Video Prediction

Independent Work Report (MAE 340, Spring 2021)
Advisor: Professor Olga Russakovsky

Matthew Coleman

April 26, 2022

This project represents my own work,
in accordance with the University regulations.
/s/Matthew Coleman

Contents

1 Introduction

While humans cannot perfectly predict the future, they are indeed capable of inferring a great deal of information about near events in the future, and this knowledge greatly aids them in planning out their actions, such as which movements to take to reach a goal. This ability to forecast the future is a direct result of an understanding of causality that is learned through observation and interaction [3].

A great amount of human predictions are erroneous in major respects, but even the humans least adept at inferring far-off outcomes and consequences still are masters of learning very near-term ones. For example, humans have a good sense for where a car will move in the street, or which direction a pedestrian may continue walking. Even a young child can predict where to toss a football to a moving receiver, and even this small knowledge reveals an infinite wisdom compared to the most advanced video prediction methods.

The task of video prediction is comprised of several open challenges in computer vision; it uses some of the most recent model architectures that have been developed and it even contends directly with an impossible task altogether, which is to predict the future. Although it is a particularly confusing task, it also has the potential for immense impact and immediate practical applications, such as in autonomous driving [6], video interpolation [8] and most interesting in the context of this report, robotic control systems [5].

This project will examine the current state-of-the-art in video prediction machine learning models, report on several experiments carried out by implementing and testing such a model on various existing datasets, and attempt to make meaningful conclusions about video prediction and learning causality.

2 The Task of Video Prediction

The task of video prediction is to construct an approximation for the completion of a sequence of frames, given only the initial sequence of frames. Formally, given an ordered set of n image frames $\boldsymbol{X} = (X_1, X_2, X_3 \cdots X_n)$, the task is to predict the latter m frames of the sequence $\boldsymbol{Y} = (Y_1, Y_2, Y_3 \cdots Y_m)$, each frame of which having the same dimensions, for example with c channels, height h, and width w. At each inference in training, the model predictions $\hat{\boldsymbol{Y}} = \left(\hat{Y}_1, \hat{Y}_2, \hat{Y}_3 \cdots \hat{Y}_n\right)$, with the same dimensions as the inputs, are conditioned on the input sequence $\boldsymbol{X}$, and the model weights are updated typically by the gradient of a loss function computed between the predictions and ground truth sequence $\boldsymbol{Y}$ directly. Critically, since there is no human intervention or labeling required for the model to do this, and models typically are able to learn from the implicit temporal organiziation of the video data, video prediction is a self-supervised task [9].

3 Families of Prediction Models

Modern video prediction models tend to adopt several canonical architectures, which are normally simple and easily generalizable for specific tasks, such that they can be used as building blocks or blueprints within larger architectures. Understanding what each architecture seeks to do on a high level is imperative to understanding what kind of output one should expect from the model, since they are each very unique, and research implementations make use of them in different ways.

For example, RNNs and generative networks are each successful and well-tested paradigms used in many other machine learning domains outside of computer vision, but they are especially utilized within video prediction because of their key properties and strengths, particularly of RNNs to work on time-sequential data and of generative networks to "imagine" new data within a distribution. These paradigms are used extensively in Convolutional LSTM, FutureGAN [2], and SAVP models [7], as well as many other models in cutting-edge research. A short description of each family and its relevance to video prediction is given below:

3.1 Recurrent Neural Networks

Formally, a Recurrent Neural Network (RNN) is the end-result of a mathematical analysis of a nonlinear first-order non-homogeneous ordinary differential equation describing the evolution of a state signal s as a function of time, along with an input signal x. The canonical statement of an RNN is given below in the form of a discrete Delay Differential Equation (DDE) [11]:

$$\begin{aligned} s_t &= W_s s_{t-1} + W_r r_{t-1} + W_x x_t + \theta_s \\ r_t &= G(s_t) \end{aligned} \tag{1}$$

With W_s, W_r, and W_x as weights which are either multiplied or convolved with their respective signals, and θ_s as a bias term which is typically added in element-wise fashion to s. This equation also includes the read-out or output signal r, which is the activation $G(z)$ of the state signal, and can be seen as the "output" of the network. All together, this equation describes all of the moving parts of an RNN.

As a toy example (and ignoring some technical tricks that would be required to make this actually happen), a network of this type could be capable of transcribing seqeunces of a lecture by evaluating discrete audio samples recorded from the event and outputting the spoken words in text form [11]. Another extremely common application of vanilla RNNs is machine translation, in which the input sequence is text in one language and the output is meant to be the translation of that text into another language.

An easily understandable depiction of the RNN described in Equation 1 can be found in Figure 1. This figure also shows the "unfolding" or "unrolling" of the model, which is just a way of seeing each time-step of the state signal s_t, input signal x_t, and ouput signal r_t (here replaced by $\hat{y}_t$, denoting the network inferences) as they are produced over time.

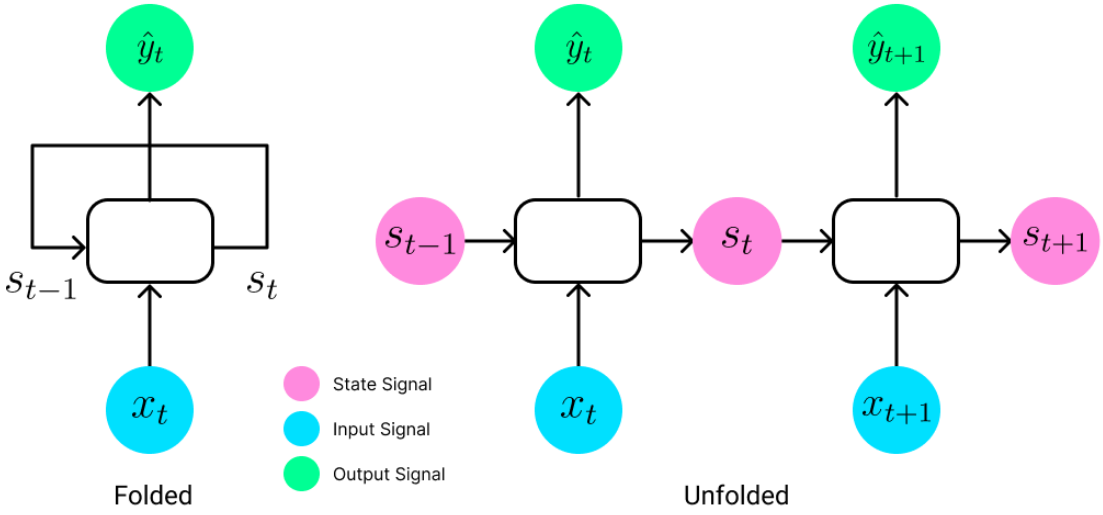

Figure 1: RNN Architecture

RNNs are generally trained using a techinque called Backpropagation Through Time (BPTT), in which the gradient of a loss function taken at a certain instance in time is used to update the model parameters recursively through the history of the model, using the chain rule. In this way, RNN models trained over long sequences have their parameters updated by the product of many Jacobian matrices, which, just like a product of many real numbers, can vanish or explode very easily [10]. For this reason, along with several others having to do with the long-term stability of RNNs, the Long Short-Term Memory (LSTM) cell was developed with nonlinear, data-dependent controls in the form of several "gates" that control input to and output from the cell's state [11].

