PyLightcurve-torch: a transit modeling package for deep learning applications in PyTorch

Mario Morvan , Angelos Tsiaras , Nikolaos Nikolaou , and Ingo P. Waldmann
Department of Physics and Astronomy, University College London, Gower Street, London, WC1E 6BT, UK; mario.morvan.18@ucl.ac.uk
Received 2020 December 28; accepted 2021 February 16; published 2021 March 17

Abstract

We present a new open source python package, based on PyLightcurve and PyTorch Paszke et al., tailored for efficient computation and automatic differentiation of exoplanetary transits. The classes and functions implemented are fully vectorised, natively GPU-compatible and differentiable with respect to the stellar and planetary parameters. This makes PyLightcurve-torch suitable for traditional forward computation of transits, but also extends the range of possible applications with inference and optimization algorithms requiring access to the gradients of the physical model. This endeavour is aimed at fostering the use of deep learning in exoplanets research, motivated by an ever increasing amount of stellar light curves data and various incentives for the improvement of detection and characterization techniques.

Key words: Exoplanets – Transits – Photometry – Neural networks

1. Introduction

Exoplanets science discoveries have relied largely on our ability to extract precise information from stellar light curves. In the case of transiting exoplanets, this requires high precision photometric or spectroscopic measurements, and involves a transit model with one or several parameters to be determined. However, transit light curves often contain other sources of temporal variability caused by the instrument or the host star, which need to be accounted for in a preliminary or joint processing step. Consequently, forward modeling of transits needs to be considered as part of the data processing pipeline yielding the planetary parameters.

The complexity and growing amount of exoplanets light curves hint at the use of deep learning to help alleviating the issues encountered with traditional modeling techniques. Indeed, tremendous progress has been made recently in time series analysis owing to the recent development of deep learning, producing several successful applications and promising solutions for problems ranging from time series classification Fawaz et al. (2019) and forecasting Hewamalage et al. (2020) to denoising and anomaly detection Chalapathy & Chawla (1901). Among the various fields benefiting from such technical progress, astronomy has not been an exception. Several recent contributions to exoplanetary science (Zingales & Waldmann 2018; Passegger et al. 2020; Yip et al. 2020 for spectra, Shallue & Vanderburg 2018; Pearson et al. 2018;

Morvan et al. 2020 and Nikolaou et al. 2020 for photometric light curves) involve the use of deep learning methods.

The recent successes of deep learning can be traced back to the introduction of backpropagation as a technique which has allowed for the efficient optimization of neural networks Rumelhart et al. (1986). Today all major deep learning frameworks including TensorFlow Abadi et al. (2016) and Pytorch implement a way to automatically compute gradients of scalar outputs of functions, with respect to their inputs and parameters. To use and include a function in a data flow graph created by one of the deep frameworks above, the function must first be implemented using the buildings blocks of the framework: the functions and the numerical objects (multidimensional arrays, often called *tensors*) specific to the framework's language. So far, to the best of the authors' knowledge, none of the existing transit modeling codes has been designed to allow automatic differentiation and thereby joint end-to-end training with artificial neural networks. This is precisely the purpose of PyLightcurve-torch, which provides a user-friendly transit modeling tool to facilitate the generation, inference and optimization of transit models in a framework compatible with deep learning modules. We chose the language of PyTorch due to its user-friendliness, flexibility, efficiency and growing popularity. As the PyTorch syntax is very close to that of NumPy, it facilitates the easy conversion of NumPy codes to PyTorch, and reduces the learning curve for research communities who are used to scientific programming in NumPy.

PyLightcurve-torch[1] is adapted from PyLightcurve,[2] which is one of the most efficient open-source transit modeling packages available. PyLightcurve performs some numerical approximations rather than solving the fully analytical transit model to enable vectorisation of computations with NumPy Harris et al. (2020) and gain in efficiency, thus providing a good template for designing differentiable and scalable code. Four different limb-darkening laws are natively available in PyLightcurve, as well as several utilities for database access and fitting that we are not considering here. For more details about PyLightcurve's physics models, implementation and performance.

The remainder of this article discusses the implementation (Section 2), performance (Section 3) and applicability (Section 4) of PyLightcurve-torch.

