AMATH 563 Final Project

Inferring Low-Dimensional Latent Structures with Recurrent Neural Networks

Zachary McNulty
zmcnulty, ID: 1636402

Abstract: Future state prediction is a fundamental challenge within applied mathematics. Success in sequential prediction tasks such as those found in natural language processing has relied heavily on the use of neural networks. Recent studies suggest this success is due to the emergence of low-dimensional representations of the latent space underlying the task at hand. This latent space provides an optimal coordinate system which best explains how the sequence evolves over time. While this latent space may be predictable in some tasks, it is reasonable to believe some kind of grammar forms the latent space of an NLP task, it is much more ambiguous in others. This paper aims to provide a framework for studying the latent space of natural scenes and movies learned in recurrent neural networks under a predictive setting.

All code used for this project can be found *here* on my *GitHub* page.

University of Washington
June 2019

Introduction and Overview

As the demand for high-quality future-state predictions grows, the use of neural networks will become more and more ubiquitous. Given an ample amount of data, neural networks can outperform practically any other technique in a model-free setting. By now, this success comes at no surprise. Large neural networks, sometimes consisting of millions of parameters and nonlinearities, map inputs into such a high-dimensional space that there likely exists some kind of simple representation within that space. Recent studies suggest some of this success can be attributed to the emergence of low-dimensional structures within the neural representation [1]. These studies have shown that neural networks are capable of extracting low-dimensional structures embedded in a high dimensional space inherent to the task at hand [1][2]. Often, these low-dimensional structures provide insights about the fundamental rank and relevant characteristics of the system at hand. In some tasks, the fundamental structure is fairly predictable: we might expect the structure learned in a natural language processing task to be some form of grammar. However, one task that does not have such an obvious structure is prediction within a natural movie scene. As humans, our visual system is remarkably good at building a representation of the world around us which fosters future state predictions, allowing us to navigate and interact with our environment with ease. There has been an immense amount of work done in the field of image processing trying to answer these very questions, but a lot of it has been done on stationary images or frame-by-frame. It is not clear whether the act of moving through the natural world and the dynamics of that motion has any effect on the fundamental structure of this visual system. The abstraction of recurrent neural networks under a predictive learning framework embeds temporal information within the task of building a representation for the visual space which might allow us to study these natural movies from an interesting perspective. The goal of this paper is to develop a framework for studying the representations neural networks build of video under this predictive task. The simple example represented in this paper aims to be a stepping-stone towards more complicated natural scenes. Previous work has found success using the paired convolutional and recurrent neural network framework that is used in this paper to predict these sorts of natural movies [3]. While this work shows some evidence of the encoding of latent variables in the recurrent neural networks, the paper does not explore these findings in full detail. Furthermore, they do not explore how the structure of the visual scene might be intertwined with the dynamics of motion.

Theoretical Background

Neural networks are a form of supervised machine learning that consist of a set of nodes, often arranged in collections called layers, and a set of weighted (one-way) connections between these nodes. Each node in the network takes on a certain activation level that depends on the weighted sum of the activations of all the nodes that have connections to it. Often, this sum is manipulated in some way by an activation function σ to generate the final activation. Below are a few common activation functions:

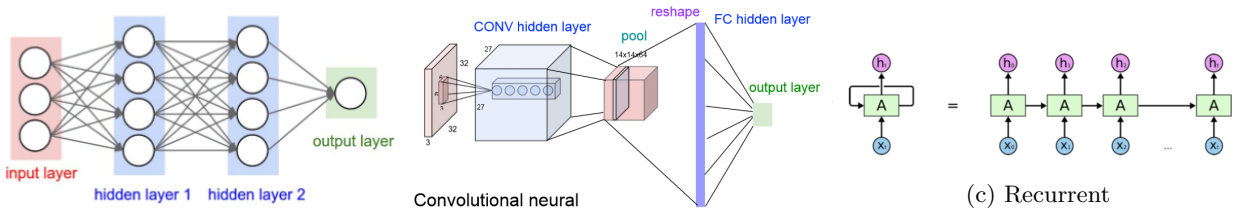
$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad sigmoid(x) = \frac{1}{1 + e^{-x}} \qquad RELU(x) = max(0, x) \qquad linear(x) = x$$

This can introduce nonlinearities into the system, improving its ability to capture nonlinearities in the data, and give desirable properties to the activations (i.e. the sigmoid activation function $\frac{1}{1+e^{-x}}$ squishes the activations into $[0, 1]$). Thus, the activation of a given node v_i is:

$$a_i = \sigma\left(\sum_j W_{ji} a_j + b_i\right) \tag{1}$$

Data is fed into the network through a set of nodes called the input layer and this defines the initial activations. From here, the way in which these activations, and hence information, is passed on varies between different types of neural networks. Three of the main classes of neural networks are Feedforward, Convolutional, and Recurrent neural networks, shown in Figure 1.