3.2 Long Short-Term Memory (LSTM) Cell

The changes described in the following section take the form of a modified system of equations [13]:

$$\begin{aligned} f_t &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f) \\ i_t &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i) \\ \tilde{c}_t &= \tanh(W_{\tilde{c}h}h_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}) \\ o_t &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \tanh(c_t) \end{aligned} \tag{2}$$

The key differences in the LSTM are the four gates denoted by f, i, $\tilde{c}$, and o. Other than these, the cell state c and output h are analogous to the original RNN definition. Now, the input and output gates are able to "learn" how to add information to the cell

state or take information for inferences, and the forget gate is able to remove information from the cell state entirely. To see how this is done, recall that the sigmoid funtion $\sigma(z)$ has an output range of $(0, 1)$, meaning that element-wise multiplication by embeddings activated by sigmoid can minimize and essentially nullify the cell state. In the input gate, this same technique is applied for i onto $\tilde{c}$, and in the output gate it is applied by o onto c, meaning that i decides what from h_{t-1} is allowed into the cell state and o decides what from c_{t-1} is allowed out of the cell state into the inference. A diagram of these connections is shown in Figure 2.

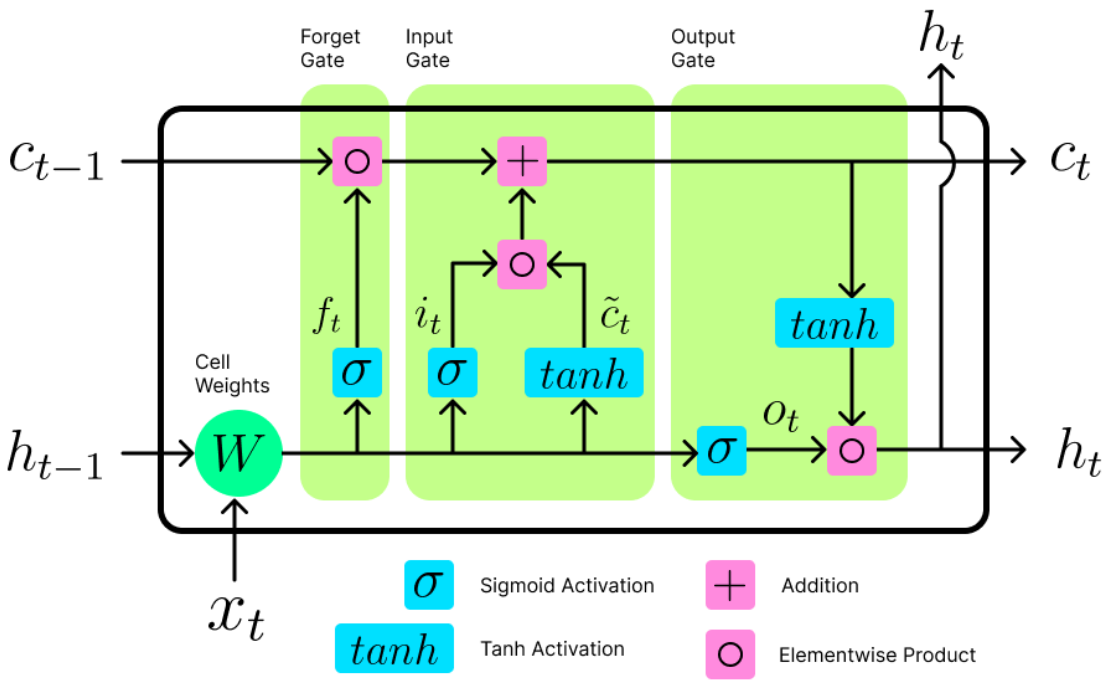

Figure 2: LSTM Cell Architecture

In this way, LSTM cells are capable of mitigating some of the drawbacks of RNNs such as vanishing and exploding gradients, and they are capable of learning how to make good inferences for very different sequences. In other words, they are better able to approximate both y_{t_1} and $y_{t_2 \gg t_1}$ using the same parameters.

Now that some of the general architectures and methodologies are laid out, however, the question still remains of how these models actually predict the continuation $\boldsymbol{Y}$ of a sequence of frames $\boldsymbol{X}$ with any sort of accuracy. In order to do this, one final methodology must be examined, namely Sequence to Sequence (Seq2Seq) learning, which will lead to the final implementation of a Convolutional LSTM used for experimentation in this project.

3.3 Sequence To Sequence Learning

Seq2Seq learning is only a slight modification to the original usage of RNNs, but they represent an elegant solution to a classical shortcoming of most Deep Neural Networks (DNNs), which is that, despite their flexibility, they can only be applied to problems in which inputs and outputs can fit into fixed, discretized, vectors, and therefore must be of a certain length or size. Seq2Seq learning solves this problem for video prediction as well as many other tasks by using 2 LSTMs: an encoder to read the sequence and learn to create an embedding vector, and a decoder, which is passed the embedding vector and learns to generate the predicted sequence [12]. A diagram of this procedure is shown in Figure 3.

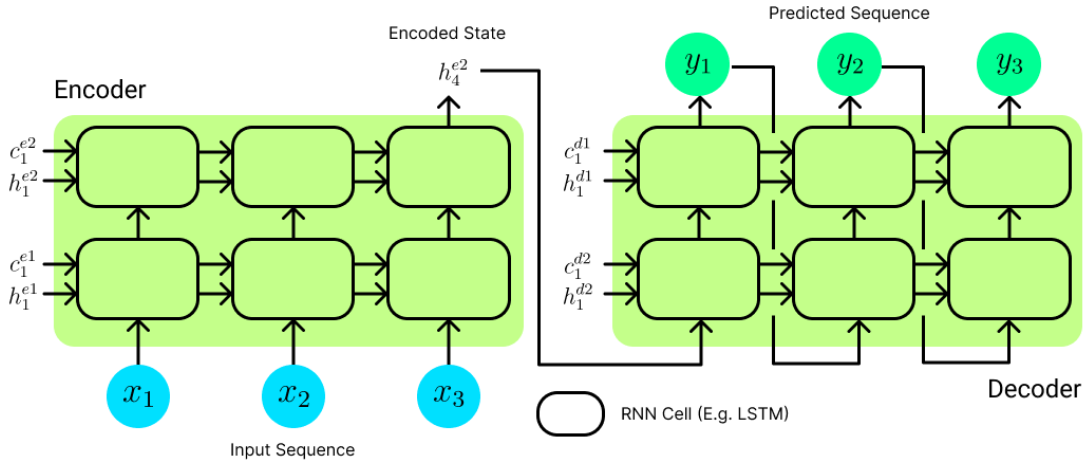

Figure 3: Seq2Seq Architecture

The diagram shows how the input sequence x is fed into the encoder LSTM for 3 steps (in practice this can be any length), how the encoder generates an embedding vector h_4 which is passed to the decoder as an input, and how the decoder generates the predicted sequence by passing the predicted output $\hat{y}_t$ back into itself at the next time step as an input, continuing until all predicted frames are generated.

