DLKoopman: A deep learning software package for Koopman theory

Sourya Dey SOURYA@GALOIS.COM
Eric William Davis EWD@GALOIS.COM
421 SW 6th Avenue #300, Portland, OR 97204, USA

Editors: N. Matni, M. Morari, G. J. Pappas

Abstract

We present DLKoopman – a software package for Koopman theory that uses deep learning to learn an encoding of a nonlinear dynamical system into a linear space, while simultaneously learning the linear dynamics. While several previous efforts have either restricted the ability to learn encodings, or been bespoke efforts designed for specific systems, DLKoopman is a generalized tool that can be applied to data-driven learning and analysis of any dynamical system. It can either be trained on data from individual states (snapshots) of a system and used to predict its unknown states, or trained on data from trajectories of a system and used to predict unknown trajectories for new initial states. DLKoopman is available on the Python Package Index (PyPI) as 'dlkoopman', and includes extensive documentation and tutorials. Additional contributions of the package include a novel metric called Average Normalized Absolute Error for evaluating performance, and a ready-to-use hyperparameter search module for improving performance.

Keywords: koopman theory, deep learning, python package, autoencoder, software tool

1. Introduction

The abundance of data, combined with the rise in computational power, has enabled the creation of increasingly powerful machine learning systems to model and predict the world around us. In particular, *deep learning* – i.e. machine learning implemented via neural networks (NNs) comprising multiple layers – has become extremely popular due to its ability to 'intelligently' learn the rules of any system from its available data. One such data-driven application of deep learning is to learn the dynamics of a system purely from its states, or snapshots. This is especially useful in situations where the exact rules governing the system are prohibitively hard and/or time-consuming to understand and analyze, whereas data can be easily and plentifully collected from the system.

Dynamic Mode Decomposition (DMD) is a technique to analyze nonlinear systems by approximating them using linear dynamics, i.e. *linearizing* them. However, an arbitrary nonlinear system will, in general, be poorly suited to linearization. *Koopman theory*[1], first introduced in Koopman (1931), overcomes this limitation by extending DMD to encode states of a nonlinear system into *observables* in a different domain, performing linearization in this *encoded domain*, then decoding back into the original input domain. This domain shift is critical to improving the fidelity of the linear model, leading to significantly better approximations. The obtained linear model is incredibly powerful since linear techniques can be used to easily *predict unknown states* of the system, as well as *predict entire trajectories* of how the system behaves starting from new initial conditions.

1. Also known as Koopman operator theory, or just Koopman operator.

© 2023 S. Dey & E.W. Davis.

A key challenge with implementing Koopman theory is finding an encoding for the original input domain into a linearizable domain. Particularly suited to this task is the *autoencoder* deep learning architecture, which consists of an encoder NN learning to convert input data to an encoded domain suited to linear approximations, connected to a decoder NN simultaneously learning the inverse function of the encoder so as to convert back to the original input domain.

Our core contribution in this paper is introducing DLKoopman – an open-source Python package to implement Koopman theory. It can be installed via `pip install dlkoopman`. To the best of our knowledge, we are the first effort to create a software package for Koopman theory that is a) *general and reusable*, in the sense that it can operate on data from any system and perform state prediction, as well as trajectory prediction, and b) *complete*, in the sense that it uses a deep learning pipeline that includes learning both – the encoding into a linearizable domain, and the corresponding linear dynamics. DLKoopman bridges the gap between two schools of prior work – a) software packages that use DMD without learning an encoding into a domain that ensures good linearization, and b) efforts that use deep learning for the encoding, but lack general software tooling and are usually restricted to trajectory prediction.

We make two additional contributions via the package. Firstly, we introduce a novel error function – *Average Normalized Absolute Error (ANAE)* – which is useful for quantifying performance and comparing different models. Secondly, we include a ready-to-use *hyperparameter search module* that can provide significant performance gains when performing Koopman approximations.

1.1. Mathematical background

The mathematical details relevant to this paper are in Dey (2022), which we summarize here. For a more extensive mathematical treatment, the reader can refer to several sources such as Kutz et al. (2016); Tu et al. (2014); Williams et al. (2015). Suppose a system is described as $\frac{d\boldsymbol{x}}{di} = f\left(\boldsymbol{x}(i)\right)$, where $\boldsymbol{x}$ is the (generally multi-dimensional) state of the system indexed by i, which may, but need not necessarily, be time. The system can be sampled to obtain its states at various discrete indexes, which can be described as $\boldsymbol{x}_{i+1} = F\left(\boldsymbol{x}_i\right)$. Here, $f(\cdot)$ and $F(\cdot)$ are (generally nonlinear) functions encapsulating the dynamics of how the system evolves. The first step in applying Koopman theory is to transform the original $\boldsymbol{x}$ domain into an encoded domain $\boldsymbol{y}$ using an encoder $g(\cdot)$:

$$\boldsymbol{y} = g\left(\boldsymbol{x}\right) \tag{1}$$

State evolution in the encoded domain is linear, i.e. $\frac{d\boldsymbol{y}}{di} = \boldsymbol{\mathcal{K}}\boldsymbol{y}(i)$, where $\boldsymbol{\mathcal{K}}$ is the Koopman matrix. This can be solved as:

$$\boldsymbol{y}(i) = e^{\boldsymbol{\mathcal{K}}i}\boldsymbol{y}(0) = \boldsymbol{W}e^{\boldsymbol{\Omega}i}\boldsymbol{W}^{\dagger}\boldsymbol{y}(0) \qquad \forall i \in \mathbb{R} \tag{2}$$

