Koopman Autoencoder for Videos

Armaun Sanayei
Stanford University
armaun@stanford.edu

Abstract

Koopman Operator Theory is a theory used in computational physics for decomposing high-dimensional periodic data into an embedding space in which dynamics can be progressed linearly despite the original system dynamics being non-linear. The Koopman Operator is approximated through the matrix method, Dynamic Mode Decomposition. Since the method is linear, it does not work well with abstract dynamics in video data. In this project, I use ResNet, image segmentation, and autoencoder NNs to build a Koopman Autoencoder that embeds frames of the video and progresses them in the autoencoder embedding space to perform video prediction and interpolation.

1. Introduction

Video prediction is important in the context of compressing and predicting/filling in incomplete videos. In this project, I use techniques from computational physics and dynamical systems to encode videos in continuous (time dependent) encodings and extrapolate the encodings and decode them to extrapolate the actual dynamics in the video.

Current computational physics approaches applied to time series data rely on linear transformations to embed the dynamics in a lower dimensional space. This method is called Dynamic Mode Decomposition [6]. The method works very well for simple 1D time-series data and even 2D fluid flow data; however, it would be difficult to apply such a framework to full videos.

Videos do not have simple dynamics that can be decomposed linearly. For example, in computational physics time series data, linear combinations of variables can have noticeable patterns, but such dynamics are significantly abstracted in the pixel format of videos.

The main contribution this project makes is using an autoencoder based on ResNet and MASK CNN (image segmentation) to first encode the videos and use the encodings in a Koopman operator. Moreover, we train both operators (the autoencoder and the Koopman operator) together.

The goal and the final result were that the image segmentation played a crucial role in converting the video dynamics into more usable variables which can be progressed easily through a Koopman operator.

2. Related Work

2.1. Dynamic Mode Decomposition

Dynamic mode decomposition [6] is a linear matrix method used for decomposing time-series data. It is based on the foundation of Koopman Operator Theory, which holds that all non-linear systems can be mapped to a linear space in which an infinite dimensional linear map/functor can continuously be applied on the data to advance the system in time. This map is called the Koopman Operator. Dynamic Mode Decomposition effectively approximates the Koopman Operator finitely using a matrix **K,** such that:

$$\mathbf{X'=KX}$$

where X' is the data ordered such that

$$X' = [\vec{x}_2, \dots \vec{x_n}] \text{ and } X = [\vec{x}_1, \dots, \vec{x}_{n-1}].$$

Finding the K matrix is very useful for predictions, given any observation at time *t,* we can always predict time *t+m* by applying K to the m^{th} power. However, such an inversion is almost always impossible with high dimensional data (finding X inverse is nearly impossible when it has hundreds of thousands of dimensions). Therefore, PCA/SVD is used to decrease the dimension of the model. Therefore, at its heart, DMD is a lower-order-modeling technique that uses PCA features of a system to understand its dynamics. This method has been and is being used extensively to analyze time series data in computational physics and meteorology due to its simplicity and the few assumptions it makes. It will be the baseline method we use for comparison in this project.

It is important to note that DMD provides very useful information through the eigenvalues of the K matrix. The spectral decomposition of K is often complex, and the eigenvalues have unique interpretations. For example, given an eigenvalue (usually visualized on a unit circle Figure 4):

$$\lambda = \mu + i\omega \longrightarrow e^{\lambda} = e^{\mu}(\cos(\omega) + i\sin(\omega))$$

Now, this shows that the different features that DMD extracts have an associated frequency omega and an exponential growth/decay term associated with mu, the real component. This shows why DMD has been so powerful in computational physics, many natural systems can be decomposed into a combination of oscillation and exponential growth and decay.

2.2. Koopman Auto Encoder

The Koopman Auto Encoder [1] is a NN autoencoder that utilizes DMD. This autoencoder is the inspiration for my approach and was made in the context of analyzing physics data. Specifically, in the case of fluid flow data, certain flows are stored in such a way (e.g. pictures, vector graphs, moving camera, etc.) such that each data point is hard to interpret on its own or linearly. In these cases, it is useful to mimic what PCA does in DMD (i.e. decrease the dimension), but instead of using linear PCA, you expand the possibilities by using a non-linear autoencoder.