This diagram also shows two layered LSTM cells, which is another key modification used in this project's implementation. Note that each layer has it's own states h and c, and at each time step, the cells pass their h embedding up to the next cell as an input. Deep, multilayered LSTMs have been shown to significantly outperform shallow LSTMs for language translation [12], however, this project will also analyze the effect of depth on LSTM inferences for video prediction.

In sum, these are the main methods used in this report's implementation of an LSTM model for video prediction, which can be found at this GitHub repository.

4 Implementation and Training Details

5 Datasets

6 Experiments

The main experimental procedure carried out in this report is the training and testing of the Convolutional LSTM model on the MovingMNIST, KTH, and BAIR datasets, however, in order to more gain a more robust understanding of LSTM features and limitations, as well as to debug the training mechanisms in a much simpler setting, it was useful to first test a Linear LSTM model on one-dimensional sequential data before moving fully to video prediction. Three datasets of this type were implemented: a dataset consisting of generated sin waves with random frequency and phase offset, a dataset consisting of NASDAQ close price data from 8 tech companies, and a dataset consisting of randomly generated points. The choice of these datasets were intended to each test the model with a varying degree of stochasticity, the sin waves being the most deterministic, or predictable, the random points being the most stochastic, or random, and the stock price data hopefully being somewhere in between. Some specific details about these datasets are described in the following sections.

When researching and implementing a prediction model of any sort, it is important to consider that certain sequences are simply impossible to predict, and that even if the model was "perfect" by human standards, it would still fail to perform this task perfectly. The interesting question here is not whether the model will be able to achieve a certain accuracy in predicting the sequences but rather where or how exactly will it fail? And where it does fail, which features of the model's architecture and methodology are to blame?

All these questions lead directly into one of the major concerns in video prediction research, which is that the task is inherently hard to judge [9], particularly that pixel-wise loss functions, such as the Mean Squared Error (MSE) function used to train this model, cause models to prefer blurry results that average out multiple possibilities for the next time step rather than a single, clearer image that could be very wrong using pixel-wise metrics. Models trained in this way are not directly learning to produce images but merely to appease the loss function, and these results may not be as clear or useful to humans.

6.0.1 GeneratedSins Dataset

This dataset was generated by simply plotting a sinusoid function:

$$\boldsymbol{X}(t) = \boldsymbol{Y}(t) = \frac{1}{2}(\sin(\alpha t + \beta) + 1) \tag{3}$$

With random α and β. This results in a sin wave with a random frequency and phase offset plotted within the range $[0, 1]$. Although this dataset has random features, once the sin wave is detected and learned by the model, it should be relatively easy to predict, since sin waves are perfectly periodic. $\boldsymbol{X}$ is evaluated from $[0, 0.5)$, and $\boldsymbol{Y}$ is evaluated from $(0.5, 1]$. In sum, the train split consists of 22000 samples, and the test set consists of 7000.

6.0.2 GeneratedNoise Dataset

This dataset was generated by using pytorch's random tensor function `torch.rand()`, which generates a tensor of specified dimensions, where each entry is a float uniformly distributed within $(0, 1)$. To make this data a little easier to view, this data was then smoothed using `scipy.ndimage.filters.gaussian_filter1d()` function, which convolves the signal with a 1-dimensional gaussian kernel with $\sigma = 1$. The train split consists of 22000 samples, and the test set consists of 7000.

6.0.3 Stocks Dataset

This dataset was sourced using Alpha Vantage [1], a free online stock price API. Close price data is provided from the past two years for the NASDAQ tickers 'GOOGL', 'MSFT', 'TSLA', 'AAPL', 'AMZN', 'NVDA', 'FB', and 'AMD'. This data is normalized by subtracting the minimum and dividing by the maximum for each sequence individually, such that each sequence pulled from the dataset lies within $(0, 1)$, and is therefore comparable to the other datasets. The train split consists of 22277 samples, and the test set consists of 7312 samples.

6.1 Sequence Prediction

The MSE loss was recorded as the model was trained, and a plot of this is shown in Figure 4. In addition, after training was complete, a plot was generated of the model's average loss on the test split as the predicted sequence progresses, and it is shown in Figure 5. The average loss over the whole test split is shown in Table 1.

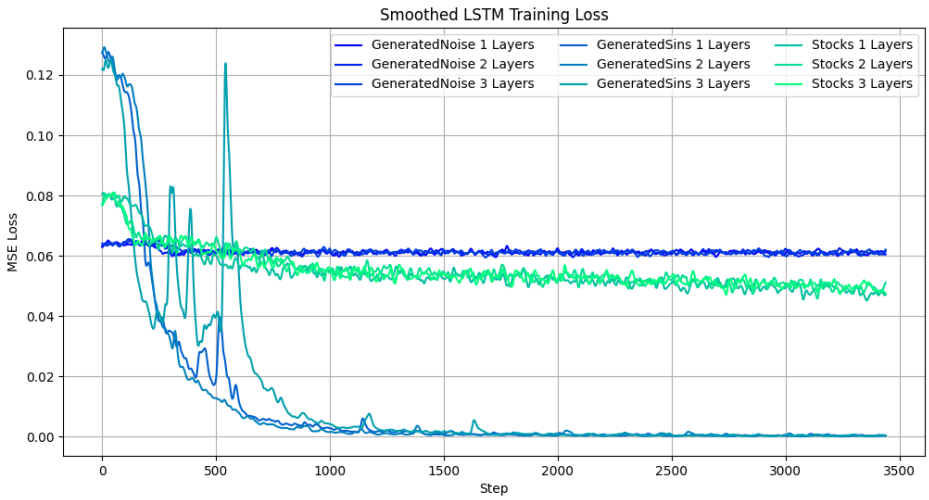

Figure 4: LSTM Training Loss on GeneratedSins, GeneratedNoise, and Stocks Datasets for Models with Varying Number of Cell Layers

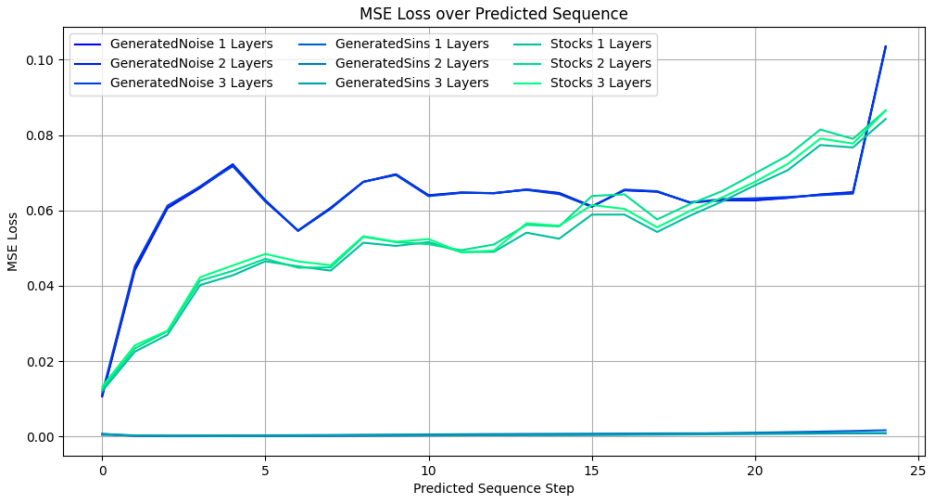

Figure 5: LSTM Test Loss over Predicted Sequence Steps for GeneratedSins, GeneratedNoise, and Stocks Datasets for Models with Varying Number of Cell Layers