The Koopman matrix characterizes the system and contains information about it; in particular, its eigenvectors $\boldsymbol{W}$ are referred to as the DMD modes, and the associated eigenvalues $\boldsymbol{\Omega}$ govern how the system behaves as it is evolved. Linearization is an incredibly powerful technique since any unknown state of a system can be easily computed from any of its known states using well-developed linear techniques, as done above to compute the unknown $\boldsymbol{y}(i)$ state from the initial state $\boldsymbol{y}(0)$.

In the discrete sampled equivalent of the above, the system is described as $\boldsymbol{y}_{i+1} = \boldsymbol{K}\boldsymbol{y}_i$, where $\boldsymbol{K}$ is the Koopman matrix. This can be solved as:

$$\boldsymbol{y}_i = \boldsymbol{K}^i\boldsymbol{y}_0 \qquad \forall i \in \mathbb{Z} \tag{3a}$$

$$= \boldsymbol{W}\boldsymbol{\Lambda}^i\boldsymbol{W}^{\dagger}\boldsymbol{y}_0 \qquad \forall i \in \mathbb{Z} \tag{3b}$$

The eigenvalues $\mathbf{\Lambda}$ from the discrete case are related to the general eigenvalues $\mathbf{\Omega}$ as $\mathbf{\Lambda} = e^{\mathbf{\Omega}\Delta i}$, where Δi is the sampling interval for discretization.

The encoded domain is finally decoded to obtain states in the original domain:

$$\boldsymbol{x} = g^{-1}(\boldsymbol{y}) \tag{4}$$

1.2. Highlights of DLKoopman

The core components of our DLKoopman package are a) an autoencoder architecture that learns an encoder $g(\cdot)$ suited to linearization, and its corresponding decoder $g^{-1}(\cdot)$, and b) a method to learn the Koopman matrix. The latter can be achieved in two different ways, which also correspond to the two primary tasks the package performs – *state prediction* and *trajectory prediction*.

1.2.1. STATE PREDICTION

The goal here is to learn the dynamics of a system from its known states, then predict the values of its unknown states. As an example application, Sec. 2.4.1 describes DLKoopman performing state prediction of the pressure on the surface of an airfoil at unknown angles of attack.

The input data is $\{\boldsymbol{x}_i, i \in I \subset \mathbb{R}\}$, where $I = \{i_0, i_1, \cdots, i_m\}$ is a set of indexes at which the states of the system have been measured. These input states are encoded to $\{\boldsymbol{y}_i\}$, and grouped into matrices $\boldsymbol{Y}_{\text{prev}} = \left[\boldsymbol{y}_{i_0} \cdots \boldsymbol{y}_{i_{m-1}}\right]$ and $\boldsymbol{Y}_{\text{next}} = [\boldsymbol{y}_{i_1} \cdots \boldsymbol{y}_{i_m}]$. The Koopman matrix can be computed as $\boldsymbol{K} = \boldsymbol{Y}_{\text{next}}\boldsymbol{Y}_{\text{prev}}^{\dagger}$, which uses Singular Value Decomposition (SVD). The subsequent workflow involves computing its eigendecomposition $\boldsymbol{W}$ and $\mathbf{\Lambda}$, using $\mathbf{\Lambda}$ to compute $\mathbf{\Omega}$, then using Eq. (2) to compute $\{\boldsymbol{y}_{i'}, i' \in I' \subset \mathbb{R}, I' \cap I = \emptyset\}$. Finally, these are decoded to get $\{\boldsymbol{x}_{i'}\}$, which are the predicted states of the original system at indexes $\{i'\}$ where they were not measured. (Values of i' can be positive, negative, or fractional, which respectively correspond to forward extrapolation, backward extrapolation, or interpolation.) All of these steps are included in the deep learning pipeline.

1.2.2. TRAJECTORY PREDICTION

Classical applications of Koopman theory provide data in the form of trajectories, i.e. 'rollouts' of a system from an initial state for a fixed number of indexes into the future. The goal of trajectory prediction is to learn the dynamics of the system from a given number of known trajectories, then predict unknown trajectories starting from new initial states.