2. Code Design

The numerical programming code was adapted from PyLightcurve transit modeling library. Indeed the main functions `exoplanet_orbit`, `transit_duration`, `transit_flux_drop`, `transit` and `eclipse` have been translated to PyTorch while preserving their names, structure, and parameters. However several major changes have been introduced along with the conversion to PyTorch. These have been summarized in the list below.

From NumPy to PyTorch NumPy arrays and operations have been converted respectively to PyTorch tensors and their corresponding operations. This means that the input parameters of corresponding main functions must now be of type `torch.tensor` and of shape broadcastable to `(batch_size,)` where `batch_size` is the number of instances of each transit parameter. While PyTorch tensors share many similarities with NumPy arrays, they further allow for GPU acceleration and automatic differentiation.

Vectorisation Main functions have been further vectorised to allow inputting 1D-arrays for each transit parameter in addition to scalars. Expressed in another way, this allows the user to provide batches of inputs to the main functions instead of individual sets of parameters. In the PyTorch version of PyLightcurve, this not only enables batch learning—i.e., optimization based on groups of observations considered jointly—but also fully leverages the GPU acceleration advantages on multi-dimensional tensors.

Flexibility of input shapes By allowing inputs of parameters of broadcastable shapes, the use of the main functions remains intuitively flexible, while saving memory and time in some specific cases. Indeed, when intermediate computations can be shared across a batch, such as the planetary positions vector for scalar orbital parameters, only a vector of batch dimension 1 is computed and used for later computations of transit flux drops, even if the latter are multidimensional.

TransitModule class A class named `TransitModule` has been implemented to facilitate the use of `transit` and `eclipse` functions, their optimization and embedding in deep learning pipelines. Indeed `TransitModule` first manages transit parameters and intermediate computations in an object-oriented fashion (see Listing 1 for a basic example). For convenience, the transit parameters passed as attributes of a module undergo checks, type, shape and device casting to make sure correct inputs are passed to the PyTorch functions performing the actual transit computations. Second, `TransitModule` inherits the `torch.nn.Module` along with its methods and parameters internal management. Furthermore, as the main parent class of all neural networks implemented in PyTorch, instances of `torch.nn.Module` can easily be combined together, facilitating the embedding or combination of our transit models with neural networks.

Algorithm 1. Listing 1. Basic use of Wrapper Class `TransitModule` Computing Transit and/or Eclipse Flux while Inheriting `torch.Module` Class

```
from pylightcurve_torch import TransitModule
# Model definition
tm=TransitModule            # time array-like object
  (time, **params)            and dict of params
tm.activate_-              # gradient activation
  gradient                    for "rp"
  (''rp'')
# Forward transit computation
flux=tm()                  # with module's defined
                             parameters
flux=tm                    # with substituted
  (e = 0., t0 = 3.4)          external parameters
# Loss and backward pass
err=loss                   # loss function
  (flux, **data)             evaluation
err.backward()             # backward propagation
                             of gradients
tm.rp.grad                 # gradient acces for
                             parameter "rp"
```

3. Performance

Several tests were conducted to assess the performance of PyLightcurve-torch and compare it to PyLightcurve. First of all, a sanity check was carried out to ensure both codes provide the same outputs when provided with the same inputs, up to to a precision level below $0.1 ppm$, on average, for default precision settings.

Two experiments were then performed with the aim of comparing the computational efficiency of both codes on CPU