Table 1: LSTM Average Test Losses vs. Number of Layers

Layers	**GeneratedNoise**	**GeneratedSins**	**Stocks**
1	5.964×10^{-2}	4.950×10^{-4}	5.227×10^{-2}
2	5.967×10^{-2}	6.298×10^{-4}	5.431×10^{-2}
3	5.961×10^{-2}	4.804×10^{-4}	5.381×10^{-2}

6.1.1 GeneratedSins Experiment

The results shown in the previous figures confirm this dataset to be the most predictable of the three. This can be observed in the plot of training losses in Figure 5, which shows that models trained on GeneratedSins data generally decreased the most over training out of the three, and in the plot of average test losses over the predicted sequence in Figure 4, which shows that GeneratedSins led to the most accurate predictions over longer sequences by far. Additionally, Table 1 shows that the average test loss from this dataset was only 10^{-2} times the average losses of the others. A Test inference of a 2-layer LSTM model trained on this dataset with the ground truth sequence underlaid is shown in Figure 6.

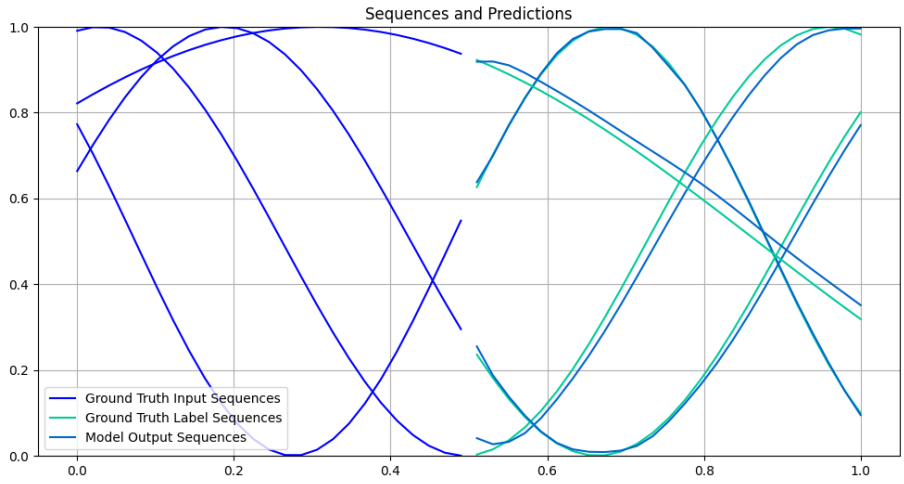

Figure 6: 4 LSTM Inferences on Test Split of GeneratedSins Dataset

As clean and impressive as these predictions are, however, they are not perfect, and still decrease in accuracy as the sequence progresses. While the errors are minor in this demonstration, the shortcoming reveals a pretty major property of the Seq2Seq architecture (and all sequence prediction architectures, not just Seq2Seq), which is that in general, predictions will tend to decrease in accuracy as they become longer. In most situations, this is because real data is at least some part stochastic, meaning that, as time goes on, a sequence can have many possible outcomes, and the state of the sequences at any far-off point in time is a composition of all the previous random events that take place. In this situation, however, a feature of the Seq2Seq modeling is most likely the culprit (excluding possible training and implementation errors), since there is *no* inherent stochasticity of a sin wave, that is, it is not a random process at all and once it is established at any point in time it will continue in that way forever.

In this situation, when the model feeds each LSTM cell's output directly back into itself as an input for the next time step in order to generate the full sequence (See Figure 3), if any prediction is off by a tiny amount, then the next prediction will likely share and even increase that error. While it is likely that this is the phenomenon causing the majority of the error in this situation, the small initial errors are most likely due to incomplete convergence of the model, since the training and test set should theoretically approach the same distribution as more samples are drawn...

6.1.2 GeneratedNoise Experiment

This dataset, in contrast, was nearly impossible for the model to learn satisfactorily, as evidenced by the training loss plot shown in Figure 4 which is essentially flat. Additionally, over the test sequences on average in Figure 5, the model increases most rapidly in error as the prediction increases in duration. While this result is somewhat expected for this

dataset, it is still concerning that the model is capable of predicting any amount of a random signal for any duration. The answer to this question lies in the details of the dataset implementation, specifically that the points are not purely random; they become slightly cross-correlated after being smoothed by the gaussian filter, due to the convolution action which sums neighboring points. A batch of test inferences showcasing this phenomenon is shown in Figure 7.

As for the model implementation and architecture, one important aspect of the results on this dataset stands out, which is that the model predicts nothing but a flat line centered at 0.5 after these initial predictions. This may seem like an uninteresting result, but what it shows is fundamentally the same issue that will cause blurry predictions in actual video predictions, and this is one of the current challenges discussed in video prediction research [9].

To analyze the issue here, it is necessary to closely evaluate the loss function used in training this model:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4}$$

The issue with this loss function is that it assumes that the underlying data distribution is Gaussian. Here, it is nearly the case that $X, Y \sim U(0, 1)$, and the expected value of the uniform distribution $\mu = \frac{1}{2}(1 - 0) = 0.5$ will minimize the MSE loss, even if μ itself has very low probability. This feature of pixel-wise loss functions has negative side-effects for video prediction, since the mean prediction does not usually yield very realistic frames, and these types of predictions are not as directly useful to humans.

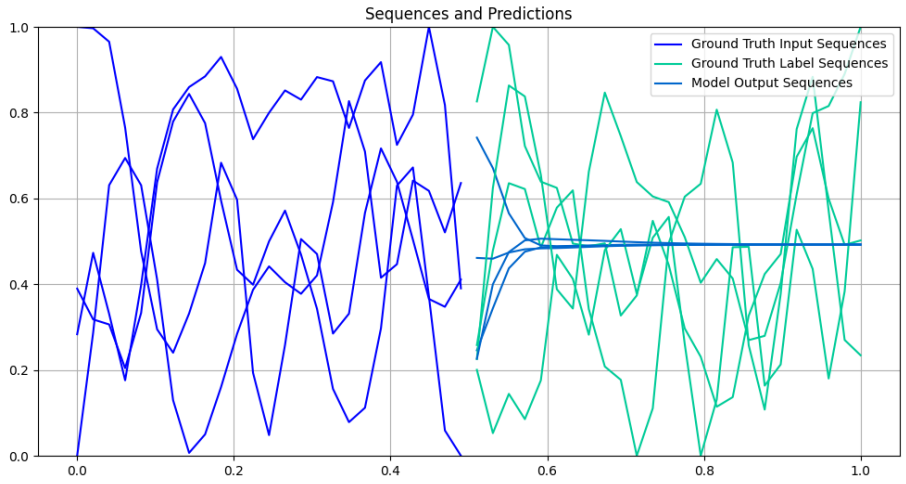

Figure 7: LSTM Inference on GeneratedNoise Dataset, with Ground Truth Input X in Blue, Ground Truth Label Y in Green, and Model Prediction in Purple

6.1.3 Stocks Experiment

Prior to performing this experiment, it was difficult to hypothesize how the LSTM model would perform on the stock price dataset, partly because the author knows very little about finance in general, but also because if it could be done reliably, then it would not be hard to make a lot of money very easily! However, from the previous experimentation with the GeneratedNoise dataset, it was reasonable that stock data points should be more cross-correlated between neighboring points, since stock prices should have some sort of momentum, that is, if they are rising they should be slightly more likely to continue to rise, and vice versa.