The input data is $\left\{\left[\boldsymbol{x}_0^j, \boldsymbol{x}_1^j, \cdots, \boldsymbol{x}_m^j\right], j \in \{j_1, j_2, \cdots, j_J\}\right\}$ – the sequence inside square brackets is a trajectory j starting from initial state $\boldsymbol{x}_0^j$ and rolled out up till state $\boldsymbol{x}_m^j$; there are J such given trajectories. The pipeline begins by encoding all states in each trajectory. While the Koopman matrix can be computed using SVD as before, it can be slow for lengthy trajectories. Hence, for trajectory prediction, the DLKoopman package models the Koopman matrix $\boldsymbol{K}$ as the weights of a linear NN layer – a multi-layer perceptron (MLP) with equal number of input and output neurons, no bias, and no activation function or other source of nonlinearity. This can be directly applied to evolve the system instead of performing the eigendecomposition. A linear layer incurs the significant limitation of not being able to predict states for negative or fractional indexes since it cannot be applied backwards or a fractional number of times, however, this limitation is irrelevant to trajectory prediction since trajectory indexes are only integral and always advance forward.

Then, given new initial states $\left\{\boldsymbol{x}_0^{j'}, j' \notin \{j_1, j_2, \cdots, j_J\}\right\}$, the trained linear layer $\boldsymbol{K}$ can be used to generate complete trajectories $\left\{\left[\boldsymbol{x}_0^{j'}, \boldsymbol{x}_1^{j'}, \cdots, \boldsymbol{x}_m^{j'}\right]\right\}$ for each of them using Eq. (3a).

1.3. Related Work

We first discuss other general-purpose software packages in literature that implement Koopman theory. The Python package `PyKoopman` (Kaiser and de Silva (2020)) – built using the `PyDMD` package (Demo et al. (2018)) for performing DMD only – can do state and trajectory prediction. However, the user needs to provide a ready-made encoding $\boldsymbol{y}$ as input. In the absence of such an input, the encoder used is an identity function, i.e. $\boldsymbol{y} = \boldsymbol{x}$, wherein Koopman theory reduces to DMD only. The Python package `pykoop` (Dahdah and Forbes (2022)) allows the user to use particular functions for encoding $\boldsymbol{x}$ into $\boldsymbol{y}$, such as radial basis functions. A related example is the Matlab toolbox `koopman` (Budisic (2017)), which constructs $\boldsymbol{y}$ as the Fourier Transform of $\boldsymbol{x}$. We note that these packages impose a certain amount of restriction regarding how the encoded states are derived from the input states.

The motivation to learn encoded states unrestrictedly with sole focus on achieving good linearization led to the development of deep learning models such as autoencoders to obtain $\boldsymbol{y}$ from $\boldsymbol{x}$. There have been other efforts along these lines, generally built to train on input trajectories and perform trajectory prediction. While code exists for some of these efforts, such code is usually bespoke and serves to demonstrate the specific results in the respective papers.

Lusch et al. (2018); Champion et al. (2019) focus on finding parsimonious / sparse representations for dynamical systems using autoencoders. In particular, Lusch et al. (2018) learns the Koopman matrix using a linear NN layer, and includes an auxiliary network to learn continuous eigenvalues. Alford-Lago et al. (2022) created the deep learning DMD (DLDMD) effort that has some similarity to our work, but lacks code. Takeishi et al. (2017) uses linear-delay embedding for time-series data. Azencot et al. (2020) uses recurrent NNs to learn consistent dynamics, similar to Otto and Rowley (2019)'s Linearly Recurrent Autoencoder Network (LRAN) to learn low-dimensional encodings, while Geneva and Zabaras (2022) uses transformers. Otto and Rowley (2019); Williams et al. (2015) learn the elements of the Koopman matrix directly, following which they perform its eigendecomposition. Yeung et al. (2017); Li et al. (2017) use feed-forward networks to encode the original data, but they do not convert back to the original input domain using a decoder. In particular, Yeung et al. (2017) uses NN layers to learn the Koopman matrix, while Li et al. (2017) learns it directly. Mardt et al. (2018); Wehmeyer and Noé (2018) use deep learning for Koopman theory applied to the specific domain of molecular kinetics. Finally, note that the very recent work of Lew (2023) discusses how numerical differentiation may overcome the limitation of linear layers being restricted to predicting positive integral indexes only. However, we reiterate that learning the Koopman matrix directly, as discussed in Sec. 1.2.1, completely overcomes the problem by generalizing prediction to any real-valued index.

2. The DLKoopman package

This section describes our core contribution – DLKoopman. The DLKoopman Python package is available on PyPI and can be installed via `pip install dlkoopman`. The current version is `1.1.2` at the time of final submission, and can be cited using Dey (2023). Source code is available at `https://github.com/GaloisInc/dlkoopman`. The `README` gives a broad overview

of the package and links to tutorials. The complete API reference and documentation can be found at https://galoisinc.github.io/dlkoopman/.