A feedforward neural net simply passes activations from the previous layer to the next one in a linear way. Typically, every node in a given layer has a connection to every node in the previous layer and these networks are called fully-connected. As such, these networks require a huge number of parameters, making them slow to train and easy to overfit. Nonlinear operations are introduced using the activation functions discussed above. This is important because many tasks require these nonlinearities.
For example, many classification tasks act on data that is not typically linearly separable and thus any linear classifier will perform poorly on its own.



(a) Feedforward (b) Convolutional (2D) (c) Recurrent

Figure 1: Three common architectures for neural networks. Images from UW CSE 446 course slides.

Convolutional neural networks (CNNs) were designed to capture local information. Rather than having each node connected to every node in the previous layer, the node only forms connections with a small region, sometimes called a **receptive field**, of nodes in the previous layer. The set of weights forming these connections is often called a **filter** and is shared across all nodes in the convolutional layer. Commonly, each convolutional layer has several of these filters per receptive field. The idea is that each filter searches for a specific local feature throughout the entire input and the receptive fields localize these features. This shared-weight, convolutional architecture vastly reduce the number of parameters in the network, making them quite appealing for training on large inputs. Often, convolutional layers are combined with max/min pooling layers. These layers also have receptive fields, but they condense the dimension of the input by simply outputting a single value for each receptive field: the max/min activation within that receptive field. In doing so, these layers emphasize the importance of only the existence of given features rather than the exact position they exist within the input. A common application is images, in which case the receptive fields form $2D$ windows that are scanned across the image.

Recurrent neural networks (RNNs) were designed to process organized sequences of input. As we see in Figure 1c, not only is the next input feed forward into the network, but the output of the previous input is fed back in. This allows the network to store a "memory" of the previous inputs in the sequence, better allowing it to see how the system evolves input to input in the sequence (e.g. over time).

Algorithm Implementation and Development

Constructing Input Set

For this paper, our initial input set consists of videos containing objects whose motion is dictated by a dynamical system. In the example shown in Figure 2 the object is subject to the dynamics of a spring-mass system along some axis through the origin. This will be the system we use throughout the paper. These videos embed the essentially 3-dimensional dynamical system, specified by the angle of oscillation, the object's distance from origin, and the object's velocity, in the high-dimensional pixel space of the image[1]. To generate a meaningful

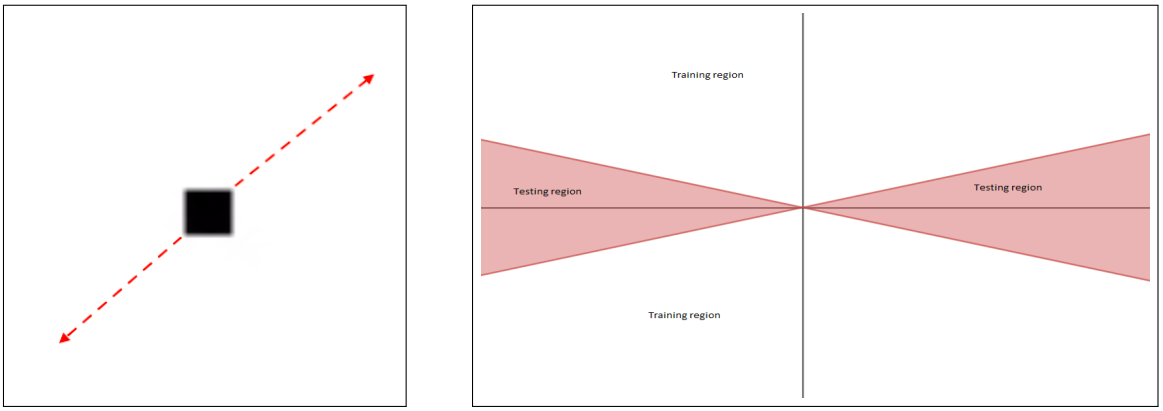


Figure 2: The left shows an example input: an object oscillating at an angle through the origin of the screen. The right gives a visual of how training and testing sets were generated. For training movies, an angle was randomly chosen in the white regions and a movie was made with the object oscillating through the origin along that angle. For the testing movies, an angle was chosen from the red region instead.

[1]There are many other ways we could have done this high-dimensional embedding, but this way is more appropriate for the future goals of the project which includes processing natural movies

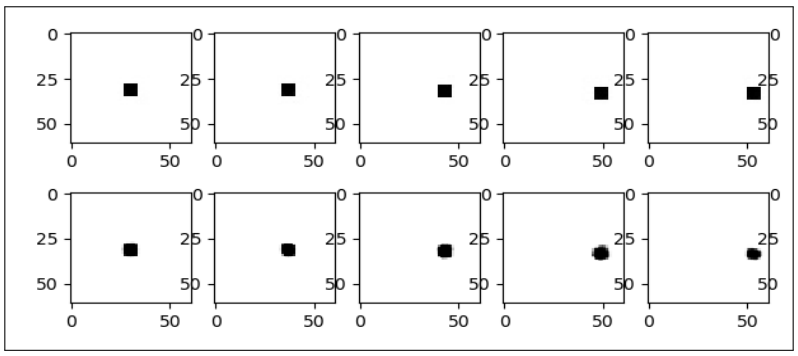


Figure 3: Autoencoding results for testing dataset. Top row shows the true initial state and bottom row shows the reconstructed state after running the image through the encoder and decoder. As we can see, the training success generalizes well.

testing dataset we omitted some possible angles of the axis from the training dataset. This is shown in Figure 2. Specifically, we set aside 20% of all possible angles for testing. This will allow us determine whether or not the learning properly generalizes.

