Implementation of Lagrangian Neural Networks in Julia

Authors:
Ulrik Unneberg, Henrik Tidemann Kaarbø

1 Introduction and motivation

Langrangian mechanics is a flexible framework used by physicists to create effective and successful models for physical systems with finite degrees of freedom which respect specific continuous symmetries. The symmetries are manifested in the mathematical object referred to as the Lagrangian $\mathcal{L}$, from which the system's equation of motions can be derived, and the framework can be used to derive physical theories like Newtonian mechanics, general relativity and quantum field theory (*Lagrangian Mechanics (Wikipedia)* n.d.). Importantly, Noether's theorem shows that these symmetries $\mathcal{L}$ result in the system's dynamics having conserved quantities (Lancaster and Blundell 2014, ch. 10). For example, if $\mathcal{L}$ has no explicit time-dependence, the system's total energy remains a conserved quantity. However, deriving the correct Lagrangian for a system can be challenging and even small physical systems where subsystems interact, for example a set of interacting particles, can generate complex and chaotic dynamics while still conserving the quantities like energy required by the system's symmetries. This can make it difficult to derive equations of motion (EoM) from first principles, and even if they are well known, numerical models may be very expensive computationally as interactions are often functions of distance between different coordinates. This opens the possibility of approximating the unknown dynamics by using Neural Networks (NNs) for both learning the dynamics and approximating them to alleviate the computational costs.

However, it has been shown that normal neural networks struggle to capture system's conserved quantities when they encode the system's equations of motion directly (in the form of Neural ODEs) and are trained on the the observed system dynamics (Cranmer et al. 2020). Lagrangian Neural Networks (LNNs) are a recent addition to this family of neural networks that have been proposed to model and discover the dynamics of physical systems introduced by Cranmer, that are yet to be implemented in scientific machine learning packages for the Julia language. Instead of encoding the EoM directly, these neural networks encode the aforementioned Lagrangian $\mathcal{L}$ from which the equations of motions are easily derived. A related physics-based approach inspired by Hamiltonian mechanics, where the NN encodes the Hamiltonian of a system, introduced by Greydanus et al. 2019, has already been implemented in Julia and has been shown to generate quantity conserving dynamics when the training data obeys conservation laws. However, for the Hamiltonian framework to generate accurate dynamics and learn conserved quantities, the input coordinates of the Hamiltonian Neural Network must fulfill the strict requirement of canonical coordinates, i.e. all coordinates q_j and their momenta p_j need to fulfill

$$\{q_j, p_j\} = \sum_i \frac{\partial q_j}{\partial q_i}\frac{\partial p_j}{\partial p_i} - \frac{\partial p_j}{\partial q_i}\frac{\partial q_j}{\partial p_i} = \delta_{ij} \text{ and } \{q_i, q_j\} = \{p_i, p_j\} = 0, \text{ for } i \neq j,$$

a requirement not needed in the Lagrangian framework where dynamical coordinates (positions and their velocities) are sufficient. Subsequently, the dynamics and conservation laws of a system may be learned without explicit knowledge of the system's canonical coordinates and momenta, which can be challenging to both derive in theory and measure in practice, especially when some interactions may be unknown, yielding a substantial advantage for LNNs. Perhaps more importantly, measured data's coordinates are seldom expressed canonically nor includes the measurements needed to express them. For example the canonical momentum of a charged particle in an electromagnetic field is $m\mathbf{v}-e\mathbf{A}$ where $\mathbf{A}$ is the electromagnetic vector potential which is not directly measurable, but is related to the measurable magnetic field $\mathbf{B}(\mathbf{x}, t) = \nabla \times \mathbf{A}$ which together with the charge e may or may not be known. This offers up a large advantage to LNNs over HNNs where no such a priori information is needed about the training data. It is important to stress that conserved quantities due to continuous symmetries are ubiquitous in physics and does not end with energy. Other important conserved quantities due to such symmetries are for example the conservation of angular and linear momentum.

In this project, we implement LNNs in Julia to facilitate the use of LNNs in scientific research, engineering applications and potentially our own future research in physics.

2 Theory

To understand how LNNs work as defined by Cranmer et al, some background theory is needed. We begin by defining the *action* S as

$$S = \int_{t_0}^{t_1} L(\mathbf{q}_t, \dot{\mathbf{q}}_t, t)dt = (\text{Classically}) \int_{t_0}^{t_1} (T(\mathbf{q}_t, \dot{\mathbf{q}}_t) - V(\mathbf{q}_t, t))dt, \tag{1}$$

where the coordinates $\mathbf{q}_t$ and their time derivatives $\dot{\mathbf{q}}_t$ must span the possible states of the system uniquely. Let $x_t = x(t) \equiv (\mathbf{q}_t, \dot{\mathbf{q}}_t)$ be the state vector at time t. The action of a path taken by these generalized coordinates between the times t_0 and t_1 is observed in nature to be one for which the action is stationary, that is, the functional derivative $\delta S = 0$ along this path. Applying this variational principle and using calculus of variations gives the Euler-Lagrange equation(s) for the system

$$\frac{d}{dt}\nabla_{\dot{q}}\mathcal{L} = \nabla_q \mathcal{L}. \tag{2}$$

However, if we do not know the form of $\mathcal{L}(q, \dot{q})$, this form gives little clue of the dynamics of the system. By using the multivariate product rule, the equation can be rearranged into the equivalent form

$$(\nabla_{\dot{q}}\nabla_{\dot{q}}^{\top}\mathcal{L})\ddot{q} + (\nabla_q\nabla_{\dot{q}}^{\top}\mathcal{L})\dot{q} = \nabla_q\mathcal{L}, \tag{3}$$

which we can solve for $\dot{q}$, yielding

$$\ddot{q} = (\nabla_{\dot{q}}\nabla_{\dot{q}}^{\top}\mathcal{L})^{-1}[\nabla_q\mathcal{L} - (\nabla_q\nabla_{\dot{q}}^{\top}\mathcal{L})\dot{q}].$$

which can be used to express the right hand side of an ordinary differential equation defining the coordinates' dynamics