2.1. Training

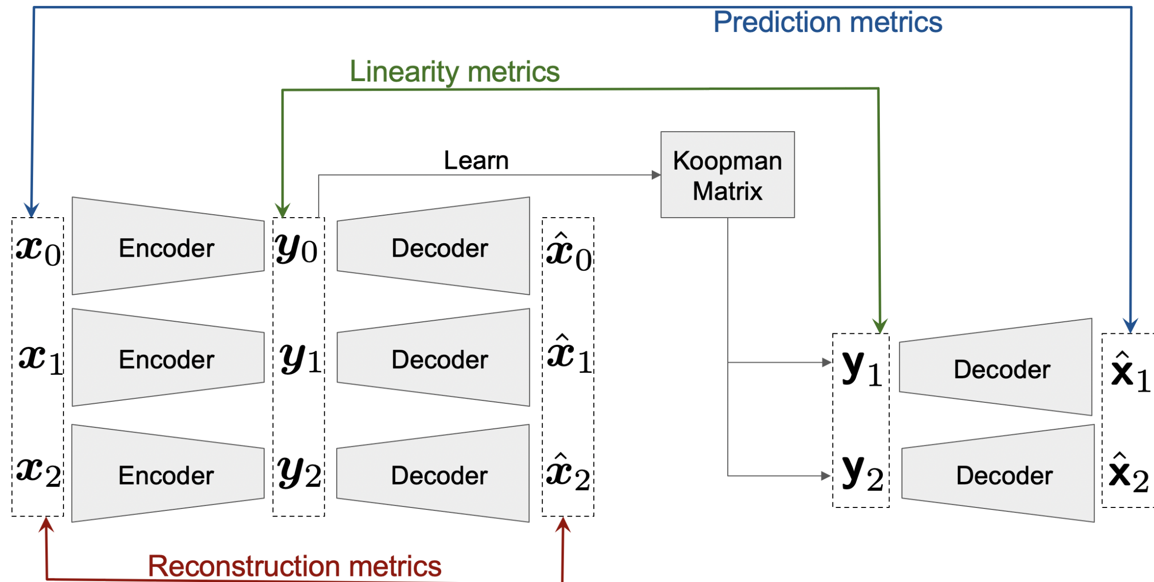

Figure 1: The training algorithm for a small example with three input states $[\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2]$, which can be input to either `StatePred` or `TrajPred` (trajectory superscript omitted in figure). These are passed through an encoder to get encoded states $[\boldsymbol{y}_0, \boldsymbol{y}_1, \boldsymbol{y}_2]$. These are passed through a decoder to get $[\hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2]$, and also used to learn the Koopman matrix. This is used to derive predicted encoded states $[\mathbf{y}_1, \mathbf{y}_2]$, which are passed through the same decoder to get predicted approximations $[\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2]$ to the original input states. Training proceeds by minimizing errors – reconstruction between $\{\boldsymbol{x}\}$ and $\{\hat{\boldsymbol{x}}\}$, linearity between $\{\boldsymbol{y}\}$ and $\{\mathbf{y}\}$, and prediction between $\{\boldsymbol{x}\}$ and $\{\hat{\mathbf{x}}\}$.

The two core modules of the package are `StatePred` for state prediction, and `TrajPred` for trajectory prediction. They have a similar algorithm during the training phase, the aim of which is to optimize three metrics – reconstruction, linearity, and prediction. These are standard metrics found in Koopman theory literature (Lusch et al. (2018); Alford-Lago et al. (2022)). The training algorithm is described in Fig. 1.

The input data consists of states $\{\boldsymbol{x}\}$ of the system that we wish to model. `StatePred` requires the states to be accompanied by indexes, e.g. $\{\boldsymbol{x}_{10}, \boldsymbol{x}_{15}, \boldsymbol{x}_{19.5}, \cdots\}$ (for the purposes of training, indexes are rounded to have equal spacing, i.e. $\boldsymbol{x}_{19.5}$ will be converted to $\boldsymbol{x}_{20}$). `TrajPred` does not require indexes since the states are assumed to form a trajectory $\left[\boldsymbol{x}_0^j, \boldsymbol{x}_1^j, \boldsymbol{x}_2^j, \cdots\right]$. These states are passed through a MLP neural network – the encoder – to obtain encoded states $\{\boldsymbol{y}\}$. The user needs to provide the dimensionality of the encoded states via the required argument `encoded_size`.

The encoded states are passed through a MLP – the decoder – to obtain reconstructed states $\{\hat{\boldsymbol{x}}\}$. The *reconstruction error* is computed between $\{\boldsymbol{x}\}$ and $\{\hat{\boldsymbol{x}}\}$, minimizing this ensures the decoder is learning the inverse function of the encoder.

For `StatePred`, the encoded states are used to perform SVD to obtain the Koopman matrix. Its eigendecomposition is used to generate predictions $\{\mathbf{y}\}$ (note the different font) for encoded states by using Eq. (2). An important consideration here is the *rank* of the SVD. Using a lower rank usually results in approximations that generalize better by reducing overfitting, however, a rank too low will incur bias. A lower rank also reduces the probability of numerical issues when performing backpropagation (Contributors (2022b,a)). The user needs to provide `rank` since it's a required argument. For `TrajPred`, the initial encoded state $\boldsymbol{y}_0$ is evolved via a linear layer, which serves as the Koopman matrix. This can be used to generate the entire predicted trajectory $[\mathbf{y}_1, \mathbf{y}_2, \cdots]$ by using Eq. (3a). The *linearity error* is computed between $\{\boldsymbol{y}\}$ and $\{\mathbf{y}\}$, minimizing this ensures a good Koopman matrix is being learnt that can achieve linearization.