Neural Network Architecture and Training

The network consists of three main components: an encoder, an RNN, and a decoder. The encoder consists of a series of convolutional and max pooling layers which reduce the 61 x 61 pixel image to a 4 x 4 x 4 condensed representation in feature space. Of course, this is still much higher dimensional than the underlying dynamical system. Nonetheless, it condenses many of the invariances in the videos such as the white background and the shape of the object. The decoder consists of a series of convolutional and upsampling layers which essentially unpack this condensed representation. The encoder and decoder are trained together in the absence of the RNN on an autoencoding task: given a specific frame, simply recreate it. Thus, together the encoder builds this condensed representation and the decoder extracts the original input from this condensed space, completely isolated from temporal information and the predictive task. It is then the job of the RNN to move the object within this condensed space. We used a simple, fully-connected RNN. As such, we flattened the 3-dimensional output of the encoder to feed it into the RNN. Afterwards, we reshape the output of the RNN back to 3D to feed into the decoder. For the training of the RNN, the encoder and decoder weights are held fixed. Then, the RNN is trained on a sequence prediction task. Given a sequence of frames in the video, the RNN must recreate a new sequence of frames of the video moved Δt frames ahead in time. In this paper, we chose $\Delta t = 5$, but this is of course arbitrary and its meaning depends on the frame rate of the videos. Roughly, this corresponds to about a quarter oscillation of the object as we can see in Figure 4. In both cases, the ADADELTA optimizer and binary crossentropy loss function (as the pixel values are normalized to [0,1]) were used to train the network weights and 20% of the training data was set aside for cross-validation. An L_1 penalty of 10^{-5} was added on all network weights to help encourage a sparser representation of the system. More details on specifics of the network can be found in Appendix B.

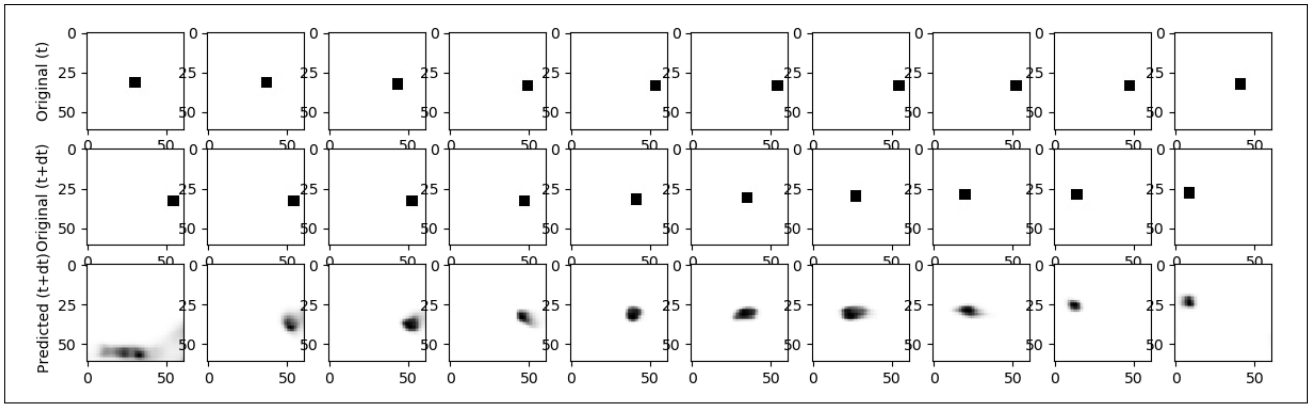


Figure 4: RNN prediction results for testing dataset. Top row shows the true initial state, middle row shows the true final state, and bottom row shows the predicted final state. As we can see, the training success generalizes well.

