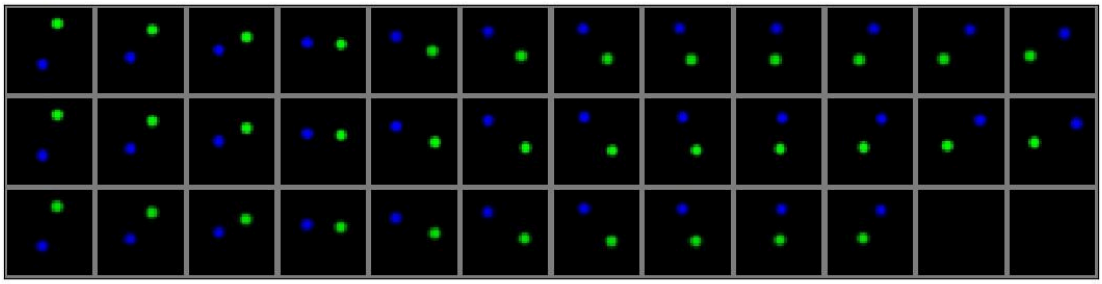
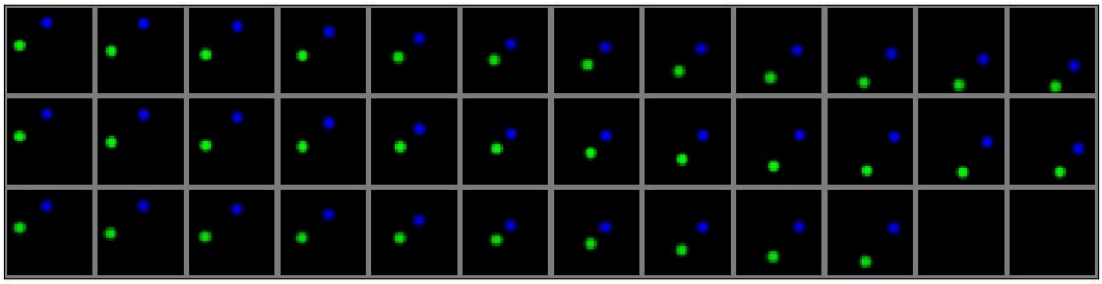
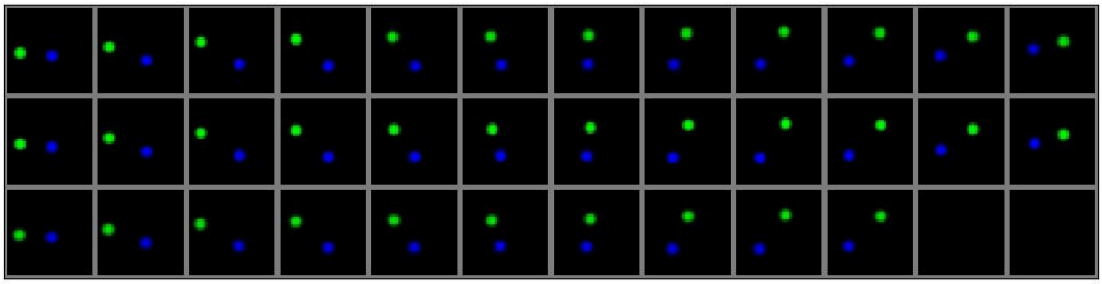
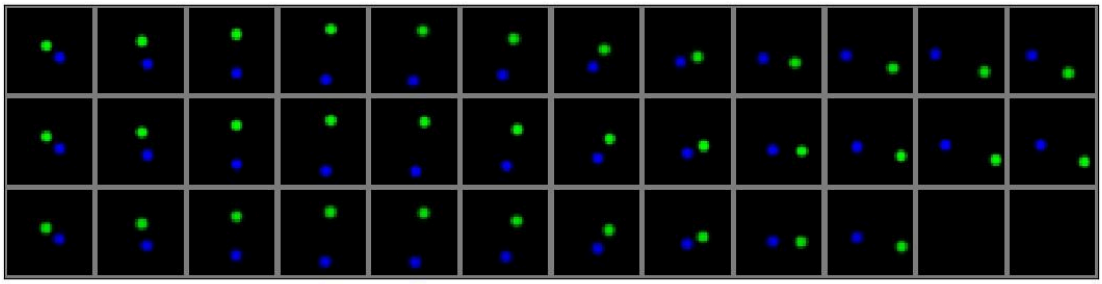
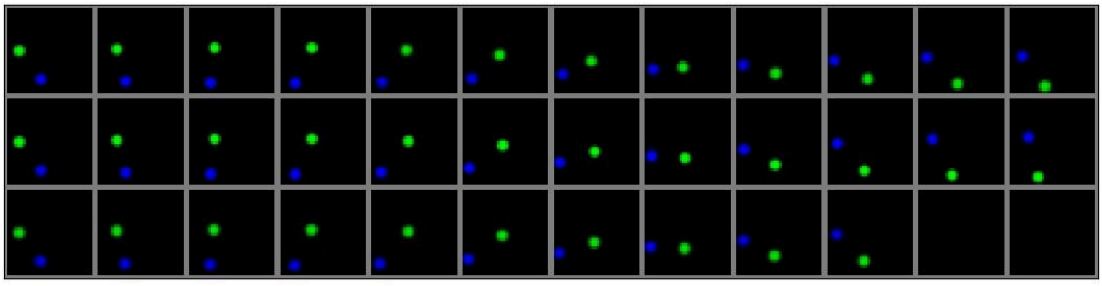
helped the model generalize and handle different data scenarios better, although the improvement is modest.