The predictions for encoded states / trajectories are decoded to obtain predictions $\{\hat{\mathbf{x}}\}$ (note the different font) for input states / trajectories. The *prediction error* is computed between $\{\boldsymbol{x}\}$ and $\{\hat{\mathbf{x}}\}$, minimizing this ensures a good overall pipeline that learns the autoencoder and Koopman matrix.

2.2. New predictions

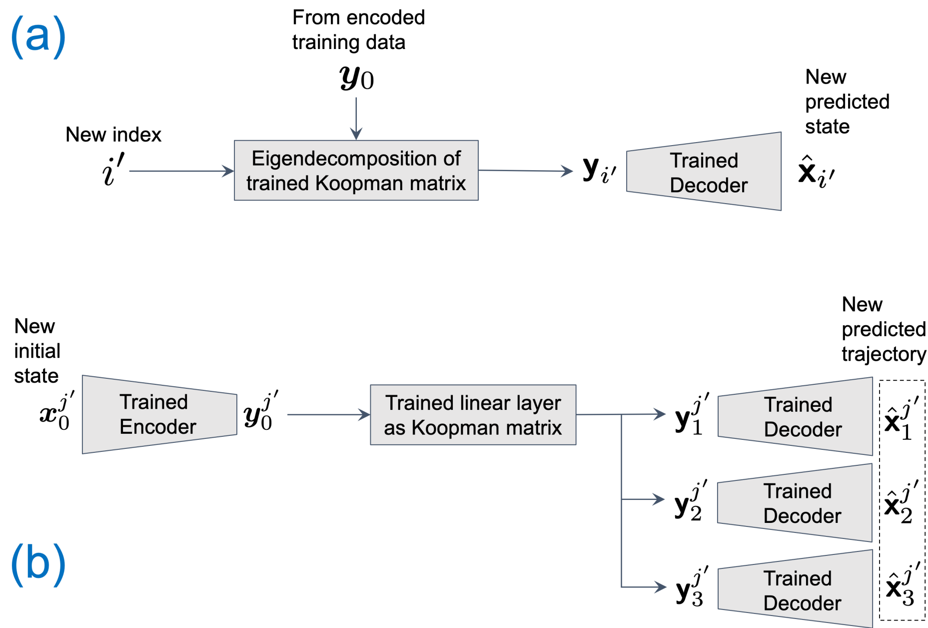

Figure 2: After training, (a) the `StatePred` can be used to compute predicted states for new indexes such as i', (b) the `TrajPred` can be used to generate predicted trajectories for new starting states such as $\boldsymbol{x}_0^{j'}$.

New predictions can be computed after training finishes. For `StatePred`, given any new index i' not present in the input data, the architecture can predict $\hat{\mathbf{x}}_{i'}$, as shown in Fig. 2(a). For

`TrajPred`, given any new initial state $\boldsymbol{x}_0^{j'}$ not present in the input data, the architecture can predict the trajectory j' as $\left[\hat{\mathbf{x}}_1^{j'}, \hat{\mathbf{x}}_2^{j'}, \cdots\right]$, as shown in Fig. 2(b).

2.3. Metrics

As described in Sec. 2.1, three metrics are computed to judge the goodness of a training run – reconstruction, linearity, and prediction. The overall loss function is:

$$L = L_{\text{lin}} + \alpha \left(L_{\text{recon}} + L_{\text{pred}}\right) + \beta L_{\text{Autoencoder}} + \gamma L_{\boldsymbol{K}} \tag{5}$$

This overall loss L is optimized when performing backpropagation to train the deep learning architecture. L_{lin}, L_{recon} and L_{pred} are computed using Mean Squared Error (MSE). α is a term that is used to weight the losses for quantities that are outputs from the decoder, i.e. reconstruction and prediction loss. β is the coefficient of weight decay $L_{\text{Autoencoder}}$ for the autoencoder parameters. γ is the coefficient of a penalty $L_{\boldsymbol{K}}$ for the elements of the Koopman matrix. For `StatePred`, $L_{\boldsymbol{K}}$ is the average absolute value of the elements of $\boldsymbol{K}$, while for `TrajPred` – where $\boldsymbol{K}$ is a linear layer – $L_{\boldsymbol{K}}$ is weight decay for the parameters of the linear layer, and $\beta = \gamma$.

While losses are suitable for optimizing during gradient descent, they are dependent on the scale of the data. This makes them unsuitable for human-readability in terms of how well a `StatePred` run is performing. This motivates us to introduce a novel metric Average Normalized Absolute Error (ANAE), which can be defined between two tensors of same size as:

$$\text{ANAE}(\boldsymbol{p}, \boldsymbol{q}) = \text{Avg}_{p_i \neq 0}\left(\frac{|p_i - q_i|}{|p_i|}\right) \tag{6}$$

where $\boldsymbol{p}$ is the reference, and $\boldsymbol{q}$ is the prediction. Note that $\boldsymbol{p}$ and $\boldsymbol{q}$ can have any number of dimensions, they will be flattened before computing ANAE.