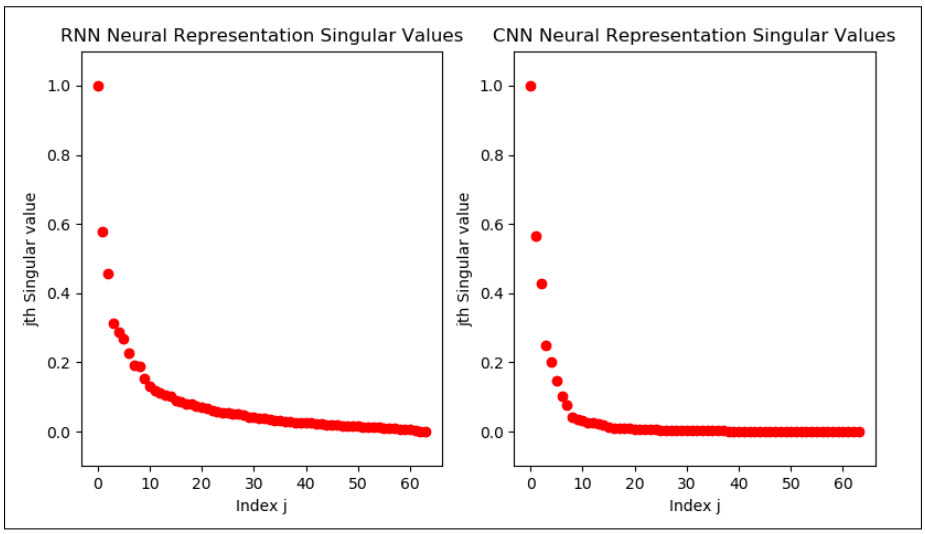


Figure 5: Singular values of RNN (left) and encoder (right) neural representations in response to all videos in training and testing datasets.

Analysis

Once the RNN and autoencoder are properly trained, we will extract the activations of neurons following decoding (the input to the RNN) and in the RNN itself to a series of test datasets. At this point, we are confident the system can accurately produce the expected output so the final prediction is not so important. Since the output of the encoder is 3-dimensional, we will vectorize its activations to study its neural representation. We will perform PCA on these (demeaned) activations, however this alone will likely not capture the true dimensionality of the system as the RNN and decoder likely store information in a nonlinear manner. However, results from [1] suggests these representations are likely stored on a low-dimensional, nonlinear manifold. If this manifold encodes information about the system (e.g. the axis of oscillation or initial direction of motion) we would expect videos that share these characteristics to be localized along this manifold. If this is the case, standard classification techniques should be able to reasonably separate videos varying in these characteristics within the RNN's neural representation. To test this, we generated two new datasets of images that had two disjoint groups of localized angles for their axis of oscillation. In one dataset, the axis of oscillation was roughly vertical and only the initial direction of motion, up versus down, varied between the two groups (angles of $70° - 120°$ and $250° - 290°$ respectively). In the other dataset, the axis of oscillation was roughly horizontal and only the initial direction of motion, left versus right, varied between the two groups (angles of $160° - 200°$ and $-20° - 20°$ respectively). In both the left/right and up/down case, we used simple linear SVM as our classifier. Our datapoints were the neural representations for each frame in the 80 different movies generated, labeling each frame either as 1 or 0 for up/down or left/right depending on which test set was used.

Next, we will see if any specific neurons within the RNN or encoder exhibit noticeable patterns over time, frame by frame, or in response to specific angles. In the former case, we would expect some sort of oscillatory behavior due to the dynamics of the object in the video, while in the latter case we might expect some neurons in the RNN to exhibit activity localized to only a specific subset of angles of oscillation. The latter might suggest these neurons are somehow encoding the angle of oscillation as a result of the prediction task. To do so, we will first simply plot each neuron's activations as a function of frame and angle. To further explore the time dynamics of the whole neuron population within the RNN and encoder, we will perform DMD on the set of all activations.

Computational Results

As we can see in Figure 3, the autoencoder performs incredibly well. Not only does it capture the appropriate location of the object, but its general shape as well. Furthermore, there are no errant patches of black anywhere in the background. This gives us confidence that the encoder and decoder regions of the neural network can adequately capture the invariances in the videos. In Figure 4 we see the RNN predicts the future state (5 frames ahead in the video) fairly well. It has some difficulties with the first frame but quickly improves. This is likely due to the fact that in sequence based learning you have more and more information to work off of the farther into the sequence you go. On the first frame, the only information the RNN has seen is the first frame so it is impossible to even determine which direction the object is moving in at that point. Once the