The data indicates that there was an increase in both prediction and reconstruction losses when comparing performance on a new, augmented test set against the original test set. Specifically, the prediction loss rose from 11.956585 to 15.404195, suggesting that the introduction of data augmentation may have added complexity or variability, presenting more challenges for the model in making accurate predictions. Additionally, the reconstruction loss, which measures the model's ability to reconstruct input data, also saw a significant increase, from 0.7104164 to 2.0544598. This increase implies that the augmented data may be more complex, and the model may not have adapted well to these new variations.

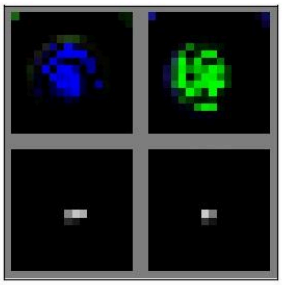
These results are consistent with expectations, considering that the model was not fully optimized during the training phase and did not converge on the best hyperparameters. Moreover, the new test set, having been subjected to various transformations, realistically increased in complexity, slightly elevating the difficulty in making accurate predictions.

Results

- **Figure 2**

- 





- **Figure 4**



-
- **Table 1**

Updating Codebase to TensorFlow 2.x

Since the code for the paper has been written in TensorFlow 1.12.0 in Python 3.6, there are some dependency issues when getting set up, especially when trying to accelerate processing with a gpu. TensorFlow 2 adds some important efficiencies in computation and automates a lot of the gpu handling; TensorFlow 1.x requires a separate tensorflow-gpu package, which has its own dependencies with graphics drivers and cuda, whereas this is handled automatically in 2.x and thus only one tensorflow package needs to be installed.

We were unable to ever get the original codebase working with GPU processing, we *think* that this is because the newer GPUs used on our local systems and in the cloud environments provided by Kaggle and Google Colab are not compatible with the combination of tensorflow<=1.15, cudnn<=7.5 and cuda<=11. For this reason, as well as those already mentioned, we looked at updating to tensorflow 2.x in Python 3.10, (initially 2.16 but decided in the end with 2.15 for for better compatibility with other packages). With this upgrade comes better readability and ease of use with the Keras API, and should lend the paper to better reproducibility and flexibility.