Two papers have made such an autoencoder ([1] and [2]). In both instances, the autoencoders were small and were designed specifically for certain types of fluid flow or pendulum data (where the data is already significantly distilled). As one will see in the following sections, the autoencoders from these papers fail when it comes to video data. Specifically, [1] cannot accept images since it is configured specifically for one dimensional data, and [2] has too small of an autoencoder to begin to decompose the images (it is used for distilled fluid flow data, given velocity, acceleration, vorticity, at grid points in time). These encoders cannot handle image segmentation or classification or object detection, which are necessary as a foundation for understanding video dynamics.

My goal is to make a larger autoencoder that uses ResNet and Mask CNN (image segmentation) to use DMD on video data.

2.3. ResNet and MASK CNN

As has been discussed in CS231N lecture, ResNet [4] and MASK CNN [5] are both important models in the history of NNs and are very powerful given their broad training. In this project, I used ResNets and Image Segmentation in parallel in the autoencoder to decompose the video data.

As discussed in class, I performed transfer learning, simply removing the final linear layer in both models and adding my own, which I trained along with rest of the encoder and autoencoder.

3. Data

The data used in this paper is partly generated (noiseless), partly physics data (noisy), and finally video data (significant noise).

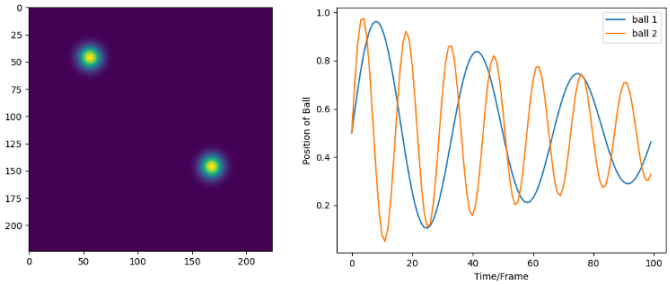


Figure 1: (Left) Image of two balls bouncing, (Right) Amplitude and frequency of two balls over the frames.

3.1. Bouncing Balls

The toy model used throughout for the analysis was a generated video of two balls bouncing in the air. This served as the perfect toy model for dynamics since there were multiple objects to keep track of, both had independent dynamics (different frequency and amplitude), and proper dynamics could be easily verified.

3.2. Fluid Dynamics

I also used Fluid Dynamics data:

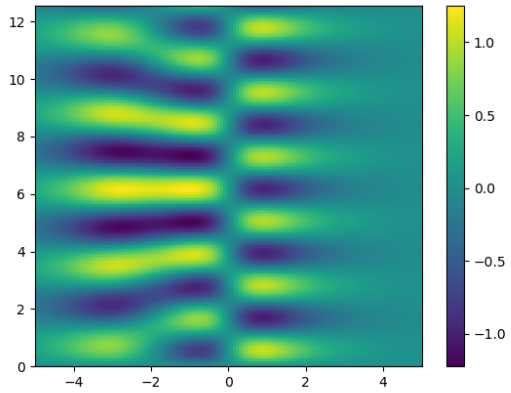


Figure 2: Example Snapshot Fluid Dynamics Data

Finally, for the main dataset I used a subset of the Moments in Time Dataset (MITD) from MIT [3]. The original dataset is 275 GB with millions of 3 second videos at high resolution. Instead of using the entire dataset, I used 50 videos, 60 frames per second, and with a resolution of 224 by 224. I used the resolution of 224 by 224 since that is the minimum resolution for optimal ResNet performance [4]. As we will see later, the primary videos that performed best were not of people but of things moving in a slow pattern (e.g. a candle burning, a car passing by a street, flowers swaying in the fields). I picked these videos, other videos of people/animals moving failed for my method but also much worse in the DMD method on its own, such large losses were not

useful for determining the benefits/drawbacks of any of the methods.