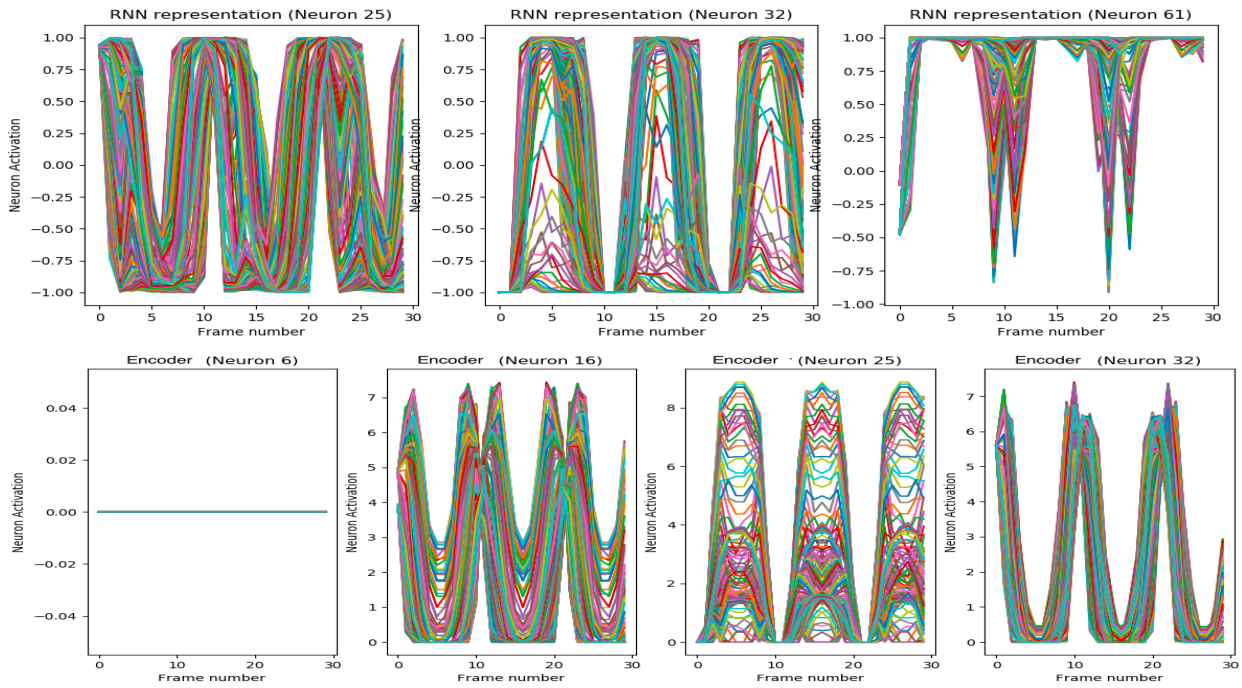


Figure 6: Neural activations of RNN (top) and encoder (bottom) over the duration of the movie. Each line corresponds to a single movie. In both the RNN and the encoder, we see clearly periodic behavior. Furthermore, neuron 25 in the RNN seems to exhibit almost sinuisoidal behavior.

autoencoder and the RNN were fully trained, we could extract the neural representations, the activations of all neurons in a given layer, in both the RNN and at the end of the encoder in response to our full dataset. Figure 5 shows the results of PCA on these neural representations. As expected, there does not seem to be a strong low-dimensional, linear structure to these neural representations. In the case of the RNN, the singular values are slow to decline and thus there are many seemingly relevant dimensions in a linear sense. The singular values for the encoder decay faster, suggesting it is lower dimensional in a linear sense, but there are still many more relevant dimensions than we would expect from this autoencoder which simply has to capture the (x, y) coordinates and the basic structure of the object.

Using the trained encoder and RNN, we extracted the neural representations for the left/right and up/down datasets discussed earlier in the analysis section. Under a linear SVM, the RNN neural representations could be classified with 95% accuracy, suggesting it is almost perfectly linearly separable. On the other hand, the encoder's representations could only be classified with 76 % accuracy. Since the encoder has no sense of time, it only sees a single frame at a time, we can use its accuracy as a baseline for comparison. The significantly higher accuracy of the RNN suggests the neural representations for each frame have information on the direction of motion embedded within them. Furthermore, as the encoder does not exhibit this high of accuracy, it seems this information is generated by the prediction task rather than simply being passed along from the encoder.

Next, we plotted the time-dynamics of these neural representations as well as the tuning curves for the activations as a function of the angle of oscillation. In Figure 6 we see a few examples of the most common time dynamics found within both the RNN and the encoder. It is difficult to interpret these with much certainty, but they might give some clues into some of the underlying structures of the system that could be grounds for future exploration. For example, in Figure 6 we see some of the neurons in the encoder never activate regardless of the frame number. This could be due to L_1 penalty we placed on all network weights. Since the stimulus is relatively simple to autoencode, this suggests the network could encode it with significantly fewer neurons. Furthermore, in both the RNN and the encoder we observe some sort of periodic behavior. In the RNN, neuron 25 seems to be exhibiting almost sinusoidal motion, suggesting it may play a role in encoding the simple harmonic motion of the oscillation. While some neurons in the encoder exhibit periodic motion as well, most of it is similar to that of neuron 16 with two peaks near the top of oscillation. Since the autoencoder only encodes spatial information, this might be caused by points near the edges of the oscillation where the object passes, reaches the extremum, then returns in the opposite direction shortly after. To see if there are any overarching trends in the time dynamics of these neural representations, specifically oscillatory behavior, we will find the DMD of these representations across all neurons. In Figure 7 we can see the DMD modes for the