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ (\nabla_{\dot{q}}\nabla_{\dot{q}}^{T}\mathcal{L})^{-1}[\nabla_q\mathcal{L} - (\nabla_q\nabla_{\dot{q}}^{T}\mathcal{L})\dot{q}] \end{bmatrix} =: \mathcal{F}(q, \dot{q}). \tag{4}$$

At this point, we replace the Lagrangian $\mathcal{L}$ with a Neural Network $LNN(x,p)$ that approximates it. To train this neural network, the solution of the ODE as defined by (4), $x(t,p) = (q_t, \dot{q}_t)$, is compared to the data from the dynamics of a real system using a cost function $C(x(t,p), D)$ which is minimized with respect to the LNN's parameters p. To do so, the partial derivatives of the cost function with respect to p are needed, which means that the sensitivity of the ODE is needed. As further derivatives of the second order derivatives within $\mathcal{F}(q,\dot{q})$ are incompatible with current implementations of autodifferentiation in Julia, sensitivity analysis methods that make use of autodifferentiation are out of the reach of this project. Thus, the sensitivity $\frac{\partial x(t,p)}{\partial p_i}$ is (regrettably) approximated by the inefficient Ma et al. 2021 finite difference

$$\frac{\partial x(t,p)}{\partial p_i} \approx \frac{x(t, p + \Delta p_i) - x(t,p)}{\Delta p_i}. \tag{5}$$

More about this choice in Section 7.

3 Main contribution

The main contribution of this project is an implementation of a Lagrangian Neural Network framework in Julia, which from our understanding has not been done yet. With our framework, one should be able to model any dynamical system $x(t) = (q_t, \dot{q}_t)$, which can be modeled by a system of ordinary differential equations (ODE), using only an observed data set $D = (D_j)_{j\in[|D|]}$, where each training vector $D_j = (t_i^{(j)}, x^{(j)}(t_i))_{i\in[n_j]}$ is an observed evolution of the dynamics we want to model. If this data respects certain continuous symmetries, the model should be able to learn them and generate dynamics which conserves the correct quantities. By specifying a Neural Network architecture $LNN(x,p)$ using the Julia package Flux, one can with our framework learn the system's Lagrangian, and thus give physically sound predictions of its dynamical behaviour.

4 Implementation

The implementation in full with documentation can be found at https://github.com/henrtk/18.337-Project-LNN (active link), and is based on the Julia packages DifferentialEquations and Flux. The experiments conducted can be found as notebooks located in the "examples"-folder. The implementation is naive to both the selected model architecture of LNN and the choice of solver used to evolve and thus also the solver used to train the system, which can be chosen by the user. These are specified upon the initialization of a NeuralLagrangian object, where keyword-arguments are later given to DifferentialEquations.jl's solve()-function. The NeuralLagrangian contains a LagrangianNN struct which when called with an input state x returns $\mathcal{F}(x)$ as defined by (4). When the NeuralLagrangian object is called with an initial condition, a chosen ODE-solver and the LNN's parameters, the NeuralLagrangian object sets up the ODE-problem given by $\mathcal{F}$ and solves it with the given parameters and the previously set keyword-arguments. Discussion on the default options is left to section 7.

4.1 Implementation details

For the implementation to remain naive to the chosen architecture for the LNN, the gradients and hessians in eq. (4) are calculated using the auto-differentiation packages packages Zygote and ForwardDiff. As the Lagrangian should have a single output and can have an arbitrary number of coordinate inputs, the use of Zygote's reverse mode differentiation is appropriate. For the second order derivatives with respect to the input, forward-over-reverse mode is used, as the dimension of the gradient is equal to that of the input. In such cases, forward mode is generally more computationally efficient as we've learned in this course. However, it should be noted that for the purposes of the experiments conducted in this project, an implementation using forward mode *only* is faster as the input dimension in our case is small.

The ODE defined by $\mathcal{F}$ in (4) also includes the inverse of the Hessian of the LNN with respect to the input velocities $\dot{q}$. Because there is no guarantee that the Hessian is non-singular, we, like Cranmer, opted to use the Moore-Penrose pseudoinverse in its place to guarantee a solution is found. In the case that the Hessian is non-singular, its pseudoinverse is equal to its inverse. Mathematically, the singularity of the Hessian means that the Lagrangian is at a saddle-point. This means that there are multiple paths that can be taken out of this point where the action defined in (1) is stationary, ie. there is no unique trajectory out of this point when this Hessian is singular, and the dynamics are indeterminable from the Euler-Lagrange equations in eq. (2) alone. One may argue that such a situation is unphysical. Nevertheless, the uniqueness of the pseudoinverse effectively breaks the indeterminacy and absolves us of this problem, and prevents the code from crashing.