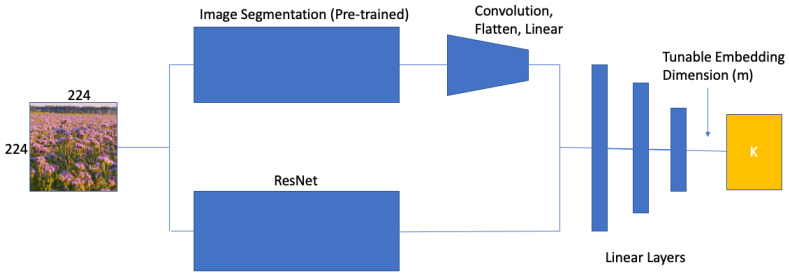


Figure 3: Neural Network Architecture

4. Methods

4.1. Neural Network Pipeline/Architecture

The neural network architecture can be split up into three key parts, the encoder, the Koopman operator, the decoder.

The encoder (non-pre-trained components):

```
(lin1_res): Linear(in_features=512, out_features=512, bias=True)
(lin2_res): Linear(in_features=512, out_features=256, bias=True)
(conv1_seg): Conv2d(21, 10, kernel_size=(4, 4), stride=(2, 2), padding=(2, 2))
(maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
(conv2_seg): Conv2d(10, 5, kernel_size=(4, 4), stride=(2, 2), padding=(2, 2))
(maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(flat): Flatten(start_dim=1, end_dim=-1)
(lin1_seg): Linear(in_features=980, out_features=512, bias=True)
(lin2_seg): Linear(in_features=512, out_features=256, bias=True)
(lin1_com): Linear(in_features=512, out_features=512, bias=True)
(lin2_com): Linear(in_features=512, out_features=256, bias=True)
(lin3_com): Linear(in_features=256, out_features=256, bias=True)
```

The encoder ran ResNet and Image Segmentation in parallel. ResNet (with the last layer removed) gives an output of 512 dimensions, which was put through linear layer. The image segmentation gives a 214 by 214 segmented image back. We run this through convolution and max pool layers to get a 16 by 16 by 4 image, which we then flatten and run through a few linear layers. Finally, we concatenate this 1D output with the ResNet output and put it through a few more linear layers until we send it into the Koopman operator.

Before we talk about the Koopman operator, let us discuss the decoder. The decoder components are:

```
(dfc3): Linear(in_features=256, out_features=4096, bias=True)
(dfc2): Linear(in_features=4096, out_features=4096, bias=True)
(dfc1): Linear(in_features=4096, out_features=9216, bias=True)
(unflat): Unflatten(dim=1, unflattened_size=(256, 6, 6))
(upsample1): Upsample(scale_factor=2.0, mode=nearest)
(dconv5): ConvTranspose2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
(dconv4): ConvTranspose2d(256, 384, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
(dconv3): ConvTranspose2d(384, 192, kernel_size=(3, 3), stride=(2, 2), padding=(3, 3))
(dconv2): ConvTranspose2d(192, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(dconv1): ConvTranspose2d(64, 3, kernel_size=(12, 12), stride=(4, 4))
```

The decoder is simply a combination of linear layers and deconvolutional layers that return a 224 by 224 color image.

The Koopman operator is a segment of the neural network that is independent of the encoder and decoder. As discussed in the introduction, the Koopman operator is a matrix with a very specific structure. We will discuss it in the next section.

4.2. Koopman Operator

Given an input of one image, the autoencoder outputs a list of T images, where T is determined by the user (in our case T=20). T represents how many time steps ahead we the neural network to predict.

To give an example of the pipeline, let us set T=3 and let us ssume the image that we put into the neural net was $\mathbf{I_t}$, where t is to indicate that it is image number t in a sequence of images. In this case the autoencoder will encode the image $\mathbf{I_t}$ into a vector $\mathbf{x_t}$, we then will run $\mathbf{x_t}$ through the Koopman operator section of the the neural network three times. In other words, we will get:

$$x_{t+1} = Kx_t$$

$$x_{t+2} = Kx_{t+1} = K^2x_t$$

$$x_{t+3} = Kx_{t+2} = K^3x_t$$

Now, we will store each of these vectors $\mathbf{x_{t+1}}$, $\mathbf{x_{t+2}}$, $\mathbf{x_{t+3}}$ and run all these vectors through the decoder to get three different images, each of which is a prediction of the image at t+1, t+2, and t+3 respectively. We will then compare our true images with these predictions and fix the weights of the matrix K.

To simplify finding such a K, we make two key assumptions. First, we assume that we will be dealing with a diagonalized version of K (since the linear layers in the autoencoder and decoder effectively act as eigenvector projection matrices). Note, however, this does not apply all the dynamics are linear, since we never assumed K had real eigenvalues. In fact, all/most of the important eigenvalues of K will be complex (this gives the interesting dynamics).