In practice, the results of testing the model on this dataset showed some interesting phenomenon and differences when compared to GeneratedNoise. Firstly, the model does not just predict the mean for every sequence with this dataset, instead, it attempts to fit each line individually, as can be seen in Figure 8. In addition, the average test loss over this dataset shown in Figure 5 reveals that the prediction error does not decrease as quickly as with GeneratedNoise, and the training loss plot shown in Figure 4 shows that the overall decrease in loss over the stocks dataset was greater than for GeneratedNoise. These results are therefore very interesting, as they show that the model does learn something from the stocks sequences, even though they are, at first glance, just as random and unpredictable as the random noise.

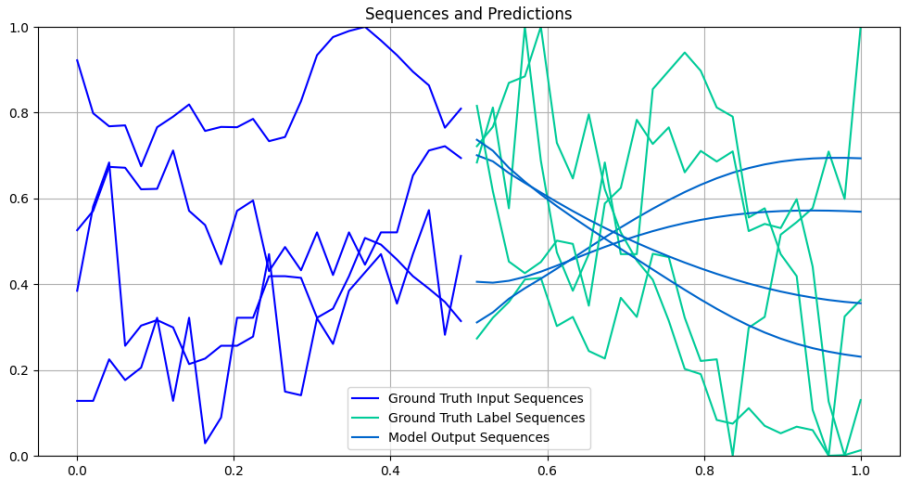

Figure 8: LSTM Inference on Stocks Dataset, with Ground Truth Input X in Blue, Ground Truth Label Y in Green, and Model Prediction in Purple

While some of the predictions from this dataset are impressive, however, and seem to almost approximate the trajectory of the stock price, a good amount of them are also mostly wrong, and even seem to diverge from the ground truth label completely. From a qualitative perspective, these predictions do not seem accurate enough to make any

serious decisions off of, however, it would be very interesting to test the performance of this model in making trading decisions as a future experiment.

6.2 Video Prediction

Now on to the real meat of this project, which is to use a variation of the previously-used model equipped with convolution in order to predict the continuation of video sequences.

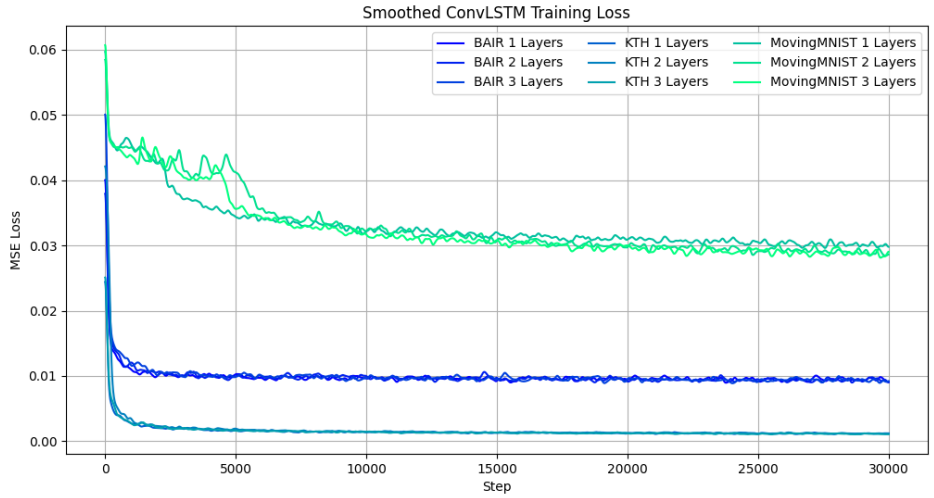

Figure 9: ConvLSTM Training Loss on MovingMNIST, KTH, and BAIR Datasets for Models with Varying Number of Cell Layers

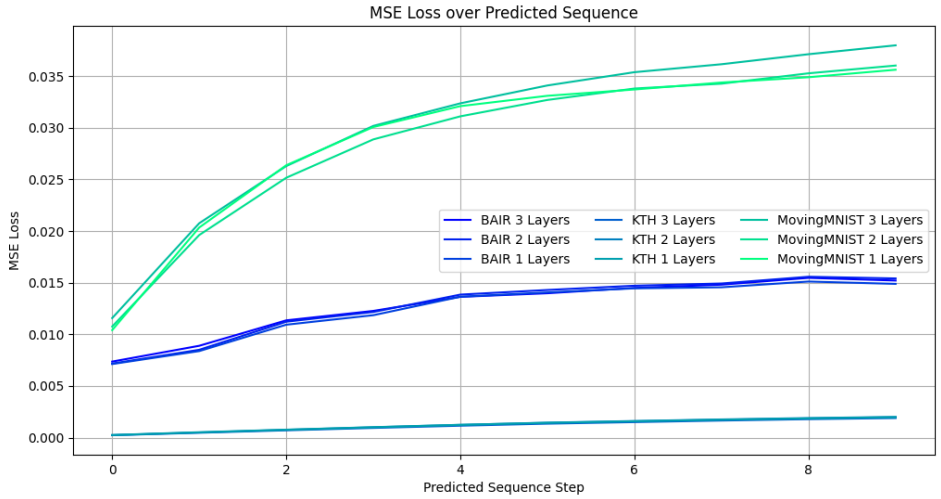

Figure 10: ConvLSTM Test Loss over Predicted Sequence Steps on MovingMNIST, KTH, and BAIR Datasets for Models with Varying Number of Cell Layers

As before, this model's training loss over the three video datasets MovingMNIST, KTH, and BAIR are shown in Figure 9, and the average test loss plotted over the prediction sequence steps for each dataset and number of cell layers is shown in Figure 10. Additionally, the average test loss is shown in Table 2.