Thus, ANAE tells us how far off we can expect each element of the prediction to be from the corresponding element in the reference. As an example, consider the reference data to have two 3-dimensional states: $[-0.1, 0.2, 0]$ and $[100, 200, 300]$. Suppose a trained `StatePred` model produces predictions $[-0.11, 0.15, 0.01]$ and $[105, 210, 285]$ for this reference data. Normalizing the absolute deviations by the absolute reference values yields 10% and 25% for the first two values, the third value is ignored since its reference value is 0, and the last three values are each 5%. Taking the average of these yields 10%. This is a useful figure. It tells us that when predicting unknown states for which reference values are not available, the user can expect the current model to predict values that are $\sim 10\%$ off. Note that just like loss, ANAE can also be computed for reconstruction, linearity, and prediction. Possibly the most important out of these is the prediction ANAE, since it tells us the error we can expect from unknown state prediction.

2.4. Example Applications

The source code for DLKoopman includes complete tutorials in the `examples/` folder.

2.4.1. STATE PREDICTION

Here, we briefly describe an application of the `StatePred`. This example attempts to predict the pressure vector across the surface of a NACA0012 airfoil at varying angles of attack – a commonly

studied computational fluid dynamics (CFD) problem (Critzos et al. (1955); Voodarla (2021)). A particular state $\boldsymbol{x}_i$ for this case would be the 200-dimensional pressure vector $\boldsymbol{x}$ at angle of attack i.

The data consists of pairs of variables prefixed by `X` for the states, and `t` for the indexes. (Note that the code uses prefix `t` for indexes since `i` is a variable commonly reserved for iterators.) Three such pairs may be provided – `Xtr` and `ttr` form the training data that must be provided, while `Xva` and `tva`, and `Xte` and `tte`, respectively, are validation and test data that may be optionally provided to check the performance of training. For this example, each of the `t` variables is a list containing values of angles of attack, while each corresponding `X` variable is a matrix of dimensions (`length(t)` $\times$ 200) where each row contains the pressure vector for a particular angle of attack.

A `StatePred` instance can be created as:

```
sp = StatePred(
    dh = dh, #StatePredDataHandler instance used to pass data
    rank = 6,
    encoded_size = 50,
    encoder_hidden_layers = [100]
)
```

which results in an autoencoder that looks like Fig. 3(a). This can be trained (and validated, if `Xva` and `tva` are provided) as:

```
sp.train_net(
    numepochs = 1000,
    decoder_loss_weight = 0.1, #alpha in the loss equation
    weight_decay = 1e-5, #beta in the loss equation
    Kreg = 0 #gamma in the loss equation
)
```

An epoch of training comprises computing the Koopman matrix using the entire training data, using it to get predictions, then computing metrics on training data, and validation data if given. If `Xte` and `tte` are provided, test statistics may be obtained after training via `sp.test_net()`.

The `utils` module of the package contains a utility to plot loss and ANAE statistics. An example plot is shown in Fig. 3(b), which achieves 6.95% prediction ANAE on test data.

Now the pressure vector can be predicted at unknown angles of attack that were not present in either of `ttr`, `tva` and `tte`. As examples, we pick an *interpolated* index $3.75°$, and an *extrapolated* index $21°$. The predictions are yielded by running:

```
sp.predict_new([3.75,21])
```

2.4.2. Trajectory Prediction

The primary difference in a `TrajPred` run compared to a `StatePred` run is that each epoch is sub-divided into batches, with each batch containing a subset of training data trajectories. The starting index of each of these trajectories is evolved via the linear layer to get predictions for the rest of each trajectory, and compute training metrics. Validation metrics are computed at the end of all batches in an epoch, while test metrics are computed at the end of all epochs, as before.

The versatility of the DLKoopman package allows it to run on different kinds of data, which naturally also includes data sets that have been used in prior literature. The source code contains an