[1] PyLightcurve-torch can be found on the following Github repository: https://github.com/ucl-exoplanets/pylightcurve-torch.
[2] PyLightcurve package can be found on the following repository: https://github.com/ucl-exoplanets/pylightcurve.
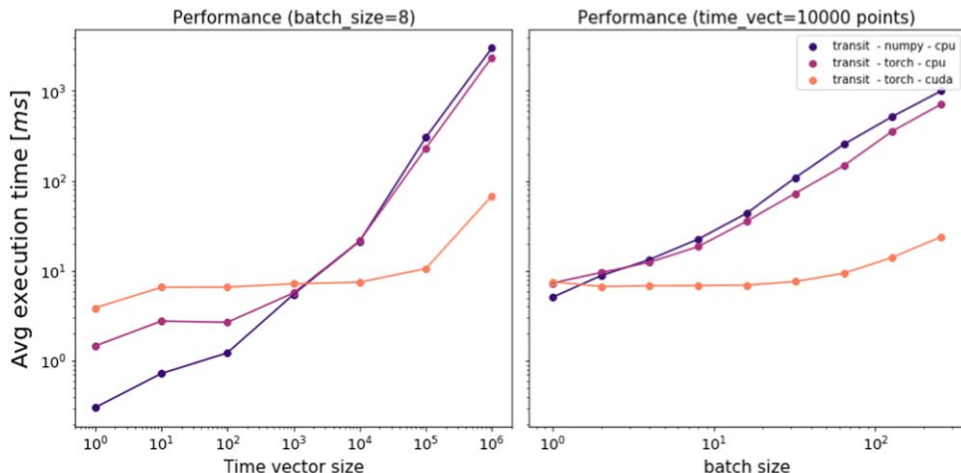
Figure 1. Execution times of PyLightcurve `transit` functions in NumPy and PyTorch, averaged over 50 executions. PyTorch functions have been run onto a CPU cluster (purple curve) and a GPU core (orange curve). Left: batch size fixed to 8 while increasing the length of the input time series. Right: the input time series size was fixed to 10000 points while increasing the batch size.

and GPU machines. In the first case the time array/tensor length was varied while keeping fixed the number of transit parameters, and in the second case the time array/tensor was fixed to 1000 while the number of parameters varied. The results are presented in Figure 1, suggesting very similar performances for the NumPy and the torch-cpu versions of PyLightcurve `transit` function. However, the GPU runs show a significant reduction in computation time, which is no longer increasing linearly with the input sizes but rather plateauing under $\sim 10ms$ for time tensor sizes smaller than 10^6 and batch_size smaller than ~256. Although these specific thresholds depend on the architecture used—which in our case consisted of 10 CPU cores, 70 GB memory and 1 Tesla-V100 GPU core—we expect GPUs to bring significant improvements in most configurations and use cases.

The ability to handle large data sets and maintain efficiency while computing a transit function or its gradient opens up new possibilities for processing exoplanet light curve data sets. Furthermore, automatic differentiability now allows the `transit` function or `TransitModule` to be included dynamically in deep learning pipelines and involved in their end-to-end optimization. The next section discusses the potential for new applications brought by PyLightcurve-torch.

4. Applications

This section, rather than being an exhaustive list of applications, aims to provide a quick overview of the types of novel applications afforded by the use of PyLightcurve-torch for designing both generative and discriminative models.

4.1. Generation

PyLightcurve-torch can be used in a static generative mode, to efficiently simulate primary and secondary transit light curves. In this case the gradients may not need to be activated (static mode: `requires_grad=False`), depending on the problem considered. Indeed, one can, for instance, create artificial data sets statically and use them to train models for various problems such as transit depth regression or event classification. While the simulated transits, created as PyTorch tensors, enable GPU acceleration, the conversion to NumPy arrays is still possible and very cheap computationally, simply by means of calling the `.numpy()` method available for `torch.Tensor` objects.