In the implementation of the training, we also benefited from the power of easy parallelization in Julia, by training over the batches using parallel threads. This enhanced the training efficiently significantly.

4.2 Description of the training step

The main component of the implementation of the LNN is the training step, so let's begin by addressing this point. Let $\mathcal{L}_p(x) := LNN(x,p)$ denote the neural network given the parameter vector p.

Given a data vector $D_j = (t_i^{(j)}, x_d^{(j)}(t_i))_{i\in[n_j]}$, in this section only denoted $(t_i, x_d(t_i))_{i\in[n]}$ for conciseness, we define the cost function as

$$C(x_t, x_p; p) = \mathrm{MSE}(x_d(t_i) - x_p(t_i))_{i\in[n]},$$

where $x_p(t)$ denotes predicted value at time t given parameter p. More precicely

$$x_p(s) = \mathrm{ForwardSolve}(\mathcal{F}_p, x_0)\Big|_{t=s},$$

where $\mathrm{ForwardSolve}(\mathcal{F}_p, x_0)$ indicate the numerical solver which solves the initial value problem

$$\dot{x} = \mathcal{F}_p, \qquad x(t_0) = x_0.$$

Our goal is then to find the sensitivity

$$\frac{\partial}{\partial p} C,$$

in order to backpropagate using some form of gradient descent. The cycle of each training step is as follows.

1. **Forward pass**

   Given a parameter vector p, which is all the parameters of our Lagrangian Nerual Network vectorized into one vector, we numerically integrate a solution to the dynamical system (2), replacing $\mathcal{L}(x)$ with $LNN(x, p)$. The output of this step is a DifferentialEquations.solve object, $x_p(t) = \text{ForwardSolve}(\mathcal{F}_p, x_0)$. In principle, any ODE solver can be used to find the forward solution. In practice, however, the efficiency of the training step is largely determined by which solver.

2. **Find sensitivity**

   Together with a data vector D_j, we can evaluate the sensitivity $\frac{\partial}{\partial p} C(x_t, x_p; p)$, using a gradient method.

3. **Update parameter**

   After evaluating the sensitivity, we update our parameter vector p using a gradient descent algorithm. We used the ADAM-optimizer with a step-size 0.001.

4. **Evaluate result**

   With the updated parameter, we can either continue training directly (going to step 1), or evaluate the result either by visual inspection by performing comparison plots with the given data, or evaluating the loss.

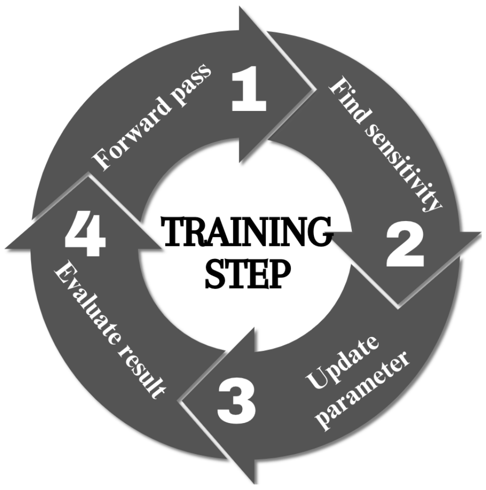


Figure 1: The training step cycle.

5 Experimenting on the double pendulum

As our example of demonstration, we have chosen to study the double pendulum - a simple yet chaotic system, which invites for challenges when modeled by an ordinary neural network. Furthermore, the system preserves energy. Energy conservation is one typical symmetry for which LNNs have shown to outperform Neural ODEs (NODEs) (Cranmer et al. 2020).

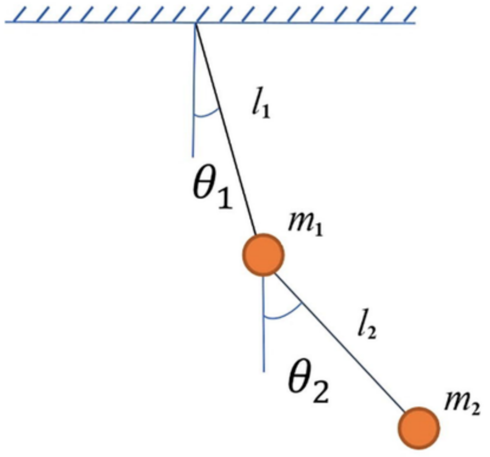


Figure 2: Illustration of the double pendulum.