The second assumption is that we assume we are dealing with the exponentiation of the diagonalized Koopman operator. In other words, it is a neat trick to save us from multiplying the Koopman operator matrix so many times by itself. We notice that if we exponentiate the Koopman operator, then we get that exponential multiplication turns into simply addition in the exponent. Therefore, instead of dealing with the matrix K, we deal with its $\mathbf{e^K}$, its exponential. This does come with some issues. Like we discussed at the very beginning, the eigenvalue of K are complex, so when we exponential complex variables, we end up getting

$$\lambda = \mu + i\omega \longrightarrow e^{\lambda} = e^{\mu}(\cos(\omega) + i\sin(\omega))$$

Which is the form we saw earlier. In order to deal with this, we double the dimension of the K matrix (e.g. if it is usually 250 x 250, we encode it as 500 x 500). The extra factor of dimensions represents the complex components. In other words, when we do matrix multiplication with the K matrix, we assume we are multiplying pairs of real

numbers that combine together to make one complex number. Therefore, our final K matrix has Jordan Normal Form with Block matrices of the form:

$$B(\mu,\omega) = \exp(\mu\Delta t)\begin{bmatrix}\cos(\omega\Delta t) & -\sin(\omega\Delta t)\\ \sin(\omega\Delta t) & \cos(\omega\Delta t)\end{bmatrix}$$

Therefore, the parameters that we have to learn are the omega_i and mu_i for each block matrix. In the PyTorch interface, we add these variables as parameters in our model, and thus they are updated with the loss function. One key observation is the multiple of delta t in the block matrix, the multiple illustrates how the exponential converts multiplication to addition in the exponent (instead of exponentiating the Koopman operator multiple times, we simply change the delta t, which in this case is the difference in index of the video frames).

4.3. Loss

The loss function is unique in this scenario since we are predicting many time steps in advance. Often, with an autoencoder, the goal is for the decoder to decode the original image that we encoded, but in this case the goal is to decode future images at a specified time step away.

The loss function has three components. The first component is:

$$\mathcal{L}_{\text{recon}} = \|\mathbf{x}_i - D(E(\mathbf{x}_i))\|_{\text{MSE}}$$

Where D is the decoder and A is the autoencoder (note that the Koopman part of the NN is not applied here). The second component is:

$$\mathcal{L}_{\text{pred}} = \frac{1}{T}\sum_{t=1}^{T}\|\mathbf{x}_{t+i} - D(K^m E(\mathbf{x}_i))\|_{\text{MSE}}$$

The final loss that I compute is:

$$\mathcal{L} = \alpha\mathcal{L}_{\text{recon}} + (1-\alpha)\mathcal{L}_{\text{pred}}$$

This alpha parameter effectively controls whether I am training the Koopman Operator part of the neural network or the autoencoder part of the neural network. An alpha value of 1 will lead me to a regular autoencoder and an alpha value of zero, will not do well. This is further discussed in the next section on training.

4.4. Neural Network Training

To train the neural network, I start off with an alpha value of zero for the first 10 epochs. An alpha value of zero ensures that I have a competent encoder and decoder. When I tried training without an alpha of zero (at the very beginning), I end up getting that the entire neural networks ends up simply predicting the average image. After 10 epochs, I gradually increase the alpha value up to ½. Furthermore, I only train 20 time steps in advance (otherwise, the memory cannot deal with 20 extra images on top of the batch size). It took 100 epochs to train with batch sizes of 50 images and each epoch being 100 iterations. I used an Adam optimizer with custom beta values.

4.5. DMD

To perform DMD, I use the general method outlined in [6]. This involves performing an SVD and pseudoinverse of the X matrices to end up getting an approximation for K. To predict future time steps, you simply apply K iteratively to your starting vector.

5. Experiments

In every case I tested, DMD performed worse than the autoencoder. Below, we will investigate specific cases and the associated losses.