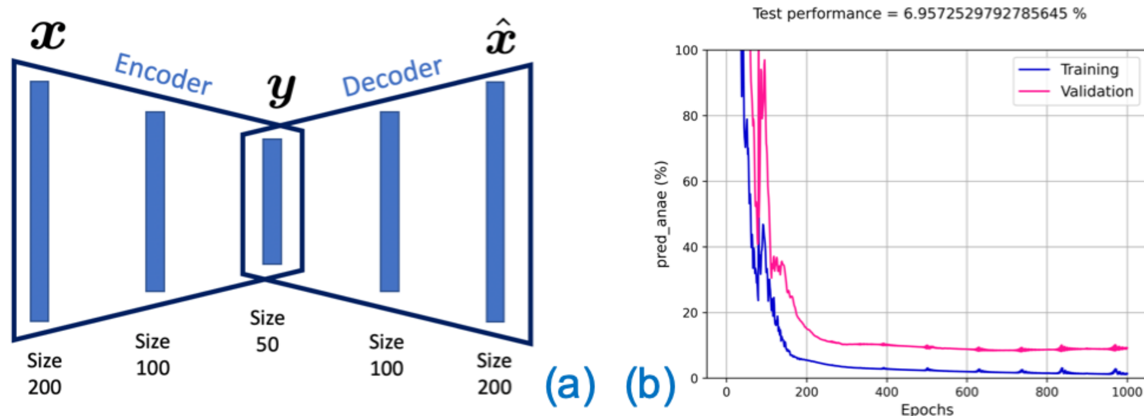

Figure 3: (a) Example autoencoder architecture. Dimensionality of the input states $\boldsymbol{x}$ and reconstructed states $\hat{\boldsymbol{x}}$ is 200, and that of the encoded states $\boldsymbol{y}$ is `encoded_size=50`. The encoder is set to have one hidden layer of dimensionality `encoder_hidden_layers=[100]`. The decoder here is the mirror image of the encoder, but it can be configured to be otherwise. (b) Example results for prediction ANAE after training a `StatePred` model on NACA0012 airfoil pressure data at varying angles of attack. The plots show prediction ANAE on training and validation data from epochs 100 to 1000, and final prediction ANAE on test data is noted at the top.

example of running `TrajPred` on data from a system exhibiting a polynomial manifold, described by the equations $\dot{x}_1 = \mu x_1$ and $\dot{x}_2 = \lambda \left(x_2 - x_1^2\right)$, with $\lambda < \mu < 0$. This has been studied previously in Brunton et al. (2016) and Lusch et al. (2018), and we use data from the latter. We trained a `TrajPred` model on $\sim$ 10000 trajectories; the results are in the `examples/` folder of the source code and have been omitted here for brevity considerations.

2.5. Hyperparameter search

Both the `StatePred` and `TrajPred` contain several arguments / hyperparameters that affect the architecture and training pipeline. A full list of these can be obtained from the API reference. Setting suitable values for these hyperparameters is important to achieving good performance. While there are techniques in literature such as Mendoza et al. (2018); Dey et al. (2020) that automatically search for the best model to use on given data, they may be challenging to adopt to DLKoopman. To ease this burden, we provide a hyperparameter search module in the DLKoopman package that is ready to be used for `StatePred` and `TrajPred`. The key idea is that the user can provide ranges for each hyperparameter to be swept over. Each set of hyperparameter values leads to a particular model; many such models are then run on the given data and the results compiled and ranked, from which the user can choose the best model(s).

The `examples/` folder in the source code contains tutorials for hyperparameter search. Here, we briefly run through an example. Suppose the user provides:

```
hyp_options = {
    'rank': [3,4,5,6], #4 options
    'encoded_size': 50, #1 option
    'encoder_hidden_layers': [[200,200],[300,200,100]], #2 options
    'numepochs': [200, 300, 400] # 3 options
}
```

This will result in a total of $4 \times 1 \times 2 \times 3 = 24$ sets of hyperparameter values, which will use different values of `rank`, `encoder_hidden_layers`, and `numepochs`. All other arguments will be set to the provided values (i.e. `encoded_size = 50`), or defaults if not provided. Hyperparameter search can then be run as:

```
run_hyp_search(
    dh = dh, # instance of either StatePredDataHandler or
    ↪  TrajPredDataHandler, depending on the problem
    hyp_options = hyp_options,
    numruns = 15, # optional
    sort_key = 'avg_pred_anae_va' # optional
)
```

Since a model can take time to run for large data sets, the optional argument `numruns` allows the user to specify how many runs they want. In this case, 15 models will be sampled from the total of 24 possibilities. The optional argument `sort_key` defines the metric to sort the final results by. In this case, the average prediction ANAE on validation data will be used to rank the models. We strongly recommend providing validation data via the data handler, as performance on validation data is a good indicator of the goodness of any model.

Finally, note that if the script is halted (such as the user stopping it forcibly, or other unforeseen interruptions), intermediate results will be available. This is particularly useful for long runs.

2.6. Miscellaneous notes

The speed of executing individual runs depends on a variety of factors such as the amount of training data, complexity of the autoencoder network, and the computing platform. DLKoopman uses Pytorch, and will run on GPUs by default if available. The user can change this default, as well as several other configuration options such as the choice to use exact eigenvectors vs projected eigenvectors for state prediction (Kutz et al. (2016)). These configuration options have been included to give freedom to the user – they are described under `dlkoopman.config` in the documentation.

3. Conclusion

We have presented DLKoopman – a software package for Koopman theory that uses a deep learning autoencoder to learn linear encodings of a system, along with learning the corresponding linear dynamics. It trains on data from snapshots of any system, and can perform both state prediction and trajectory prediction. The package is open-source and can be installed as a Python tool. It contains extensive documentation, examples, and a hyperparameter search module, and introduces a novel performance metric ANAE. We hope that DLKoopman benefits many and becomes widely used as a tool for data-driven analysis and behavioral predictions of dynamical systems. Future work will include adding features such as control inputs and additional losses.

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-20-C-0534. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA. Distribution Statement A, "Approved for Public Release, Distribution Unlimited."

The authors would like to thank Ethan Lew and Pachapakesan Shyamshankar for helpful technical discussions and insights, and Matt Le Beau for administrative help.

References

D. J. Alford-Lago, C. W. Curtis, A. T. Ihler, and O. Issan. Deep learning enhanced dynamic mode decomposition. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 32(3):033116, 2022. doi: 10.1063/5.0073893.