Table 2: ConvLSTM Average Test Losses For Each Dataset and Number of Layers

Layers	**MovingMNIST**	**KTH**	**BAIR**
1	1.249×10^{-2}	1.233×10^{-3}	2.910×10^{-2}
2	1.278×10^{-2}	1.239×10^{-3}	2.876×10^{-2}
3	1.274×10^{-2}	1.163×10^{-3}	3.019×10^{-2}

6.2.1 Moving MNIST

The model's inferences on the MovingMNIST were annoyingly mixed, with some results being quite clear, such as the one shown in 11, and others being much more muddled and blurry, such as the one shown in 12. Since the MovingMNIST dataset has elements of predictability (for example, digits bounce off the walls in a predictable way and move at a constant velocity) the expectation was that this dataset would be easy for the ConvLSTM model to learn, however, the training loss plot in Figure 9 and the average test sequence loss in Figure 10 show that this dataset was the most difficult of the three.

As to why this is the case, it could be a combination of several factors: poor bias learning, occlusion, or model depth and training time limitations.

For the first reason, it is important to look at the ways in which MovingMNIST is unique to the other datasets, for example, it is extremely bimodal, and pixel intensity values densely cluster around 1 and 0, or white and black, as they appear in images. Although the LSTM architecture (2) includes a bias term in its weights, the bias needed here is not a simple pixelwise correction, it is a correction term needed for each frame as the digits bounce around. While a bias term such as the one included in the vanilla LSTM cell implementation is capable of correcting the image predictions to be mostly black, for example, it is not capable of learning that the pixels in each digit as they bounce around should be white, since the bias terms are constant over the entire prediction. Thus, the model struggles to produce such black-and-white predictions.

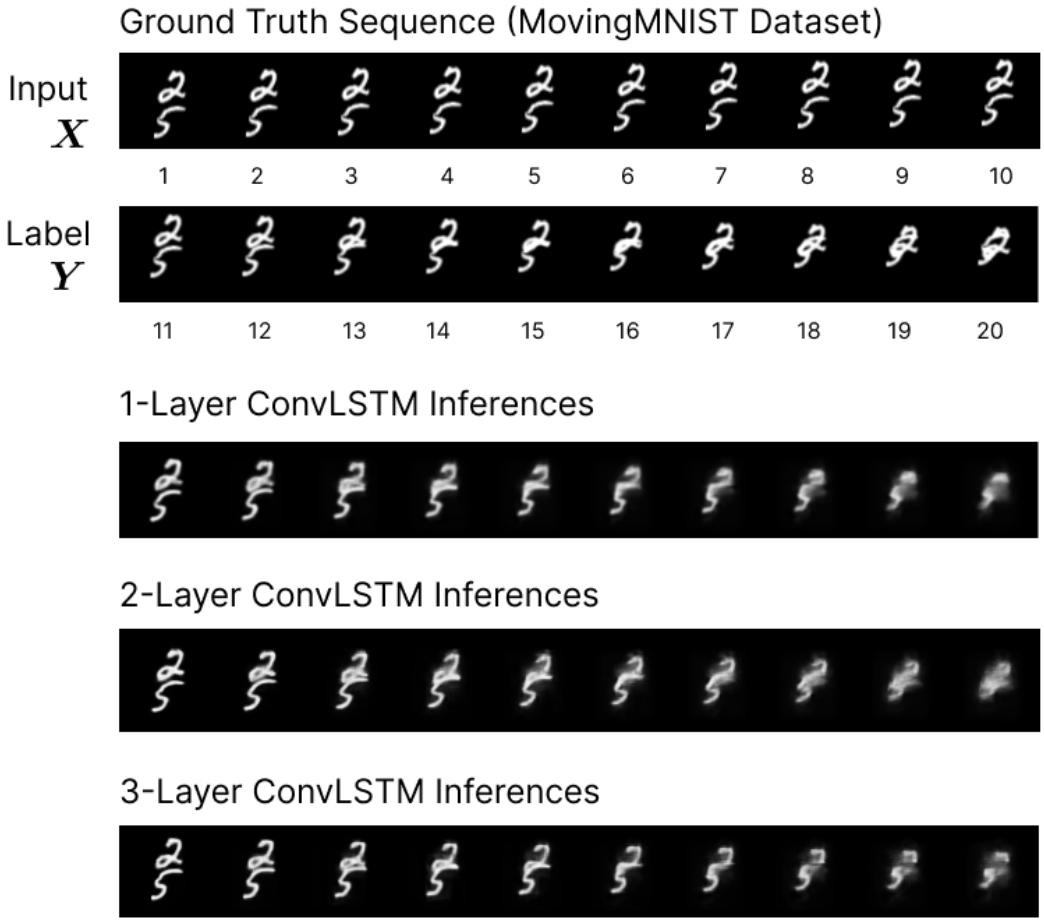

Figure 11: Convolutional LSTM Inference on MovingMNIST dataset

While the second reason doesn't account for the high training loss directly, since occlusion is only present in about half of the sequences and even then isn't a major factor in most cases, it does increase the underlying complexity of the dataset to some degree. This, combined with the third reason, could also be major contributors to the poor prediction accuracy in this case, since the model was only able to train over 10 epochs due to time and computing power constraints. Additionally, the dataset is generated on the fly from the MNIST digits dataset [4], so it would probably help to train a deeper model along with many more generated sequences.