5.1. Bouncing Balls – Toy Model

Both DMD and the Autoencoder performed well in this example with the same loss of 0.0008 with an alpha value of 0.5 (for the loss function). The reason they both performed the same is because the ResNet and image segmentation did not provide any benefits to the autoencoder (bouncing balls are already segmented and they do not contain any noticeable objects). Below are the top eigenvalues:

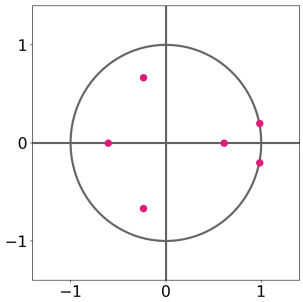


Figure 4: Eigenvalues plotted on complex plane with unit circle

As one can see the exponentials of the two balls were captured exactly (the exponential is in the y dimension, so any eigenvalue with a non-zero y dimension represents an exponential term). In this case, we had two different exponential terms with different frequencies, and both were captured. The reason that there are two of each is

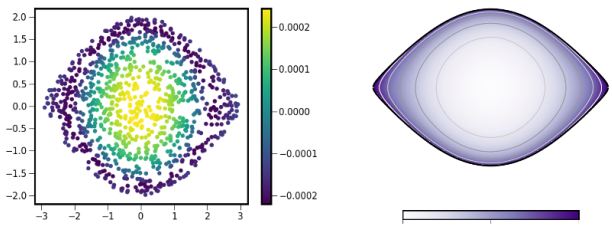


Figure 5: Eigenvalues plotted for fluid flow data

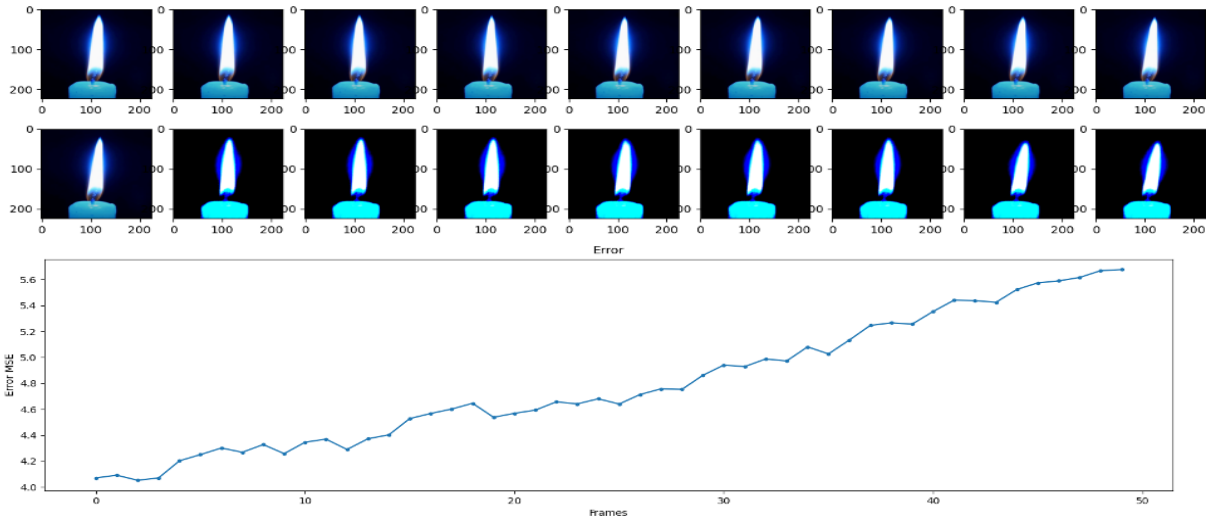


Figure 6: Example of Video Data. (TOP) True video (Middle) Prediction from autoencoder (Bottom) MSE Error

because every eigenvalue comes with a complex conjugate pair (hence the symmetry). There is one extra eigenvalue with a frequency but no exponential growth/decay. This probably indicates a background phenomenon.

5.2. Fluid Flow

The case for fluid flow gives a very good example to show case the power of these eigenvalues in smooth, periodic flow, as seen here we can extract out the attractor of the dynamics using the eigenvalues. This is seen in Figure 5. This just shows an illustrative example, more discussed [2], the main focus is on video data for this project.

5.3. Video Data

The video data from the Moments in Time Dataset that performed best was of single/groups of objects that had periodic motion. Example of the best types of video data are candles burning, flowers swaying, rivers running, cars passing by all with fixed camera and objects that move in a predictable fashion.