4.2. Gradient-based Optimisation

Having an efficient access to the transit model's output gradient with respect to the transit parameters enables gradient-based optimization without having to use approximate methods to compute gradients. Since the reverse-mode automatic differentiation (i.e., backpropagation) computes gradients efficiently, particularly for scalar outputs, we are led to consider the problem of scalar loss function minimization, an integral aspect of machine learning. Any loss function can be computed from the transit model output using differentiable functions to allow backward gradient-optimization. PyTorch implements a number of off-the-shelf optimisers which make use of the first or higher order derivatives of parameters. Moreover, several popular MCMC sampling algorithms such as Hamiltonian Monte Carlo and NUTS (ref) also require the availability of gradients, hence opening up the possibility for this class of MCMC algorithms to be used to derive posterior distributions for our transit models. Implementations of HMC
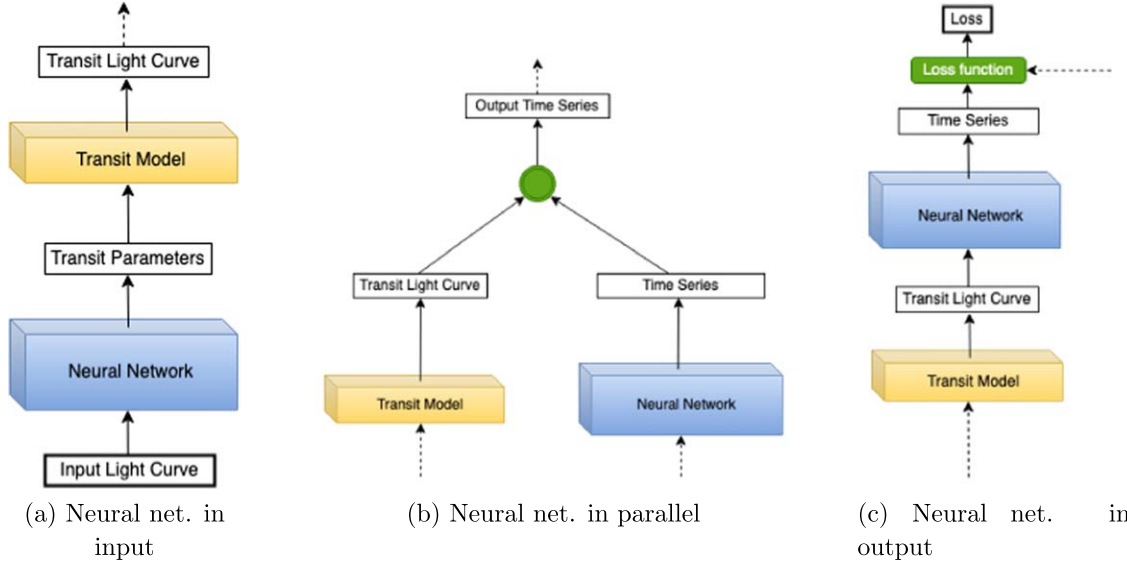
Figure 2. Schematic building blocks showing examples of combination between a neural network and differentiable transit model inputs (a) and/or outputs (b and c). Dotted arrows indicate possible additional inputs and outputs. The circular green node represents here a non parametric operation such as item-wise multiplication or addition. The loss function (in green) computes a scalar value from a time series in this case, optionally using some additional scalar or tensor values.

and NUTS are available in the probabilistic programming language Pyro Bingham et al. (2018), which is also based on Python and PyTorch. Besides exact bayesian inference with MCMC, it is worth noting that Pyro also provides a convenient framework for probabilistic, flexible and deep inference of parameters, and has been designed to be fully compatible with PyTorch tensors, functions and modules.

4.3. Combination with Neural Networks

The flexibility afforded by the autograd package and torch modules makes it particularly easy to connect the input and the output of our transit model with other differentiable functions and modules. All parameters can then be optimized in an end-to-end mode through the computational graph automatically built when defining and operating on the tensors. In practice, this means that any other differentiable module or function can:

1. provide the transit parameters as input of the transit model. A schematic example of this setup is shown in Figure 2(a).
2. be used in parallel with the transit model to provide other scalars, vectors or time series to be combined subsequently with the transit output. This setup is, for example, suitable for time series decomposition of transit light curves by means of generative models. A schematic example of this setup is shown in Figure 2(b).
3. be applied to the output time series of the transit model, transforming it to another time-series, a vector or a scalar. Note that to perform gradient-based optimization, a loss function outputting a scalar value will need to be at the

end of any functional flow. A schematic example of this setup is shown on Figure 2(c).

4.4. Experiment

Let us consider a simple example of regression problem where we use the transit module as an extra term in the objective loss function of a neural network. As such, this experiment falls into the case (a) discussed above, where a neural network output is used as input to a transit model. In the present experiment, the main task is to build a predictor for $y = R_P/R_s$, assuming the other transit parameters, θ, to be known. The data set is composed of 2000 light curves generated by PyLightcurve-torch with added Gaussian noise. Additional details about the experimental setup are provided in Table 1. All the lightcurves are univariate time-series with $T = 1000$ uniform timesteps and parameters fixed except for the target R_P/R_S and the inclination i which are sampled from uniform distributions. Note that in this experiment i is not included in the output targets, and that its variability is solely aimed at increasing the complexity of R_P/R_S prediction. We denote by Y the set of ground-truth targets and by $\mathcal{M}$ the transit model which generated the light curves.