The ODE which governs this system is given by

$$\frac{d}{dt}\begin{pmatrix}\theta_1\\ \theta_2\\ \omega_1\\ \omega_1\end{pmatrix} = \begin{pmatrix}\omega_1\\ \omega_2\\ g_1(\theta_1,\theta_2,\omega_1,\omega_2)\\ g_2(\theta_1,\theta_2,\omega_1,\omega_2)\end{pmatrix}$$

where

$$\alpha_1(\theta_1,\theta_2) := \frac{l_2}{l_1}\left(\frac{m_2}{m_1+m_2}\right)\cos(\theta_1-\theta_2) \tag{6}$$

$$\alpha_2(\theta_1,\theta_2) := \frac{l_1}{l_2}\cos(\theta_1-\theta_2) \tag{7}$$

$$f_1(\theta_1,\theta_2,\dot\theta_1,\dot\theta_2) := -\frac{l_2}{l_1}\left(\frac{m_2}{m_1+m_2}\right)\dot\theta_2^2\sin(\theta_1-\theta_2) - \frac{g}{l_1}\sin\theta_1 \tag{8}$$

$$f_2(\theta_1,\theta_2,\dot\theta_1,\dot\theta_2) := \frac{l_1}{l_2}\dot\theta_1^2\sin(\theta_1-\theta_2) - \frac{g}{l_2}\sin\theta_2 \tag{9}$$

$$g_1 := \frac{f_1-\alpha_1 f_2}{1-\alpha_1\alpha_2} \qquad g_2 := \frac{-\alpha_2 f_1 + f_2}{1-\alpha_1\alpha_2}, \tag{10}$$

with a true Lagrangian

$$\begin{aligned}\mathcal{L}(x) =& \frac{1}{2}(m_1+m_2)l_1^2\dot\theta_1^2 + \frac{1}{2}m_2 l_2^2\dot\theta_2^2 + m_2 l_1 l_2\dot\theta_1\dot\theta_2\cos(\theta_1-\theta_2)\\ &+ (m_1+m_2)g l_1\cos\theta_1 + m_2 g l_2\cos\theta_2.\end{aligned} \tag{11}$$

5.1 Experiments

To model the Lagrangian, we follow the architecture provided by Cranmer et al, in his implementation on Google Collab (*A self-contained tutorial for LNNs* n.d.). The network consists of two hidden layers with Softplus as activation functions. As our implementation rely on finite differences to find the sensitivity, this puts a severe restriction on the size of our neural network. Further analysis of complexity growth as function of parameters is found in Section 6.3. By testing with different number of neurons in each layer, we concluded that 10 neurons in each layer gave the most stable results, while still giving efficient training.

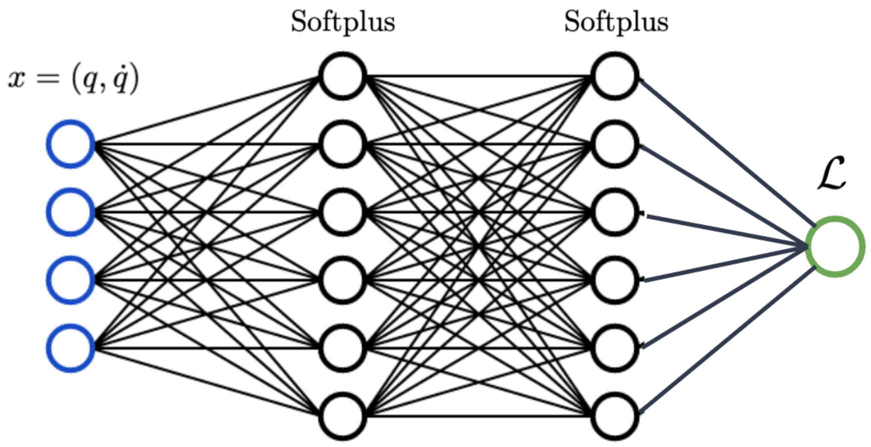


Figure 3: Architecture of LNN

5.1.1 Training and prediction

First and foremost, we want to test correctness of the scheme, by training the mentioned LNN to the point where it accurately mimics the analytical solution on the training interval. Once training is completed, we are interested in evaluating LNN's performance on a longer interval, i.e. predicting the future dynamics.

While a network as shown in Figure 3 with 10 neurons in each layer performs quite well, it lacks the flexibility to fully fit the true Lagrangian (9), which is very non-linear. The computational expense of the finite difference gradient makes it practically impossible to train a network much bigger than the one chosen to greater precision. While this is a big challenge, we are still interested in analyzing the power of the LNN at its best. Therefore, we did an *augmented training scheme* on a network with architecture as in Figure 3, but with 128 neurons in each layer. With a well-functioning AD method, a network of this size could easily be trained by the conventional training cycle as described in 4.2. The computational expense of training such a LNN with our own computers using finite differences, however, is very demanding, and as such seems in-achievable as of now. Therefore, we found applying the augmented training scheme, which will be described in the next paragraph, very useful.

The principle of the augmented training scheme is to start the training with an informed parameter vector p_0, which is not randomly given from Flux' destructure. In order to obtain this p_0, we trained the network on data points generated from the analytical Lagrangian $\mathcal{L}(x)$. In this way, we could drastically decrease the number of ordinary training steps on our LNN to achieve reasonable results. This workaround enabled us to perform realistic analysis of the performance of the LNN, even while lacking an effective gradient method. Therefore, it's important to emphasize that the large LNN is not solely trained on the data from the dynamics, but also on values from the Lagrangian directly. However, it's worth mentioning that the last training steps are indeed from the data from the dynamics, which significantly enhanced the overall performance of the network, so we believe the analysis henceforth is still reliable and insightful. While the scheme is not the optimal way of evaluating the performance of the LNN, it proved as a useful workaround tool to prove the concept of LNN, despite the lack of automatic differentiation. It is worth noting that evaluating the gradient of the cost function for the LNN with the given size ($\simeq 6000$ neurons) was on the order of 10s of minutes on our computers.

5.1.2 Energy conservation

As mentioned, one of the main benefits of LNNs are its symmetry-preserving properties. One typical example of such symmetry, is energy conservation, which is found in many large scale physical systems like planetary motion, as well as in many nano-scale systems. The double-pendulum is an example of an energy conserving system, such as moving particles in conservative fields.

NODEs have shown a tendency to struggle to capture symmetries such as energy conservation. Therefore, we are going to compare the energy conserving ability of LNNs and NODEs.

5.1.3 Computational performance

Finally, it's in order to evaluate the computational performance of LNN and NODE. To do this, we use Julia's BenchmarkTools package.

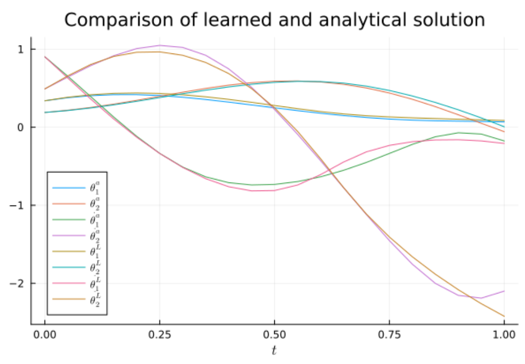
In particular, we will evaluate the time taken to find the sensitivity $\partial_p C$, which is the most time consuming part of each training step, and analyze its computational cost as a function of the number of parameters, to illustrate the complexity growth. We will do the same for NODEs, to have a benchmark for comparison. For the most robust results against possibly stiff $\mathcal{F}$, the solver used for benchmarking the LNN was the auto-switching AutoTsit5() which switches to Rosenbrock23() upon detection of stiffness.

6 Results

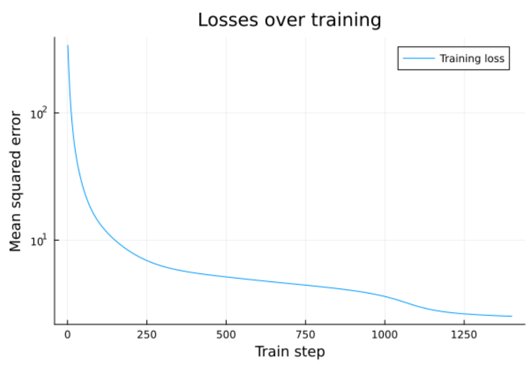
6.1 Training and prediction

After training our small LNN with $|D| = 1400$, with 10 different initial conditions, we got a result which fit the true dynamics quire well (Figure 4). By the look of the trained result and the evolution of the loss, the implementation seems to work correctly.

To evaluate the performance of the scheme, we would like to evaluate the LNN's ability to predict. To fully compare LNN and NODE, we found it necessary to train on big neural network, which can fully capture the Lagrangian of our system. Therefore, as mentioned in Section 5.1.1, we use a large neural network with an architecture as shown in Figure 3, and with 128 neurons in each hidden layers. This architecture will be used throughout this section. The training, together with the augmented training scheme, was performed on the interval $t \in [0, 1]$ for different initial coditions x_0. Thus, integrating beyond $t = 1$ and comparing it with the analytical solution will give us a metric of predicting ability. As



(a) Forward passed solution after training.



(b) Loss as function of training step.

Figure 4: Training of the small Lagrangian Neural Network.

LNN is an alternative to the NODE in these type of problems, it's natural to benchmark against a NODE solution. Therefore, we implemented a NODE as well, trained it on similar data, and did a comparison of the two on the training interval, as shown in Figure 5. They both appear well-trained.

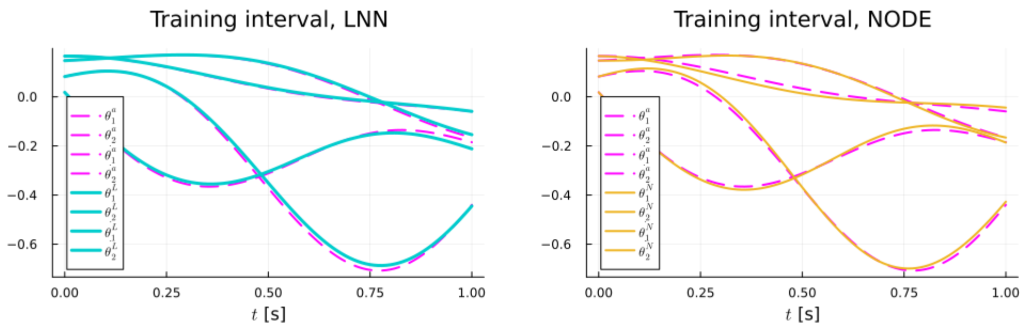


Figure 5: Comparison of LNN and NODE on training interval $t \in [0, 1]$. Analytical solution in dashed line.

To evaluate their performance on predicting the dynamics, we integrate to $t = 30$, which is well out of their training domain. The results are shown in Figure 6. As we can see, while the LNN doesn't follow the analytical behaviour exactly, it captures the shape of the dynamics much better than the NODE. The NODE, on the other hand, seem to be in a positive feedback loop, with increasing magnitude in both position and velocity. These results agree with the results found by Cranmer et al.

6.2 Energy conservation

The energy as function of time are plotted for all three solutions in Figure 7. As we can see, the LNN has some oscilation in energy, but has a steady average around the correct energy. The NODE's energy appears unstable, and seems to be blowing up for large t.

6.3 Computational performance

One final important metric to analyze is the computational performance of the LNN framework. As NODEs are the natural option to LNNs when given a data set to model without the dynamical equations, we found it apt to use its performance as a benchmark for the

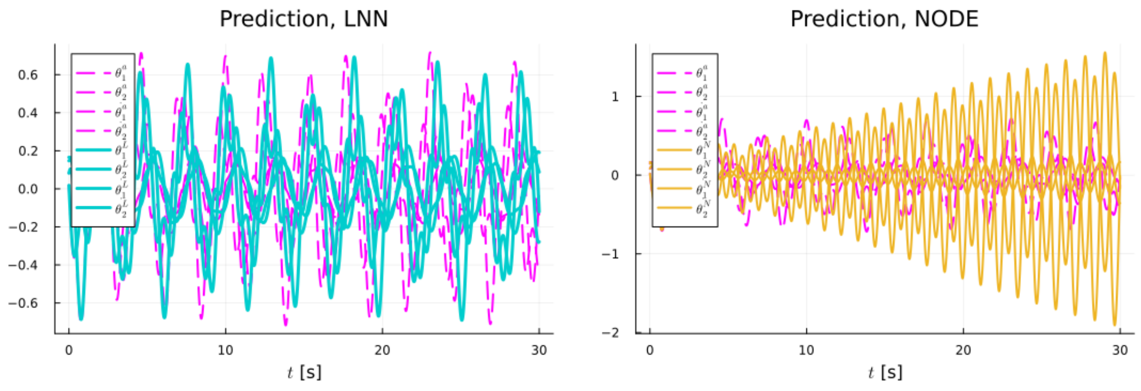


Figure 6: Comparison of LNN and NODE predicting the dynamics for $t = [0, 30]$. Analytical solution in dashed line.

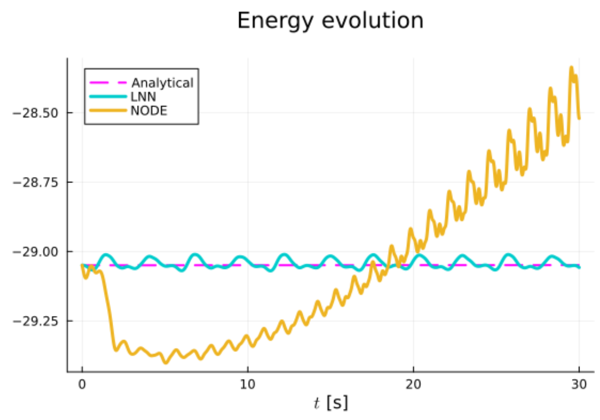


Figure 7: Comparison of LNN and NODE predicting the dynamics for $t = [0, 30]$. Analytical solution in dashed line. Though the NODE gives close predictions within the training interval, the energy is clearly not conserved as the NODE's dynamics slip into higher energy states over time.

performance of our implementation, especially given that it is possible to use adjoint sensitivity analysis to calculate its gradients which should give $\mathcal{O}(n + \text{length}(p))$ performance (Ma et al. 2021), where n is the number of coordinates in the ODE. As we can see in Figure 8, the computational cost grows linearly with the number of parameters with a growth of approximately 80 ms per added parameter. The growth should be $\mathcal{O}(n \cdot \text{length}(p))$ (Ma et al. 2021). This is tremendously expensive when compared to the NODE, seen in Figure 9, which granted has a much lower RHS evaluation cost. For NODEs using the SciMLSensitivity.jl package's automatic sensistivity analysis with InterpolatingAdjoint(), the growth in computationals cost per gradient was only 0.3 ms, two orders of magnitude less. This is despite the fact that benchmarking gives the evaluation of $\mathcal{F}$ using an LNN with 99 parameters was found to be around 35 μs, while a NODE with identical parameter size was found to cost 12.5μs.

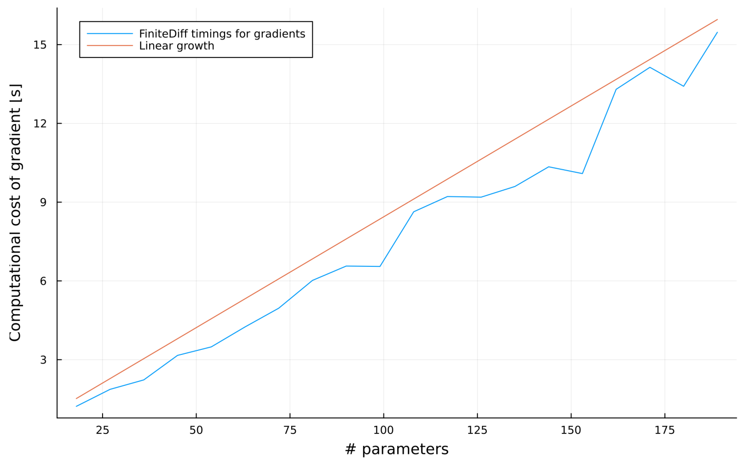


Figure 8: The computational cost for finding the gradient of the cost-function for the LNN as a function of the number of its parameters. The scale being given in seconds underlines the inefficiency of finding gradients in this way.

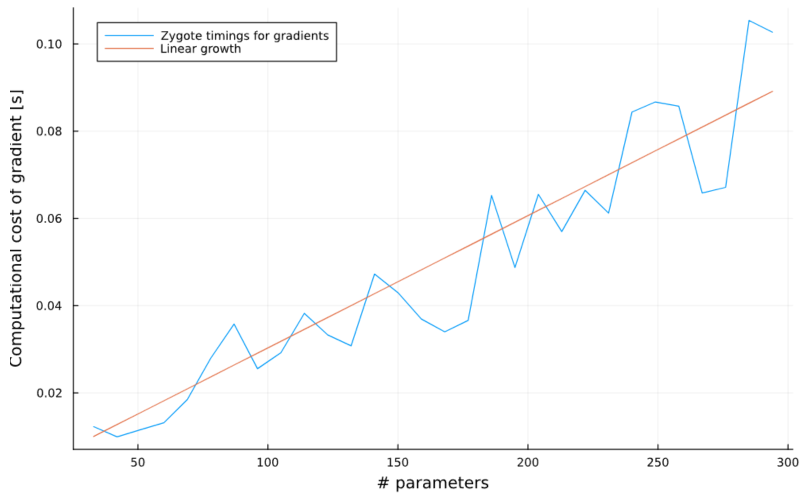


Figure 9: Computational cost for similarly sized NODEs using adjoint sensitivity analysis. The cost is significantly lower compared to the current implementation of LNN.

7 Discussion

We have through Lagrangian Neural Networks learned the dynamics of a double pendulum from data alone. This is quite a remarkable result, and Cranmer et al. have demonstrated that the power of Lagrangian mechanics also can benefit from the power of neural networks. After performing several experiments using our implementation of Lagrangian Neural Networks, it's worth discussing the results we have obtained, address weaknesses, and suggest new areas worth investigating.

In the preceeding section, we compared the performance of LNN to NODEs. While both had fairly equal starting point in terms of training accuracy, we certainly saw stronger predicting abilities from the LNN compared to the NODE. While the former showed trajectories similar to analytical, the latter revealed dynamics which spiraled out of control. This performance was quantified by the energy evolution, where the LNN give a small oscillating behavior, yet constant in trend, around the true constant energy. The NODE on the other hand, showed unstable energy evolution.

While the LNN certainly seems to outperform the NODE on this dynamical system, it's worth pointing out that this system in particular are prone to instabilities for the NODE, as its chaotic. As the system is both chaotic and energy conserving, it serves as a school example of a system Lagrangian mechanics will outperform Newtonian mechanics, and the results should be interpreted accordingly. This, in addition to the augmented training scheme, might significantly influence the results. However, as Cranmer obtained results agreeing with ours, we are quite confident that we would obtain similar results with an AD gradient, which would make the use of an augmented scheme redundant.

On the topic of computational performance, the NODE definitely outperforms the current implementation of LNN. As we saw, the growth of the computation of the sensitivity of the NODE beats LNN by a factor of approximately 250 per parameter, even though the evaluation of $\mathcal{F}$ was found to only differ by a factor of approximately 2.5. This is mainly due to the tremendous expense of computing the gradient by finite differences. The computational time of this single operation grows linearly with the number of parameters, which in turn grows super-linearly with the number of neurons in each layer (dependent on the architecture). This prevents the possibility of making a flexible network, which is absolutely necessary for Lagrangians much more complicated than the one defined in eq. (11).

The most vital weakness of our implementation is the means of finding the sensitivities $\partial_p C(p)$. As much as we hoped we could find this sensitivity using Julia's strengths, namely its effectiveness on automatic differentiation (AD), this turned out to be harder than expected. Having tried all known combinations of AD methods from packages such as Zygote, ForardDiff, ReverseDiff, TaylorDiff, and Enzyme, nothing seemed to tackle our nested-differentiation right hand side (4). According to staff at JuliaLab at MIT, and after finding several similar issues on Github, it seems to be a well-known problem that the AD methods, especially Zygote.jl, struggle on nested differentiation. Much of the problem seems to be that many of these packages mutate objects when calculating Hessian or Jacobian in order to reduce allocations. This in turn prohibits later calculating the gradient, as for example Zygote's gradient doesn't accept functions which mutate objects. Replacing all AD calls with ForwardDiff does not fix the problem either; ForwardDiff is not compatible with the pseudoinverse function in LinearAlgebra.jl pinv() which cannot perform svd!() on matrices using ForwardDiff's dual numbers. A minimum example is given below:

```
f = p -> [p p; p -p]
g = p -> pinv(f(p))
ForwardDiff.gradient(p -> g(p), p)
# MethodError: no method matching svd!(::Matrix{ForwardDiff.Dual ...
```

Replacing the pinv() function opens another can of worms, as the Hessian has been found to be close to or singular very often during training or solving, causing singular errors or extremely stiff $\mathcal{F}$ as $\det(\nabla_{\dot{q}}\nabla_{\dot{q}}^T\mathcal{L})^{-1} \to \pm\infty$ when it is close to singular which not even the autoswitching ODE-solvers seem to handle. Implementing an SVD-algorithm that is compatible with ForwardDiff is beyond the scope of this project.

To find the sensitivity, we found it necessary to perform the differentiation by the finite difference provided in the FiniteDiff package. While this dramatically reduce the efficiency and certainly limits our results, it nevertheless provides a way to find the sensitivity and hence enables implementation of the method. It's also worth mentioning that our scheme very easily could be enhanced, once an AD library in Julia like Zygote proves able to differentiate nested AD functions, adjoint sensitivity analysis would be possible and speed up performance – if so this could be implemented by essentially changing one line of code by adding SciMLSensitivity's keyword option sensealg to the LNN's solve call, which would automatically define the gradient of the ODE-solution. At that point, our code could be added onto the package DiffEqFlux.jl with minor changes, as we envisioned in our project proposal. With the current implementation's speed due to the heavy cost of computing gradients however, we deem this to be be out of reach as of now.

Finite difference gradients are computationally expensive in our case as they involve solving an ODE system for each perturbation in p, which inhibits effective learning on larger systems. In particular, for each training iteration, we must perform the ForwardSolve($\mathcal{F}_p, x_0$)-method mp times, with a finite difference method containing an m point stencil. In our scenario,

$$\mathcal{F}_p = \begin{bmatrix} \dot{q} \\ (\nabla_{\dot{q}}\nabla_{\dot{q}}^T\mathcal{L})^{-1}[\nabla_q\mathcal{L} - (\nabla_q\nabla_{\dot{q}}^T\mathcal{L})\dot{q}] \end{bmatrix}$$

is a fairly expensive right-hand-side, including both a Hessian and a matrix pseudoinverse. Changing to automatic differentiation will reduce to only one dual number calculation of ForwardSolve($\mathcal{F}_p, x_0$) for each training iteration which is probably faster and is more accurate (Ma et al. 2021) than the finite difference scheme.

As it often is with neural networks, the architecture has a large influence on the results, and it often is hard to predict or explain the behaviour of the network. Cranmer concluded a network of 128 neurons in each of their two hidden layers worked well (*A self-contained tutorial for LNNs* n.d.). When experimenting with different sizes, we found a network exceeding around 20 neurons in each hidden layer gave very unstable (stiff) solutions during training, causing solving to slow down as the number of steps needed to solve the ODEs adaptively increases or as more computationally expensive implicit solvers kick in. When training the large LNN with 128 neurons in each hidden layer, we didn't encounter instability, but that might be due to the augmented training scheme performed in advance. Since the underlying Lagrangian mechanics do not result in stiff ODEs in our case, the stiffness is an issue encountered during training. It's hard to conclude an optimal architecture in the general case, but it seems that the model with two hidden layers, together with Softplus as activations, performs well on this particular problem.

Because of these problems with the computational expense of taking gradients, we found that training using simple non-adaptive solvers like Euler(), Heun(), VCAB3(), and RK4() performed well during training. Although these methods will give large errors with respect to the actual solution of the LNN's ODE when it is very stiff, the resulting large values of the cost function $C(x_p)$ incurred by the erroneous steps due to stiffness will likely generate gradients with respect to p that favor lowering the extreme truncation error due to stiffness, as the data from the true system likely does not include such extreme jumps in state. In essence, we argue that the gradients generated by the non-adaptive solvers will result in parameters which make the LNN give less stiff ODEs, which again enables training with the more accurate adaptive methods down the line. Therefore, an initial training scheme using simple non-adaptive methods which are far less computationally expensive could enable much faster training of the LNN. The original paper by Cranmer et al. does not address these issues at all; in their implementation in Jax, they rely on jax.experimental.odeint (Cranmer et al. 2020, (code link, line 59)), a Dormand-Prince45 method, and calculate gradients using Jax's built in autodifferentiation to calculate the gradients without issue. It should however be noted that they did offer up an initialization scheme for the LNNs parameters to ease training.

8 Conclusion

We have in this paper presented our implementation of Lagrangian Neural Networks, a concept first proposed by Cranmer et al. In addition to a technical description of our implementation, we illustrated a use case on the double pendulum - a simple yet chaotic dynamical system. Furthermore, we compared the LNN with its natural competitor, the Neural ODE, on the topics of prediction, energy conservation, and computational performance. We found the LNN to outperform the NODE on prediction and energy conservation, but its performance is still quite poor compared to the NODE. This is in large due to the lack of a better way than finite differences to find the sensitivity. However, with an automatic differentiation method which can handle nested functions, our scheme is very easily enhanced to a tremendous degree.

References

A self-contained tutorial for LNNs (n.d.). https://colab.research.google.com/drive/1CS-xfrnTX28p1difoTA8uIYw0zytJkq#scrollTowmRTRTz9SJm8. Accessed: 2023-04-15.

Cranmer, Miles et al. (2020). *Lagrangian Neural Networks*. arXiv: 2003.04630 [cs.LG].

Greydanus, Sam, Misko Dzamba and Jason Yosinski (2019). *Hamiltonian Neural Networks*. arXiv: 1906.01563 [cs.NE].

Lagrangian Mechanics (Wikipedia) (n.d.). https://en.wikipedia.org/wiki/Lagrangian_mechanics. Accessed: 2023-04-15.

Lancaster, Tom and Stephen Blundell (2014). *Quantum field theory for the gifted amateur*. eng. Oxford.

Ma, Yingbo et al. (2021). *A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions*. arXiv: 1812.01892 [cs.NA].