The NN did not perform well for moving humans/animals. In these cases the DMD performed much worse, not even producing a realistic image. In the case of our NN, it often produced blurred, average motion. Since both DMD and our NN performed bad in these cases, we will only show the cases that our NN performed well on; however, in every case no matter the video DMD performed worse than our NN.

The first example is candle data in Figure 6. As seen above, the candle is flickering/swaying/etc. These were all captured by the NN. The figure above was generated based on only giving the NN the first image and having it pre-train on the candle at other parts of the video. As one can see, the first image is an exact copy (this is a simple encoding/decoding task). All future images are predicted through running the Koopman operator. As seen, the error increases over time. This makes sense since the Koopman operator is being continuously applied every iteration, so the error is adding up.

There are two interesting observations from this error. Firstly, the error looks linear and not exponential. This is surprising, since given the form of the Koopman operator, I hypothesized that error over time would be exponential, but in this case is linear. The only way this is possible is if the major exponential terms in the Koopman operator are close to zero.

The second interesting phenomenon is that if one were to watch the predicted candle video on its own. One would notice that it looks completely realistic (flickering and swaying). This indicates that in some ways, the autoencoder is predicting realistic dynamics that looks very realistic and physical but is not what was seen. This is a positive if the goal is to create realistic looking video.

Finally, we can do a comparison of the candle data with the DMD loss, we see that it is:

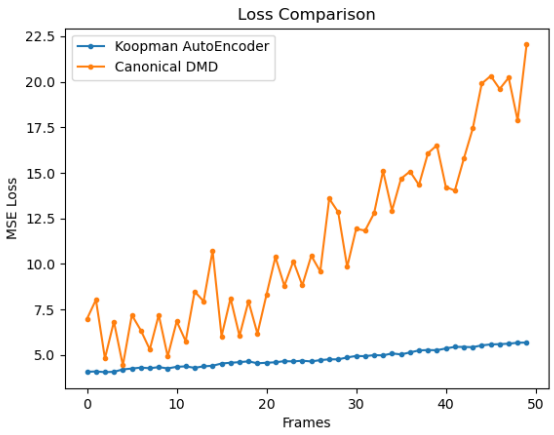


Figure 7: Loss of DMD and Autoencoder compared

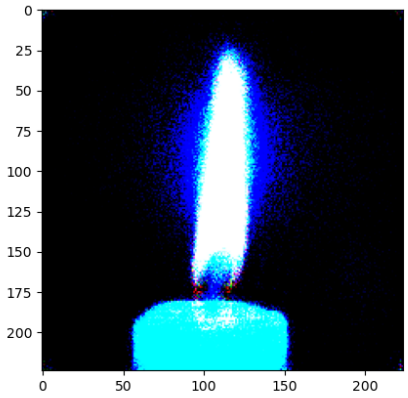


Figure 8: Example of saturation after many applications of Koopman operator

As one can see the DMD starts off predicting well, but this error grows significantly faster than the Koopman Autoencoder. The reasoning behind this (and confirmed through visually looking at DMD's predictions) is that DMD basically predicts the average of the flame for the first few iterations, then an exponential term blows up and leads to a completely white image since K is iteratively multiplied.

The unit circle plot of the candle video was simply hundreds of dots lining the x axis with only some slight perturbations in the y axis. This makes sense since the candle was not growing, everything was mostly an oscillation.

Another video example that performed well and had interesting exponential growth was one flower swaying in the wind.

In this video, there are flowers swaying in the wind. Interestingly, during the three second clip the wind picks up at the end, and some of the flowers begin oscillating

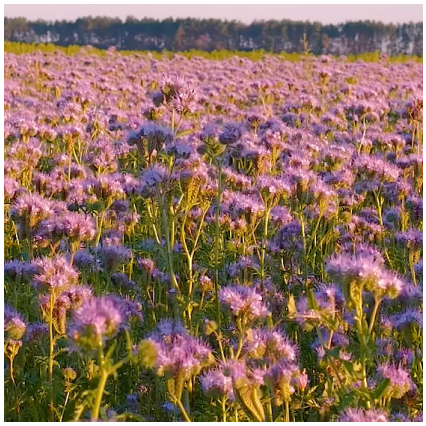


Figure 9: Snapshot of Flowers Video

faster. Two interesting eigenvalues plotted on the unit circle are shown in Figure 10.