Summary of Changes

Eager Execution

TensorFlow 2.x runs by default in eager execution mode. This simplifies debugging since operations are executed immediately, this eager execution causes bugs with TensorFlow's compatibility modules (tf.compat) making a simpler upgrade from TensorFlow 1.x to compatibility code tougher.

Keras API Integration

TensorFlow 2.x makes the Keras API its stanford for model definition and training, which is a much higher-level API and thus reduces the amount of boiler-plate code leading to more readable and flexible models. For this paper, this meant redefining the model architecture (BaseNet, PhyscisNet, Unet, Shallow Unet) using Keras layers and utilizing the built-in training loops rather than the hard coded loops from the original paper. The custom ODE cells (inheriting from a BasicRNN Cell) also had to be rewritten since BasicRNNs are no longer compatible in Keras 3.

Optimization and Training Overhaul

Manual sessions and explicit graph execution was necessary in TensorFlow 1.x, so all of this had to be rewritten and restructured by leveraging the 'tf.keras.Model' class in TensorFlow 2.x. This class comes with built-in support for training and evaluation, which needed to be adjusted to account for the custom evaluation metrics (physical parameters) used in this paper. Optimizers had to be changed to use Keras optimizers, which was an easy enough step.

Simplified State Management and Checkpointing

Checkpointing becomes simpler in TensorFlow 2.x, with the use of 'tf.keras.Model.save' and '.load' methods. To adapt to the existing code and minimise changes however, the tf.

Results

While much of the architecture was rewritten, there were some parts which still need to be completed. Specifically only the `conv_encoder` and `conv_st_decoder` functions were translated to keras layers, and the `visualize_sequence` still needs to be implemented.

Unfortunately, there are a number of other changes necessary before the code can be run to reproduce the results seen in the paper. The DataIterator class functionality needs to be rethought as it currently feeds the data in batches, whereas this is handled by Keras internally. Similarly, now that the model process is not handled so granularly, the architecture may need more adjustments to correctly feed the date from one keras layer to the next while still maintaining the functionality of the original PhysicsNet methods, since now the encoder and decoder are pulled out of the PhysicsNet keras Model and defined as individual custom Layers. The `get_batch` and `compute_loss` functions also still need to be changed, or in the case of compute loss simply added as a loss function for Keras' training. The problem we encountered for `compute_loss` is getting the physical parameters to be added as evaluation losses for training since it seems Keras only takes a single evaluation loss.

The custom ODE cells on the other hand are much more readable, using the `call` method from the keras.layers.Layer class to provide the functionality of the cells. The Model class also clearly has benefits and with a larger team and more time to work solely on this upgrade it should certainly be possible, unfortunately we were not able to get there completely. Hopefully the current changes can act as a blueprint for further upgrading the code to a modern standard.

Conclusions

- PERFECT is not easily reproducible due to underlying bugs in the code that cost significant time and effort locating
- We layed out a method for multi-classification which failed, but believe could work with the right implementation
- We carried out a one-hot to integer labeling scheme in order to achieve a pseudo-multi-classification on the MFTC twitter corpus
- We reproduced the results of 3 single-sentence datasets to provide benchmarks for lesser computing power
- Differences in performance between Finetune and PERFECT could indicate that PERFECT is indeed learning a type of pseudo-multi-classification on the MFTC corpus

The low accuracy shown by the results can be explained by multiple factors:

- Hardware: the difference between the hardware used in the paper and what we had at our disposal surely had an impact, since we had to reduce not only the two parameters mentioned, but also the number of epochs from the original 6000 to 1000.

- The under-representation of the 2048 possible integer labels with a training set of only 149 present integer labels reduces the possibility of most generalization to texts with annotations not represented, even if they are subsets of the 149 or vice-versa.
- Ablation led to increased accuracy, which might be a result of a reduction in this overfitting to the 149 labels