Two identical neural network models m_1 and m_2 are trained on this regression problem using *different loss functions*. Indeed, m_1 is only trained with a mean-squared error loss between the target parameters y and their associated predicted

Table 1
Physical and Training Parameters used for the Regression Experiment

Parameters	Distribution	Values
Ratio of radii R_P/R_S	uniform	0.01–0.1
Inclination i (deg)	uniform	87.5–92.5
Limb-darkening method	constant	linear
Limb darkening coefficient	constant	0.2
Normalised semimajor axis a/R_S	constant	15
Argument of periastron (deg)	constant	50
Eccentricity e	constant	0
Gaussian noise standard deviation	constant	10^{-5}
Light curve lengths T (points)	constant	1000
Light curves time span (days)	constant	10
Mid transit time from start (days)	constant	5
Orbital period P (days)	constant	100
Training size	…	1200
Validation size	…	400
Test size	…	400
Optimiser	…	Adam
Initial Learning Rate	steps of 10^{-6}	10^{-6}–10^{-5}
Max epochs	constant	20,000

Note. Training, validation and test sizes indicate the number of light curves randomly generated to build the training, validation and test sets respectively. The distributions the parameters are sampled from are identical for the three data sets. Whenever the distribution is not constant, the values indicate the extremes of the corresponding sampling ranges.

value $\hat{y}$:

$$\mathcal{L}_1 = \frac{1}{|Y|} \sum_{y \in Y} (y - \hat{y})^2$$

whereas m_2 is also trained to reconstruct the transit flux by embedding the transit model as an additional term in the loss function:

$$\mathcal{L}_2 = \lambda \mathcal{L}_{\text{regression}} + (1 - \lambda) \mathcal{L}_{\text{reconstruction}}$$

where:

1. $\mathcal{L}_{\text{regression}}$ is identical to $\mathcal{L}_1$ but with predicted values $\hat{y}$ from m_2 instead.
2. The second term is a transit reconstruction loss, measured as the mean-squared error between the input light curve and the reconstructed transit light curve:

$$\mathcal{L}_{\text{reconstruction}} = \frac{1}{|Y|T} \sum_{y \in Y} \sum_{t \in 1..T} (\mathcal{M}(\hat{y}, \theta)_t - x_t)^2$$

3. λ is a scalar hyperparameter balancing the relative importance between the two loss terms.

The neural network architecture chosen consists of 4 convolutional blocks followed by 2 linear layers. Both models are trained using the Adam optimiser for 20,000 steps. Even though the networks and λ parameters would be suited for hyperparameter optimization, finding the optimal solution to this problem would go beyond the scope of this study, and here
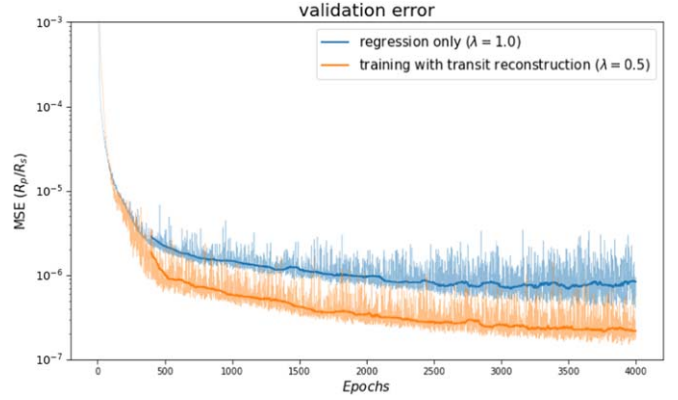

Figure 3. Validation mean-squared errors for R_P/R_S prediction with (orange curve) and without (blue curve) including a transit reconstruction term in the traning loss function. The thicker orange and blue curves merely show the moving median of the respective validation errors with a window size of 100 steps.

we simply present the results for a comparison of cases $\lambda = 0$ ($\mathcal{L}_2 = \mathcal{L}_{\text{regression}} = \mathcal{L}_1$) and $\lambda = 0.5$ (equal contributions from $\mathcal{L}_{\text{regression}}$ and $\mathcal{L}_{\text{reconstruction}}$).