As one can see, the very interesting part of this eigenvalue plot is that two of the eigenvalues are outside of the unit circle. This indicates that there is some sort of

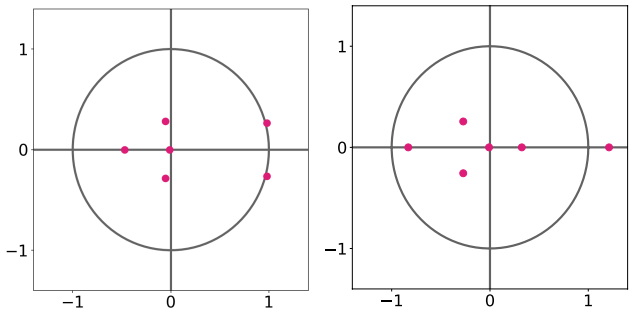


Figure 10: Koopman eigenvalues of Flowers Video

growth happening in the system. In this case, it was the wind. The fact that the Koopman Autoencoder was able to extract this information from a video of flowers in the wind was remarkable. In other words, the autoencoder was able to determine that energy was being added to the system (i.e. the wind picked up). This could have many potential applications.

Some other interesting findings/results was from car passing data. In some videos, there is a fixed camera on the side of the road and a car passing by. Interestingly, the Koopman autoencoder was able to predict the moving of cars across the screen at the correct speed; however, the

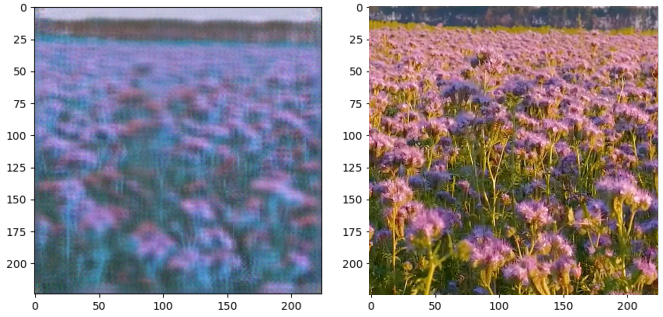


Figure 10: (Left) DMD Outputs an average version (Right) Koopman AE with swaying dynamics

Figure 7: Loss of DMD and Autoenoder comared

predicted images (instead of having the original car moving across) had a shadow of the car (i.e., a black segmented blob) moving across the screen. In these cases, it seemed like the decoder was underperforming. It was clear that the dynamics were correctly predicted by the Koopman operator, but the decoder did not have enough power to reconstruct the car. In some ways, this is reasonable given the fact that each car looked different and was hard to reconstruct from a stereo perspective (since the angle of the car was changing).

6. Conclusion

The Koopman Autoencoder clearly performed well compared to the DMD approach. The ability to segment and classify objects in an image and the non-linear transformation of an autoencoder are immensely powerful and useful in building a Koopman decomposition. The unique application of Koopman technique, usually for simple 1D physics data, to 3D (color) image data was a big step and proved to work better than expected.

It primarily worked well on videos with consistent, patterned motion and it did not work too well for action-packed images with multiple objects (like people) doing unpredictable things.

These findings give us both potential applications and future directions/investigations.

Given the ability of the Koopman operator to predict very well background objects moving in a general pattern. It could be very useful for generating and filling in background images in videos, where the main actor (e.g. a person or an object) is doing complicated activities in the foreground while the background remains relatively stable in an oscillatory pattern (e.g. a waterfall, flowers in a field, changing clouds, etc.). The Koopman autoencoder can predict these things many time steps (up to 20 reliably) in advance; moreover, even if the predictions are not exactly truthful to the physics, they do look incredibly realistic (as was the case with the candle data).

Some future directions to investigate are potentially seeing if a larger decoder could potentially help in predicting more complex behaviors and capturing more complex/changing objects. It seemed that the size of the decoder was a bottleneck in our study since the dynamics were predicted well but the objects in the image were not rendered faithfully.

Finally, another avenue of research is to do a side-by-side comparison of the Koopman operation and a RNN. Both functions have the same input and output and have an iterative nature, however, the Koopman operator has much richer dynamics (complex dynamics), which makes it harder to train but also more capable when it comes to predicting data.