Omri Azencot, N. Benjamin Erichson, Vanessa Lin, and Michael Mahoney. Forecasting sequential data using consistent koopman autoencoders. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119, pages 475–485, 13–18 Jul 2020.

Steven L. Brunton, Bingni W. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Koopman invariant subspaces and finite linear representations of nonlinear dynamical systems for control. *PLOS ONE*, 11(2):1–19, 02 2016. doi: 10.1371/journal.pone.0150171.

Marko Budisic. Koopman mode decomposition. https://github.com/mbudisic/koopman, 2017.

Kathleen Champion, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116 (45):22445–22451, 2019. doi: 10.1073/pnas.1906995116.

PyTorch Contributors. torch.linalg.eig — pytorch 1.13 documentation, 2022a. URL https://pytorch.org/docs/stable/generated/torch.linalg.eig.html.

PyTorch Contributors. torch.linalg.svd — pytorch 1.13 documentation, 2022b. URL https://pytorch.org/docs/stable/generated/torch.linalg.svd.html.

Chris C. Critzos, Harry H. Heyson, and jr Boswinkle, Robert W. Aerodynamic characteristics of naca 0012 airfoil section at angles of attack from 0 deg to 180 deg. Technical report, National Aeronautics and Space Administration, 1955.

Steven Dahdah and James Richard Forbes. decargroup/pykoop. https://github.com/decargroup/pykoop, 2022.

Nicola Demo, Marco Tezzele, and Gianluigi Rozza. PyDMD: Python Dynamic Mode Decomposition. *The Journal of Open Source Software*, 3(22):530, 2018. doi: https://doi.org/10.21105/joss.00530.

Sourya Dey. Dynamic mode decomposition and koopman theory. *arXiv preprint arXiv:2211.07561*, 2022.

Sourya Dey. DLKoopman. https://pypi.org/project/dlkoopman/, 2023.

Sourya Dey, Saikrishna C. Kanala, Keith M. Chugg, and Peter A. Beerel. Deep-n-cheap: An automated search framework for low complexity deep learning. In *Proceedings of the 12th Asian Conference on Machine Learning (ACML)*, volume 129, pages 273–288. Proceedings of Machine Learning Research (PMLR), Nov 2020.

Nicholas Geneva and Nicholas Zabaras. Transformers for modeling physical systems. *Neural Networks*, 146:272–289, 2022. doi: 10.1016/j.neunet.2021.11.022.

Eurika Kaiser and Brian de Silva. PyKoopman. https://pypi.org/project/pykoopman/, 2020.

B. O. Koopman. Hamiltonian systems and transformation in hilbert space. *Proceedings of the National Academy of Sciences*, 17(5):315–318, 1931. doi: 10.1073/pnas.17.5.315.

J. Nathan Kutz, Steven L. Brunton, Bingni W. Brunton, and Joshua L. Proctor. *Dynamic Mode Decomposition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2016. doi: 10.1137/1.9781611974508. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611974508.

Ethan James Lew. AutoKoopman. https://pypi.org/project/autokoopman/, 2023.

Qianxiao Li, Felix Dietrich, Erik M. Bollt, and Ioannis G. Kevrekidis. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10), 2017. doi: 10.1063/1.4993854.

Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature Communications*, 9, 2018. doi: 10.1038/s41467-018-07210-0.

Andreas Mardt, Luca Pasquali, Hao Wu, and Frank Noé. VAMPnets for deep learning of molecular kinetics. *Nature Communications*, 9, 2018. doi: 10.1038/s41467-017-02388-1.

Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, Matthias Urban, Michael Burkart, Max Dippel, Marius Lindauer, and Frank Hutter. Towards automatically-tuned deep neural networks. In *AutoML: Methods, Sytems, Challenges*, chapter 7, pages 141–156. Springer, Dec 2018.

Samuel E. Otto and Clarence W. Rowley. Linearly recurrent autoencoder networks for learning dynamics. *SIAM Journal on Applied Dynamical Systems*, 18(1):558–593, 2019. doi: 10.1137/18M1177846.

Naoya Takeishi, Yoshinobu Kawahara, and Takehisa Yairi. Learning koopman invariant subspaces for dynamic mode decomposition. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1130—-1140, 2017.

Jonathan H. Tu, Clarence W. Rowley, Dirk M. Luchtenburg, Steven L. Brunton, and J. Nathan Kutz. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014.

Goutham Voodarla. Analysis of naca0012 airfoil for different angle of attacks. https://skill-lync.com/student-projects/analysis-of-naca0012-airfoil-for-different-angle-of-attacks, Aug 2021.

Christoph Wehmeyer and Frank Noé. Time-lagged autoencoders: Deep learning of slow collective variables for molecular kinetics. *The Journal of Chemical Physics*, 148(24):241703, 2018. doi: 10.1063/1.5011399.

Matthew O. Williams, Ioannis G. Kevrekidis, and Clarence W. Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, 2015.

Enoch H. Yeung, Soumya Kundu, and Nathan O. Hodas. Learning deep neural network representations for koopman operators of nonlinear dynamical systems. *arXiv preprint arXiv:1708.06850*, 2017.