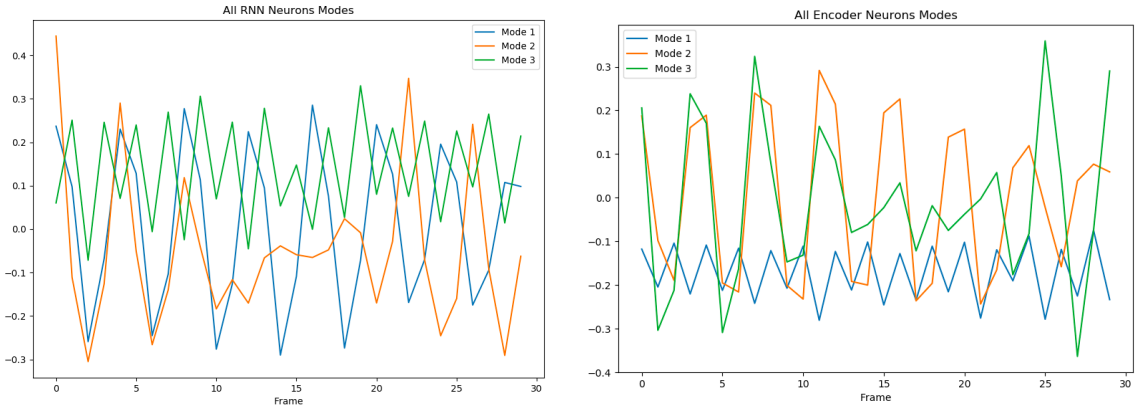


Figure 7: DMD modes for neuron time dynamics of RNN (left) and encoder (right) across all neurons. In both cases, we can see a mode that exhibits a clear periodic behavior.

time dynamics of neuron activation in both the RNN and encoder. This is across all neurons and all possible angles of oscillation. Mode 1 in the RNN and mode 2 in the encoder seem to exhibit clear oscillatory behavior, supporting our findings from Figure 6. However, the fact that the encoder also encodes this oscillation makes it difficult to determine what dynamics of the system are captured by the prediction framework rather than simply the autoencoding task.

In Figure 8 we see the tuning curves with respect to the angle of oscillation for several neurons. Again, the encoder has several neurons that appear not to react to anything. In the RNN for both neurons 14 and 58, we can see clear localization of the neurons activities to some subset of all possible angles. Interestingly, some neurons in the encoder such as neuron 44 exhibit this same kind of localization. This could be because the object can only pass through some regions in space if it takes a specific angle of oscillation. Again, the fact that both the encoder and RNN exhibit this localization makes it difficult to determine which aspects of the dynamics are captured by the prediction framework rather than simply the autoencoding task.

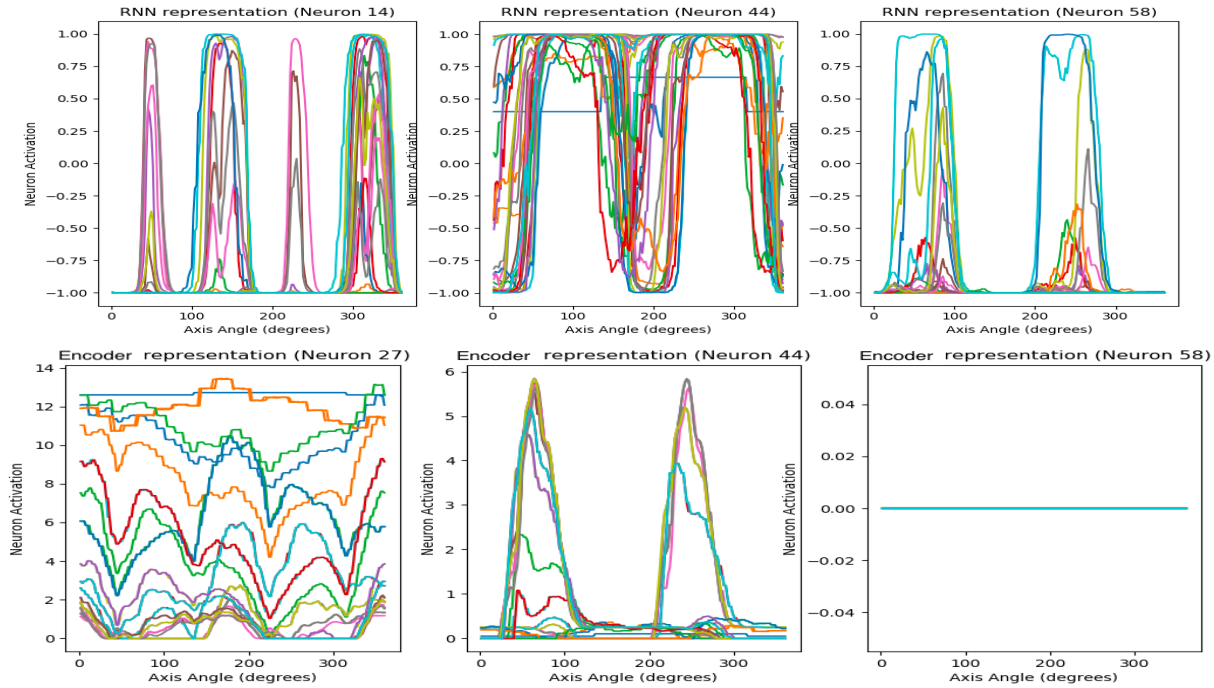


Figure 8: Neural activations of RNN (top) and encoder (bottom) over the duration of the movie. Each line corresponds to a specific frame number across all movies. Neurons 14 and 58 of the RNN and neuron 44 in the encoder show clear localization

Summary and Conclusions

While our SVM classification results provide some evidence that the RNN in the current model is storing information on the angle of oscillation, it is far from conclusive. Furthermore, this evidence is blurred by the fact that the encoder also shows some localization of activity to specific angles. Thus, it is not clear which roles the encoder and RNN play in storing information on the angle of oscillation and hence the dynamics of the system. One possible reason for this is that with the current setup, where all axes of oscillation are restricted to going through the origin, the position of the object can be enough information to extract the angle of oscillation: if in one frame the object is in the top right corner, it must be oscillating along the 45° line through the origin. In future studies we could rectify this by choosing the object to have a random starting point within the frame. In this new framework, the angle of oscillation would only be inferable through the sequence-based learning and prediction in the RNN. We would also like to further explore our findings that the angle of oscillation appears more localized in the RNN neural representation than the encoder's neural representation. One way we might go about doing this is plotting trajectories through the space defined by the first three principal components as we vary the angle of oscillation to see if there is any noticeable pattern. Nonetheless, there is still much work to be done before this framework can be applied to a natural movie.

References

[1] Stefano Recanatesi, Matthew Farrell, Guillaume Lajoie, Sophie Deneve, Mattia Rigotti, Eric Shea-Brown : Signatures and mechanisms of low-dimensional neural predictive manifolds (2018). bioRxiv 471987; doi: https://doi.org/10.1101/471987

[2] Ronen Basri, David Jacobs. Efficient Representation of Low-Dimensional Manifolds Using Deep Networks. ICLR (2017) https://openreview.net/pdf?id=BJ3filKll

[3] William Lotter, Gabriel Kreiman, David Cox. Deep Predictive Coding Networks For Video Prediction and Unsupervised Learning. ICLR (2017) https://arxiv.org/pdf/1605.08104.pdf

Appendix A

Below is a brief summary of the python libraries/functions I used during this project.

PyDMD

PyDMD is an python library that implements may common DMD algorithms and makes them simple to use and learn from. For more information, see their *documentation* or the corresponding *GitHub* page. We used PyDMD to determine some of the temporal dynamics in the neural representations.

Keras

Machine-Learning library that runs over Tensorflow (or Theano). Provides a framework to easily build, train, and test neural networks.

Sequential(): Create a neural network where all layers follow one after another (output of layer i fed directly to layer $i+1$ rather than combining multiple inputs from separate sources)

model.add(layer): Add the given layer to the neural network.

Dense(nodes): A fully-connected feedforward neural network layer. Specify the number of nodes, activation function, and any regularizers.

SimpleRNN(nodes): A fully-connected recurrent neural network layer. Specify the number of nodes, activation function, any regularizers, and whether or not you want the network to return sequences or just a single output (the final one).

model.compile(): Build the neural network and define the loss function and optimization routine.

model.fit(): Train the neural network and choose some training parameters.

model.predict(): Given a trained neural network, make a prediction given the specified input.

Appendix B

All code used for this project can be found at *here* on my *GitHub* page.

Neural Network Architecture

Figures 9, 10, and 11 below give a more detailed look at the network structure.

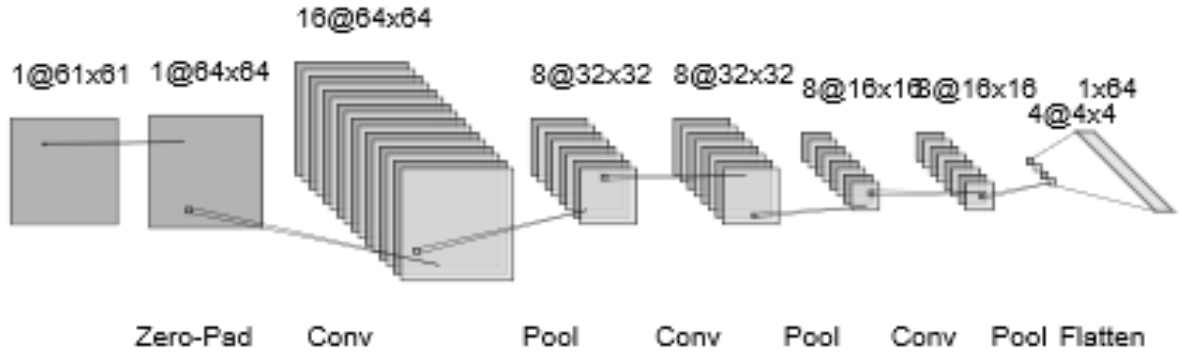


Figure 9: Neural network structure for encoder. The encoder consists of a sequence of 2D convolutional layers (kernel size 3×3) followed by max pooling layers (kernel size 2×2). When the encoding is complete, the neural representation is flattened so it can be fed into the fully-connected RNN. The decoder is just the reverse of this structure with pooling layers replaced by upsampling with identical kernel sizes.

```
Epoch 1/1
8750/8750 [==============================] - 64s 7ms/step - loss: 0.0069 - val_loss: 0.0100
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
zero_padding2d_1 (ZeroPaddin (None, 64, 64, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 16)        160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 8)         1160
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 8)         584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 8)           584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 4)           292
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 4)           148
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 8, 8, 4)           0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 8)           296
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 16, 8)         584
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 32, 32, 8)         0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 16)        1168
_________________________________________________________________
up_sampling2d_4 (UpSampling2 (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 64, 64, 1)         145
_________________________________________________________________
cropping2d_1 (Cropping2D)    (None, 61, 61, 1)         0
=================================================================
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
```

Max Pooling has (2 x 2) filter

Conv2D has (3 x 3) filter

Figure 10: Autoencoder Network Summary. Max pooling layers use a (2 x 2) window and pad with zeros. The initial Zero padding layer at the beginning pads the images with zeros. This makes the image dimensions divisible by 8 so that three rounds of max pooling followed by three rounds of upsampling do not alter the image shape. The final cropping2D layer removes the padded dimensions added by the zero padding layer. The shape of the convolutional layers above are of the form (None, height, width, number filters).

```
Epoch 1/1
250/250 [==============================] - 33s 133ms/step - loss: 0.0148 - val_loss: 0.0443
_________________________________________________________________________________________________
Layer (type)                                 Output Shape                                Param #
=================================================================================================
time_distributed_1 (TimeDistributed)         (None, 30, 64, 64, 1)                       0
_________________________________________________________________________________________________
time_distributed_2 (TimeDistributed)         (None, 30, 64, 64, 16)                      160
_________________________________________________________________________________________________
time_distributed_3 (TimeDistributed)         (None, 30, 32, 32, 16)                      0
_________________________________________________________________________________________________
time_distributed_4 (TimeDistributed)         (None, 30, 32, 32, 8)                       1160
_________________________________________________________________________________________________
time_distributed_5 (TimeDistributed)         (None, 30, 16, 16, 8)                       0
_________________________________________________________________________________________________
time_distributed_6 (TimeDistributed)         (None, 30, 16, 16, 8)                       584
_________________________________________________________________________________________________
time_distributed_7 (TimeDistributed)         (None, 30, 8, 8, 8)                         0
_________________________________________________________________________________________________
time_distributed_8 (TimeDistributed)         (None, 30, 8, 8, 8)                         584
_________________________________________________________________________________________________
time_distributed_9 (TimeDistributed)         (None, 30, 4, 4, 8)                         0
_________________________________________________________________________________________________
time_distributed_10 (TimeDistributed)        (None, 30, 4, 4, 4)                         292
_________________________________________________________________________________________________
time_distributed_11 (TimeDistributed)        (None, 30, 64)                              0
_________________________________________________________________________________________________
rnn (SimpleRNN)                              (None, 30, 64)                              8256
_________________________________________________________________________________________________
time_distributed_12 (TimeDistributed)        (None, 30, 4, 4, 4)                         0
_________________________________________________________________________________________________
time_distributed_13 (TimeDistributed)        (None, 30, 4, 4, 4)                         148
_________________________________________________________________________________________________
time_distributed_14 (TimeDistributed)        (None, 30, 8, 8, 4)                         0
_________________________________________________________________________________________________
time_distributed_15 (TimeDistributed)        (None, 30, 8, 8, 8)                         296
_________________________________________________________________________________________________
time_distributed_16 (TimeDistributed)        (None, 30, 16, 16, 8)                       0
_________________________________________________________________________________________________
time_distributed_17 (TimeDistributed)        (None, 30, 16, 16, 8)                       584
_________________________________________________________________________________________________
time_distributed_18 (TimeDistributed)        (None, 30, 32, 32, 8)                       0
_________________________________________________________________________________________________
time_distributed_19 (TimeDistributed)        (None, 30, 32, 32, 16)                      1168
_________________________________________________________________________________________________
time_distributed_20 (TimeDistributed)        (None, 30, 64, 64, 16)                      0
_________________________________________________________________________________________________
time_distributed_21 (TimeDistributed)        (None, 30, 64, 64, 1)                       145
_________________________________________________________________________________________________
time_distributed_22 (TimeDistributed)        (None, 30, 61, 61, 1)                       0
=================================================================================================
Total params: 13,377
Trainable params: 8,256
Non-trainable params: 5,121
```

Figure 11: RNN Network Summary. All layers besides RNN and the layers immediately above/below are the same as in the Autoencoder above. The layer above the RNN flattens the 3D convolutional output so it can be fed back into the RNN and the layer below it reshapes the RNN output to 3D so it can be fed back into the decoder. The TimeDistributed wrapper just allows these other layers to be applied to sequences. This allows us to feed sequences into the RNN and output sequences. The RNN layer is fully connected with 64 neurons.