The performance of both models is measured with the mean-squared-error of the predicted transit depths. The evolution of this metric evaluated on a validation subset of 400 lightcurves during learning is presented in Figure 3, which shows a significantly lower validation error for the model m_2 trained with the transit reconstruction term in the loss function. Indeed this model reaches m_1's final performance far before m_1 (in only about 2000 epochs) and continues to further improve its performance until the maximum number of epochs (20000) is reached. This indicates a clear advantage given to the model informed from the transit function and jointly trained with the proxy task of reconstructing the transit shape.

5. Summary and Discussion

In this article we outlined several possible uses of PyLightcurve-torch for generating and optimizing transits as well as embedding them in machine learning flows optimized end-to-end by gradient descent. While these examples are aimed at suggesting basic use-cases, the list is not exhaustive and various other uses might be designed in the future. Furthermore, the rich ecosystem of open-source libraries building up in PyTorch should provide ideas and help in the development of more complex applications, making use of existing support—e.g., for deep probabilistic programming, deep Gaussian processes, Bayesian hyperparameter optimization, gradient boosting and various implemented neural networks architectures.

By providing a differentiable and GPU-accelerated transit code, PyLightcurve-torch aims to facilitate and widen the use of deep learning in exoplanets research. It bridges the gap

between the precision of physical transit models and the scalability of neural networks, allowing for the efficient modeling of thousands of transit light curves now commonly available with exoplanets transit surveys such as Kepler Borucki et al. (2010) and TESS Ricker et al. (2015) from space or HATNet Bakos et al. (2004), SuperWASP Pollacco et al. (2006) and NGTS Wheatley et al. (2018) from the ground. Conversely, we hope that this code will also make exoplanets science more accessible to the machine learning community and more generally inspire the development of physics-based deep learning applications.

ORCID iDs

Mario Morvan https://orcid.org/0000-0001-8587-2112

Angelos Tsiaras https://orcid.org/0000-0003-3840-1793
Nikolaos Nikolaou https://orcid.org/0000-0001-8453-7574
Ingo P. Waldmann https://orcid.org/0000-0002-4205-5267

References

Abadi, M., Barham, P., Chen, J., et al. 2016, arXiv:1605.08695
Bakos, G., Noyes, R. W., Kovács, G., Stanek, K. Z., Sasselov, D. D., & Domsa, I. 2004, PASP, 116, 266
Bingham, E., Chen, J. P., Jankowiak, M., et al. 2018, arXiv:1810.09538 [cs, stat]
Borucki, W. J., Koch, D., Basri, G., et al. 2010, Sci, 327, 977
Chalapathy, R., & Chawla, S. 1901, arXiv:1901.03407
Fawaz, H. I., Forestier, G., Weber, J., Idoumghar, L., & Muller, P. A. 2019, Data Min. Knowl. Discov., 33, 917
Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Natur, 585, 357
Hewamalage, H., Bergmeir, C., & Bandara, K. 2021, Int. J. Forecast., 37, 388
Morvan, M., Nikolaou, N., Tsiaras, A., & Waldmann, I. P. 2020, AJ, 159, 109
Nikolaou, N., Waldmann, I. P., Tsiaras, A., & Morvan, M. 2020, arXiv:2010. 15996
Passegger, V. M., Bello-García, A., Ordieres-Meré, J., et al. 2020, A&A, 642, A22
Paszke, A., Gross, S., Massa, F., et al. 2019, arXiv:1912.01703
Pearson, K. A., Palafox, L., & Griffith, C. A. 2018, MNRAS, 474, 478
Pollacco, D. L., Skillen, I., Collier Cameron, A., et al. 2006, PASP, 118, 1407
Ricker, G. R., Winn, J. N., Vanderspek, R., et al. 2015, JATIS, 1, 014003
Rumelhart, D. E., Hinton, G. E., & Williams, R. J. 1986, Natur, 323, 533
Shallue, C. J., & Vanderburg, A. 2018, AJ, 155, 94
Wheatley, P. J., West, R. G., Goad, M. R., et al. 2018, MNRAS, 475, 4476
Yip, K. H., Changeat, Q., Nikolaou, N., et al. 2020, arXiv:2011.11284
Zingales, T., & Waldmann, I. P. 2018, AJ, 156, 268