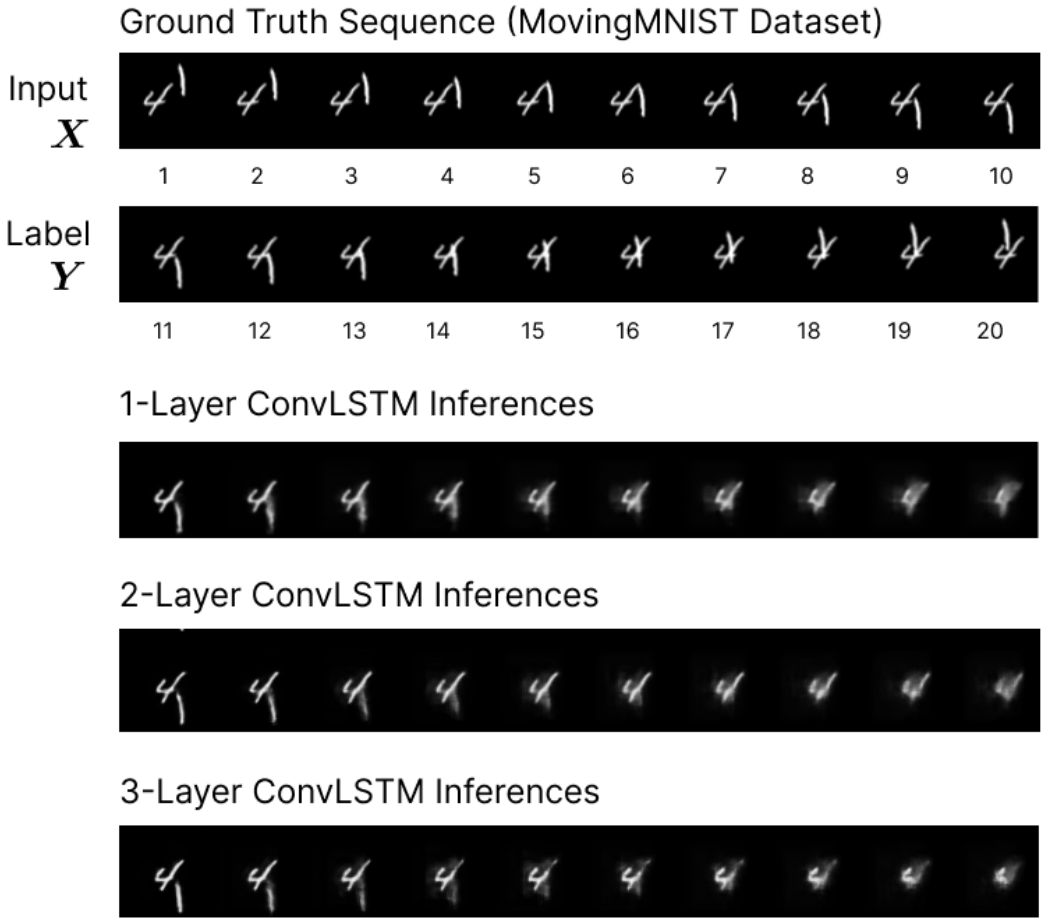

Figure 12: Convolutional LSTM Inference on MovingMNIST dataset

6.2.2 KTH

The ConvLSTM model trained and tested best on the KTH dataset out of the three, as the loss plots in Figures 9 and 10 confirm, and this success led to some of the most impressive and satisfying predictions in this project.

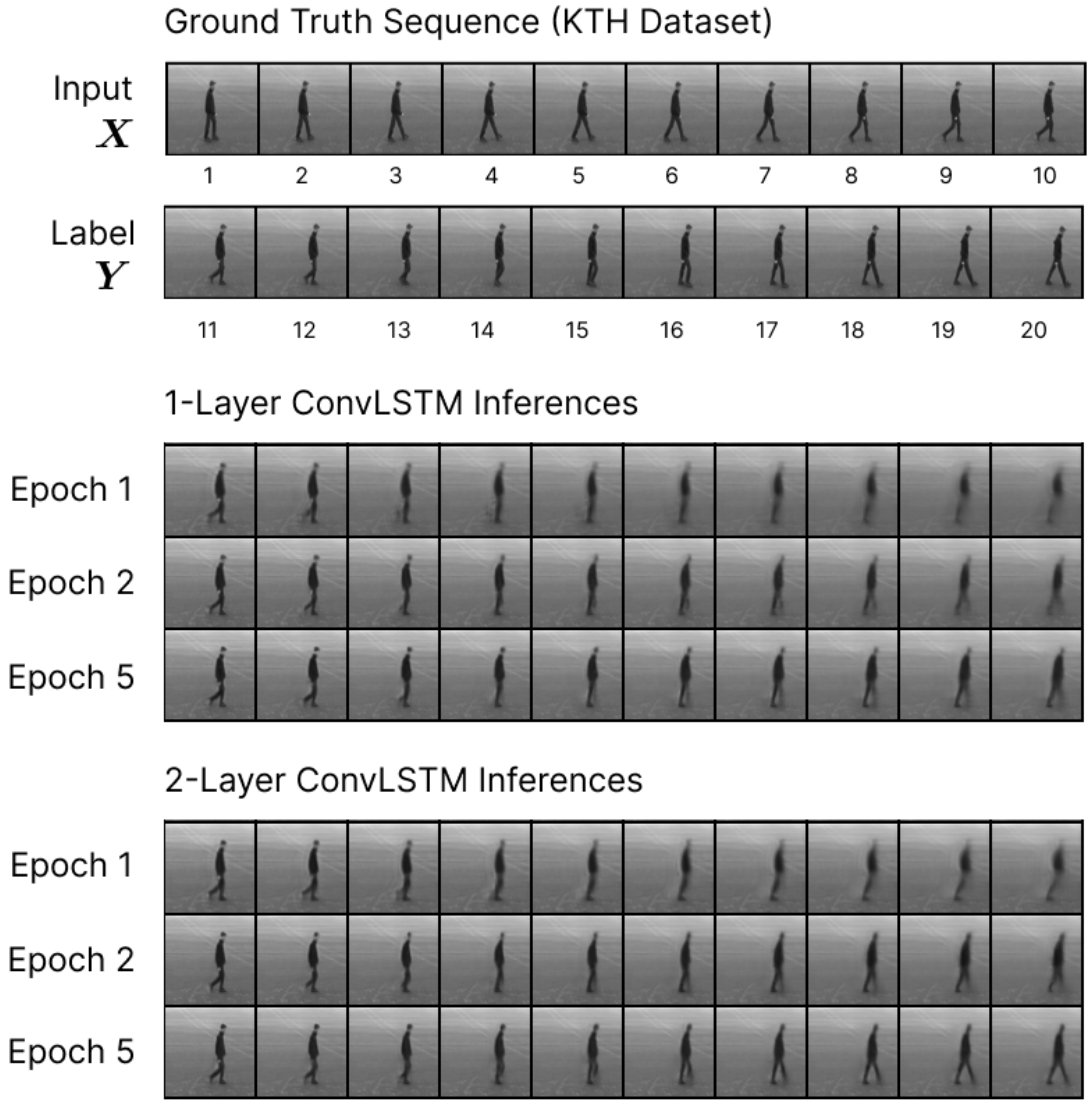

Figure 13: Convolutional LSTM Inference on KTH dataset (Sample 10)