References

[1] O. Azencot, N. B. Erichson, V. Lin, and M. Mahoney. Forecasting sequential data using consistent koopman autoencoders. In International Conference on Machine Learning, pages 475–485. PMLR, 2020.

[2] S. L. Brunton, M. Budiˇsi´c, E. Kaiser, and J. N. Kutz. Modern koopman theory for dynamical systems. arXiv preprint arXiv:2102.12086, 2021

[3] M. Monfort, A. Andonian, B. Zhou, K. Ramakrishnan, S. A. Bargal, T. Yan, L. Brown, Q. Fan, D. Gutfruend, C. Vondrick,
et al. Moments in time dataset: one million videos for event understanding. IEEE Transactions on Pattern Analysis and Machine Intelligence, pages 1–8, 2019

[4] . He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference
on computer vision and pattern recognition, pages 770–778, 2016.

[5] . Minaee, Y. Y. Boykov, F. Porikli, A. J. Plaza, N. Kehtarnavaz, and D. Terzopoulos. Image segmentation using deep learning: A survey. IEEE transactions on pattern analysis and machine intelligence, 2021.

[6] P. J. Schmid. Dynamic mode decomposition of numerical and experimental data. Journal of fluid mechanics, 656:5–28,2010.

SUPPLEMENTARY NOTES:

Supplementary code was submitted to Gradescope. The autoencoder and all the version I tried are in autoencoder.py and the trainer.py contains the training mechanism.

The autoencoder.py contains at least 5 other autoencoders that failed and did not make the cut for this paper. They all tried different methods to build a better Koopman autoencoder.

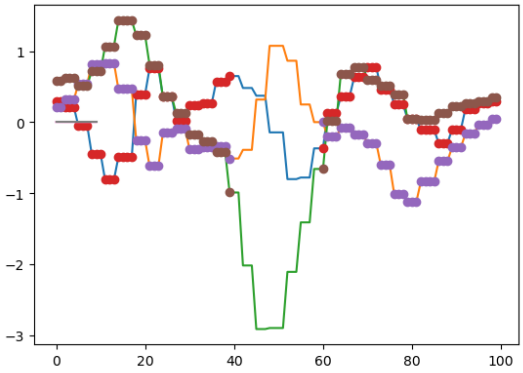
Some examples of other things I attempted:

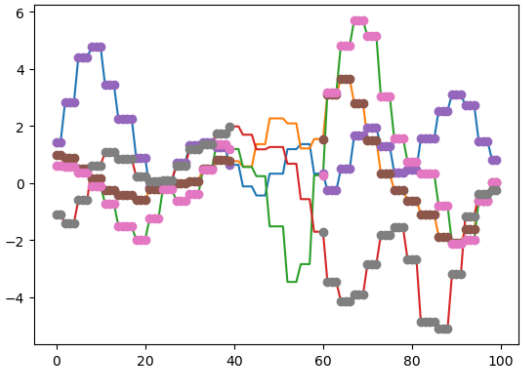
1. Original Idea: Have only an autoencoder and perform interpolation on the encoding space. The figures below indicate why that idea failed (i.e. the dynamics of encodings was very difficult to predict and interpolation was not working well).
2. Instead of having a Koopman Neural Network in the middle, perform matrix DMD in the middle of the network. This method failed because the DMD could not be trained in unison with the neural network and because running a single forward loop took a very long time (computing inverse of matrix takes a long time).

3. Instead of having an autoencoder, just have a decoder that decodes a set number of variables of the form (e^(d*t))*a*sin(b*t)+c where a,b, c, and d were learned. In this manner, we would be forcefully encoding these time-dependent continuous function. This did not train well and loss always remained high. It only local optimized when it predicted the average.

The code for all these trials is in autencoder.py. The corresponding classes are autoencoder, autoencoder_proj, and autoencoder_freq.

Also, some supplementary plots. These are some ideas that didn't work. Originally, I was hoping I could just interpolate in between the encodings of the autoencoder. This is what they look like in 3D for video data, these figures make it clear that it is almost impossible to interpolate some of these encodings:





Here is what an image of two balls bouncing looks in 3D encoding space over time: