Learning Ordinary Differential Equations with the Line Integral Loss Function

Albert Johannessen
Norwegian University of Science and Technology
albertjo@stud.ntnu.no

Abstract

A new training method for learning representations of dynamical systems with neural networks is derived using a loss function based on line integrals from vector calculus. The new training method is shown to learn the direction part of an ODE vector field with more accuracy and faster convergence compared to traditional methods. The learned direction can then be combined with another model that learns the magnitude explicitly to decouple the learning process of an ODE into two separate easier problems. It can also be used as a feature generator for time-series classification problems, performing well on motion classification of dynamical systems. The new method does however have multiple limitations that overall make the method less generalizable and only suited for some specific type of problems.

1 Introduction

The theory of dynamical systems is widely applicable for modeling problems in fields like physics, engineering and medicine. This approach will often rely on creating explicit system models resulting in differential equations. But there are systems where creating explicit models are too difficult to realistically achieve for various reasons. This motivates the use of deep learning based methods to learn representations for dynamical systems.

The Neural ODE framework [1] [2] [3] defines a neural network architecture as the limit of a residual network, thus ending up with an ordinary differential equation. The forward pass is computed by numerically integrating the network ODE forwards in time, and the backward pass by integrating the adjoint system backwards in time. Because of the inherent ODE structure of Neural ODEs they are capable of learning dynamical systems by training on sampled trajectories.

This paper presents an alternative training method by utilizing a new loss function based on line integrals. The training method is slightly faster compared to adjoint based methods, while also generalizing better when learning ODEs. However, there are multiple limitations with the new training method which makes it overall less general. The loss function is derived in the next section.

2 The Line Integral Loss

2.1 Loss Function

Consider an autonomous ODEs on the form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$ with state vector $\boldsymbol{x} \in \mathbb{R}^m$ and system dynamics $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^m$. Initial conditions $\boldsymbol{x}(t_0)$ will then evolve in time such that the trajectory $\boldsymbol{x}(t)$ in the state space is tangent to the vector field $\boldsymbol{f}$.

Now consider the case where the system dynamics $\boldsymbol{f}$ are unknown but with access to a dataset consisting of n trajectories $\boldsymbol{x}_1(t), \boldsymbol{x}_2(t), \ldots, \boldsymbol{x}_n(t)$ where each trajectory is sampled from the system at discrete points in time $t_0, \dots, t_1$. A method for training neural network representations of $\boldsymbol{f}$ can be derived by using the fact that trajectories are tangent to the underlying vector field. Define a neural network on the form $\dot{\boldsymbol{x}} = \boldsymbol{h}(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{h}(\boldsymbol{x}; \boldsymbol{\theta})$ is the represented as a neural network parametrized by $\boldsymbol{\theta}$. The line integral of a vector field $\boldsymbol{h}(\boldsymbol{x}; \boldsymbol{\theta})$ and trajectory $\boldsymbol{x}_k(t)$ is defined as $\int \boldsymbol{h} \cdot d\boldsymbol{x}_k = \int_{t_{k_0}}^{t_{k_1}} \boldsymbol{h}(\boldsymbol{x}_k(t); \boldsymbol{\theta})^T \boldsymbol{x}'_k(t) dt$ with $\boldsymbol{x}'_k(t) = \frac{d\boldsymbol{x}_k(t)}{dt}$.

A key observation is that the line integral takes its maximum value when the vector field and trajectory perfectly align, and its minimum value when they are perfectly opposed. However, when the vector field is represented by a neural network the line integral becomes potentially unbounded. This can be solved by normalizing both vectors inside the integral to make their lengths equal to 1. Dividing by the length of the interval makes the total line integral bounded between 1 and -1. Averaging over the trajectories in the training set then leads to the following optimization problem:

$$\min_{\boldsymbol{\theta}} \quad \mathcal{L} = -\frac{1}{n} \sum_{k=1}^{n} \frac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{h}(\boldsymbol{x}_k(t); \boldsymbol{\theta})}{||\boldsymbol{h}(\boldsymbol{x}_k(t); \boldsymbol{\theta})||}^T \frac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||} dt \tag{1}$$

2.2 Gradients

Solving this unconstrained optimization problem can be done with any gradient based methods commonly used in deep learning, but requires the gradient of the loss function to be known. The gradients from the loss function to the parameters can be derived to the form:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\frac{1}{n} \sum_{k=1}^{n} \frac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||}^T \left[\frac{\boldsymbol{I}}{||\boldsymbol{h}||} - \frac{\boldsymbol{h}\boldsymbol{h}^T}{||\boldsymbol{h}||^3} \right] \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}} dt \tag{2}$$

The vector-Jacobian product inside the integral can be computed efficiently using automatic differentiation. It is also worth noting that if all the trajectories have the same start time t_0 and end time t_1 it is possible to swap the order of the integral and the summation. This is often useful for vectorization purposes, which can give a significant speedup.

The full gradient derivation is given in Appendix A.

2.3 Practicalities

A significant limitation of the normalization is that the magnitude of the vector field is lost when training. In practice, the magnitude often goes towards 1 after training for enough epochs. The next section demonstrates some experiments that use loss function even with this limitation.

Another drawback of the loss function is that it requires access to the time derivatives of the trajectories in the dataset, unlike the standard Neural ODE approach which can learn dynamical system representations without derivatives. However, this limitation can be overcome by approximating the derivatives using a finite difference scheme if the sample interval h is small enough, and without too noisy data: $\frac{d\boldsymbol{x}_k(t_i)}{dt} \approx \frac{\boldsymbol{x}_k(t_{i+1}) - \boldsymbol{x}_k(t_i)}{h}$.

Computing the integral numerically must also be done with consideration to the discretely sampled trajectory points. A simple numerical method that works well enough is to use the trapezoid integration method, which can be implemented in an efficient way that does not require any more function calls than a simple Riemann sum approximation.

3 Experiments

3.1 Learning Dynamical Systems

A model is trained with the normalized line integral loss function and compared against other types of models to determine how well the training method works. All the models are set up with the same underlying neural network structure. The models to compare against are: a basic regression model that directly models the relation between a point $\boldsymbol{x}(t)$ and its time derivative $\dot{\boldsymbol{x}}(t)$, a Neural ODE (NODE) [1] that learns without knowing derivatives and finally a Second Order Neural ODE (SONODE) [4]. All four models are trained with the most suitable training algorithm for that model,

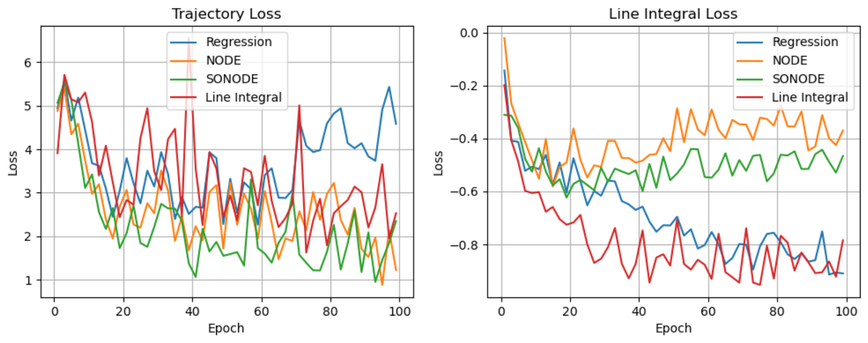

Figure 1: Loss values when training ODE models on a double pendulum system. Both loss functions are evaluated on the testing set.

independently of the other models. The regression model uses the MSE loss function applied to predicted derivatives and true derivatives when training, while the NODE models are trained by integrating numerically and comparing points along the trajectory also using the MSE loss, followed by gradient computation with the adjoint method.

The line integral model is set up to only learn the direction of the underlying vector field. Therefore the line integral model is combined with a regression model similar to the one above that only learns the magnitude. The final learned vector field is then the multiplication of the output of the two models. Decoupling the vector field into direction and magnitude turns out to be an easier problem to learn than directly regressing on the whole vector field.

The models are trained on the same trajectories sampled from a double pendulum system, which is nonlinear and chaotic making the learning process non-trivial. The trajectories are generated by integrating the system numerically. Normally distributed noise is also added to the training points, and the noise is added before the derivatives are computed with finite differences to make the setting more realistic. All the details regarding the experiment and implementation are described in Appendix B.

To compare the models, two different loss functions are computed over a testing set not seen during training each epoch. The first is similar to the loss for the NODE models in that trajectories are integrated from initial values and compared pointwise with the MSE loss function. The second is the normalized line integral computed over the learned vector fields of the models.

Figure 1 shows the resulting test losses from training all the models on the double pendulum. Looking at the trajectory loss it might seem that the basic regression model starts to overfit on the training data, and thus falls behind the other models which are all approximately performing the same. When looking at the line integral loss it is clear that the NODE and SONODE models perform significantly worse than the regression and line integral models.

3.2 Motion Classification

Now consider a time-series classification problem where the goal is to classify trajectories sampled from different dynamical systems. This experiment uses a double pendulum as the first system, and a double pendulum being controlled by a PD-controller as the second system, making a binary classification problem. A simple approach is to train a model consisting of an LSTM followed by a classifier head. This classifier will be used as a baseline for comparison. The second classifier will work by training two different line integral models on the two systems separately. To classify a new trajectory, line integrals are computed for both models and used as features for another classifier model. This experiment uses a Support Vector Classifier (SVC). More details are found in Appendix B.

Figure 2 displays the classification accuracy on the testing set for each epoch when training the classifiers described above. The SVC is reset and trained from scratch each epoch, to show how

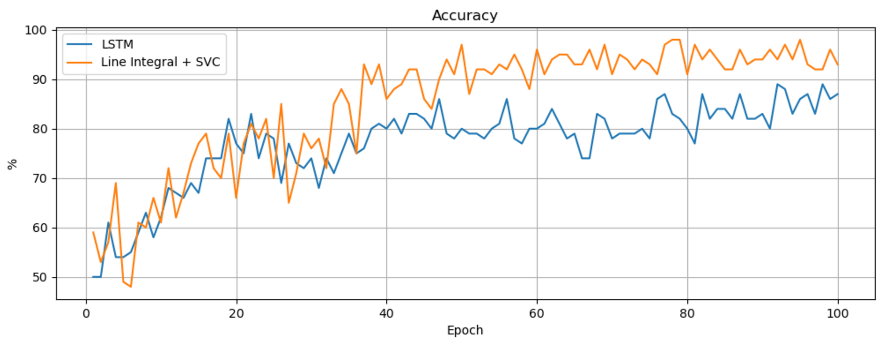

Figure 2: Classification accuracy from training two classifiers on time-series sampled from two different double pendulum systems. Evaluated on the testing set.

the accuracy improves as the line integral models learn their vector fields. The line integral + SVC classifier clearly outperforms the simple LSTM classifier.

4 Discussion

When learning dynamical systems, the line integral model is comparable in loss values to the NODE models, while significantly outperforming them on the line integral loss. This could be an indication that the line integral model learns the overall structure of the vector field better, while the NODE models are better at integrating trajectories that move initial points to their ending points without considering the details of the structure. It does however also require an additional model to learn the magnitude, which doubles the parameter count at the current setup. Other similar models could also have been useful to benchmark against to get a better overview, such as the Hamiltonian [5] and Lagrangian [6] Neural ODE models.

The line integral model is dependent on derivatives, so if the datasets were sampled at a lower frequency or contained too much noise, the derivatives could easily become too inaccurate to make the training reliable. Another consideration is that the training method only makes sense for learning dynamical systems directly, while the NODE models are more general and comparable to residual networks [7]. The gradients of the line integral loss is currently only defined to the parameters, and finding gradients to the input does not necessarily make as much sense because the whole trajectory must be considered as the input.

The motion classification experiment showcases how the line integral model can be used without the magnitude. The LSTM model is not optimized well and could probably gain much better performance. Other more advanced methods could also be used for comparison like a Neural CDE [8], but the main goal of the experiment above is to show that it is possible to use the feature generation in this manner. The current setup is also straightforward to extend to more than two classes.

The line integral method could also be extended to learning non-autonomous systems, meaning systems that also depend on time explicitly. The network could be constructed to handle time, for example taking inspiration from [9].

5 Conclusion

This paper presents a new method for learning representations for dynamical systems that is derived and experimentally tested. The method has some advantages over previous methods on certain problems, but also has many limitations making the method less useful in the general case. Future work could look into how to also learn the magnitude in a more efficient manner.

References

[1] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf.

[2] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes, 2019.

[3] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. Dissecting neural odes, 2021.

[4] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. On second order behaviour in augmented neural odes, 2020. URL https://arxiv.org/abs/2006.07220.

[5] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/26cd8ecadce0d4efd6cc8a8725cbd1f8-Paper.pdf.

[6] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks, 2020. URL https://arxiv.org/abs/2003.04630.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[8] Patrick Kidger, James Morrill, James Foster, and Terry J. Lyons. Neural controlled differential equations for irregular time series. *CoRR*, abs/2005.08926, 2020. URL https://arxiv.org/abs/2005.08926.

[9] Jared Quincy Davis, Krzysztof Choromanski, Jake Varley, Honglak Lee, Jean-Jacques E. Slotine, Valerii Likhosterov, Adrian Weller, Ameesh Makadia, and Vikas Sindhwani. Time dependence in non-autonomous neural odes. *CoRR*, abs/2005.01906, 2020. URL https://arxiv.org/abs/2005.01906.

[10] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook. 2008.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Appendices

A Gradient Derivation

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} &= \frac{\partial}{\partial \boldsymbol{\theta}} \left[-\tfrac{1}{n} \textstyle\sum_{k=1}^{n} \tfrac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \tfrac{\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})}{||\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})||}^T \tfrac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||} dt \right] \\
&= -\frac{1}{n} \sum_{k=1}^{n} \frac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \frac{\partial}{\partial \boldsymbol{\theta}} \left[\tfrac{\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})}{||\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})||}^T \tfrac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||} \right] dt \\
&= -\frac{1}{n} \sum_{k=1}^{n} \frac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||}^T \frac{\partial}{\partial \boldsymbol{\theta}} \left[\tfrac{\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})}{||\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})||} \right] dt
\end{aligned} \tag{3}$$

The partial derivative $\frac{\partial}{\partial \boldsymbol{\theta}} \left[\frac{\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})}{||\boldsymbol{h}(\boldsymbol{x}_k(t);\boldsymbol{\theta})||} \right] = \frac{\partial}{\partial \boldsymbol{\theta}} \frac{\boldsymbol{h}}{||\boldsymbol{h}||}$ can then be expanded using the chain rule:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \frac{\boldsymbol{h}}{||\boldsymbol{h}||} = \left[\tfrac{\partial}{\partial \boldsymbol{h}} \tfrac{\boldsymbol{h}}{||\boldsymbol{h}||} \right] \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}} \tag{4}$$

The innermost partial derivative $\frac{\partial}{\partial \boldsymbol{h}} \frac{\boldsymbol{h}}{||\boldsymbol{h}||}$ can be computed explicitly using the following formula for differentiating a normalized vector [10]:

$$\frac{\partial}{\partial \boldsymbol{x}} \frac{\boldsymbol{x} - \boldsymbol{a}}{||\boldsymbol{x} - \boldsymbol{a}||} = \frac{\boldsymbol{I}}{||\boldsymbol{x} - \boldsymbol{a}||} - \frac{(\boldsymbol{x} - \boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})^T}{||\boldsymbol{x} - \boldsymbol{a}||^3} \tag{5}$$

with $\boldsymbol{x} = \boldsymbol{h}$ and $\boldsymbol{a} = \boldsymbol{0}$. This leads to the expression for the gradient of the normalized vector as:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \frac{\boldsymbol{h}}{||\boldsymbol{h}||} = \left[\tfrac{\boldsymbol{I}}{||\boldsymbol{h}||} - \tfrac{\boldsymbol{h}\boldsymbol{h}^T}{||\boldsymbol{h}||^3} \right] \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}}$$

Inserting this into equation (3) leads to the final gradient expression:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\frac{1}{n} \sum_{k=1}^{n} \frac{1}{t_{k_1} - t_{k_0}} \int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||}^T \left[\tfrac{\boldsymbol{I}}{||\boldsymbol{h}||} - \tfrac{\boldsymbol{h}\boldsymbol{h}^T}{||\boldsymbol{h}||^3} \right] \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}} dt \tag{6}$$

Additionally, if all the sample times are the same $t_0, \ldots, t_1$, the order of the sum and integration can be swapped:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\frac{1}{n} \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \sum_{k=1}^{n} \frac{\boldsymbol{x}'_k(t)}{||\boldsymbol{x}'_k(t)||}^T \left[\tfrac{\boldsymbol{I}}{||\boldsymbol{h}||} - \tfrac{\boldsymbol{h}\boldsymbol{h}^T}{||\boldsymbol{h}||^3} \right] \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{\theta}} dt \tag{7}$$

B Experimental Details

B.1 Implementation

A PyTorch implementation of both experiments be found at this repository.

B.2 Data Generation

The controlled double pendulum system can be written as an ODE on the state-space form:

$$\frac{d}{dt}\begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \begin{bmatrix} (m_1+m_2)l_1^2 & m_2 l_1 l_2 \cos(q_1-q_2) \\ m_2 l_1 l_2 \cos(q_1-q_2) & m_2 l_2^2 \end{bmatrix}^{-1} \begin{bmatrix} -m_2 l_1 l_2 \sin(q_1-q_2)\dot{q}_2^2 - (m_1+m_2) g l_1 \sin q_1 + u_1 \\ m_2 l_1 l_2 \sin(q_1-q_2)\dot{q}_1^2 - m_2 g l_2 \sin q_2 + u_2 \end{bmatrix} \end{bmatrix} \tag{8}$$

with a four dimensional state vector $[q_1 \quad q_2 \quad \dot{q}_1 \quad \dot{q}_2]^T$ and two dimensional input vector $[u_1 \quad u_2]^T$. All experiments uses the parameters $m_1 = 1, m_2 = 1, l_1 = 1, l_2 = 1, g = 9.81$ for simplicity.

A new batch of size 200 is generated for every epoch during training by integrating the double pendulum (8) using a step size of $h = 0.01$ from $t_0 = 0$ to $t_1 = 1$. The initial values to be integrated are randomly sampled uniformly in a box around the origin. A testing batch of size 20 is also generated at every epoch to evaluate the performance on the two loss functions in Figure 1. Normally distributed noise with mean 0 and standard deviation 0.01 is added to the training batch, and then derivatives are computed with finite differences after the noise is added.

All the experiments are ran with the same seeding so that the data generation remains the same.

B.3 Learning Dynamical Systems

The input is set to $u_1 = u_2 = 0$.

All the trained models use the same neural network structure consisting of 6 linear layers with the $\tanh$ activation function in between layers. The layers has nodes: $4, 50, 100, 200, 100, 50, 4$, except for the SONODE with 2 output nodes and the magnitude regression part of the line integral model with 1 output node. All the models are trained with the Adam optimizer [11] using a learning rate of 0.001.

The line integral model is combined with the magnitude regression model by first normalizing the output of the line integral model and then multiplying with the magnitude. The line integral usually converges to a magnitude of 1 after some time, but the normalization makes it more stable when evaluating on the testing set early.

When training the line integral model it is not actually necessary to compute the line integral, only the gradients (2) are necessary. But the line integral can still be useful to compute to determine if the loss is decreasing.

Finally, the two loss functions evaluated on the testing set are the trajectory loss and the line integral loss. The trajectory loss takes the first values of the test batch and integrates these as intial values on the trained model. During integration, points are sampled at the same points in time as the testing batch. The integrated trajectories are then compared to the full testing trajectories using the MSE loss. The line integral loss function is computed by using the derivatives of the testing trajectories with the trained models. The derivatives of the testing batch are also approximated using finite differences, but without any noise.

B.4 Motion Classification

Two systems are now used for data generation: one uncontrolled double pendulum similar to the previous experiment and one controlled with a PD-controller. The controller is on the form: $u_1 = -0.1q_1 - 0.01\dot{q}_1$ and $u_2 = -5(q_2 - q_1) - 2\dot{q}_2$. This ends up driving the second joint angle towards the same angle as the first joint, while also slowing down the overall system. This causes the pendulum to appear more constrained in its motion.

The LSTM classifier consists of an LSTM layer with 300 hidden units followed by a neural network with layers of nodes: $300, 150, 1$ and sigmoid activations in between. It is trained as a binary classification problem on trajectories from the two systems using the Adam optimizer using a learning rate of 0.001 and the binary cross entropy loss function.

The second model trains two line integral models separately on the two double pendulum systems. These models are used as feature generators by computing normalized line integrals with trajectories to see how well they align with the two vector fields. These alignments can then be used to classify them. The simplest classifier would be to simply see what double pendulum system the trajectory aligns the most with. However, as the two systems can produce similar looking trajectories, this approach will not always be perfect. So for each trajectory to be classified, the line integral of both models are used as features for a new classifier model, here using an SVC. Figure 2 shows accuracy over time by re-training the SVC from scratch each epoch, but the simplest approach would be to fully train the two line integral models first, and then train the classifier afterwards.