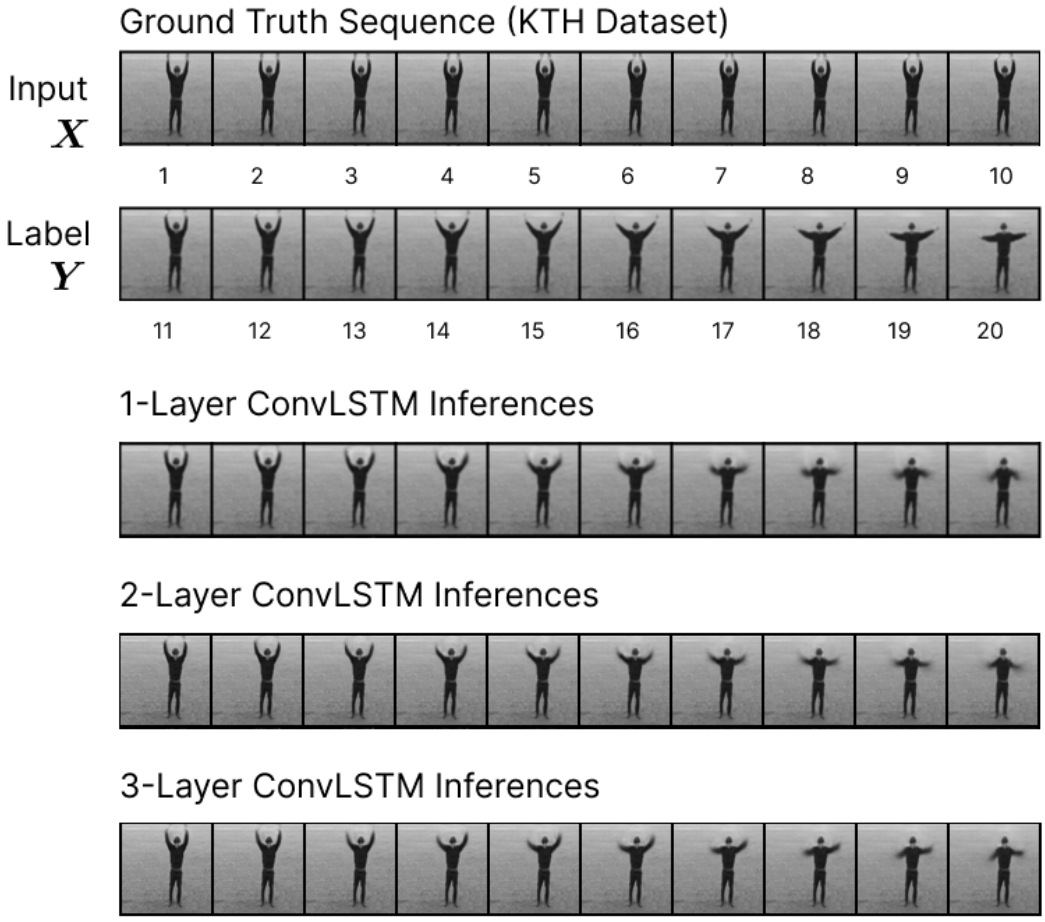

Figure 14: Convolutional LSTM Inference on KTH dataset (Sample 50)

6.2.3 BAIR

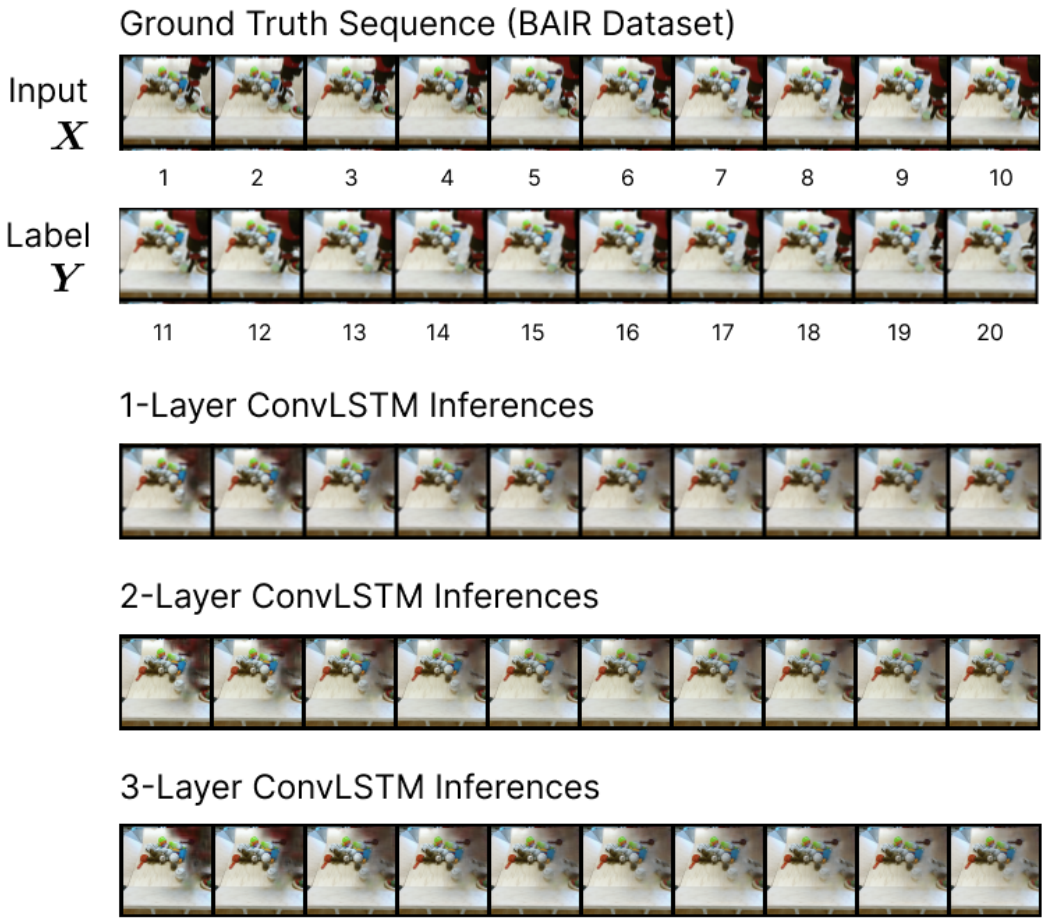

Figure 15: Convolutional LSTM Inference on BAIR dataset

7 Conclusion

References

[1] Alpha vantage, 4 2022. URL https://www.alphavantage.co/.

[2] S. Aigner and M. Körner. Futuregan: Anticipating the future frames of video sequences using spatio-temporal 3d convolutions in progressively growing gans, 2018. URL https://arxiv.org/abs/1810.01325.

[3] A. Cleeremans and J. Mcclelland. Learning the structure of event sequences. *Journal of experimental psychology. General*, 120:235–53, 10 1991. doi: 10.1037//0096-3445.120.3.235.

[4] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[5] C. Finn and S. Levine. Deep visual foresight for planning robot motion, 2016. URL https://arxiv.org/abs/1610.00696.

[6] A. Hu, F. Cotter, N. Mohan, C. Gurau, and A. Kendall. Probabilistic future prediction for video scene understanding, 2020. URL https://arxiv.org/abs/2003.06409.

[7] A. X. Lee, R. Zhang, F. Ebert, P. Abbeel, C. Finn, and S. Levine. Stochastic adversarial video prediction. *CoRR*, abs/1804.01523, 2018. URL http://arxiv.org/abs/1804.01523.

[8] Z. Liu, R. A. Yeh, X. Tang, Y. Liu, and A. Agarwala. Video frame synthesis using deep voxel flow, 2017. URL https://arxiv.org/abs/1702.02463.

[9] S. Oprea, P. Martinez-Gonzalez, A. Garcia-Garcia, J. A. Castro-Vargas, S. Orts-Escolano, J. Garcia-Rodriguez, and A. Argyros. A review on deep learning techniques for video prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 1–1, 2020. ISSN 1939-3539. doi: 10.1109/tpami.2020.3045007. URL http://dx.doi.org/10.1109/TPAMI.2020.3045007.

[10] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks, 2012. URL https://arxiv.org/abs/1211.5063.

[11] A. Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018. URL http://arxiv.org/abs/1808.03314.

[12] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks, 2014. URL https://arxiv.org/abs/1409.3215.

[13] Y. Yu, X. Si, C. Hu, and J. Zhang. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation*, 31(7):1235–1270, 07 2019. ISSN 0899-7667. doi: 10.1162/neco_a_01199. URL https://doi.org/10.1162/neco_a_01199.