Video Frame Prediction Using LSTMs

D Karthik
Department of Computer Science and Engineering
Shiv Nadar University
Greater Noida, Uttar Pradesh
dk984@snu.edu.in

Jaskaran Singh Gujral
Department of Computer Science and Engineering
Shiv Nadar University
Greater Noida, Uttar Pradesh
jg504@snu.edu.in

Nikhil Khandelwal
Department of Computer Science and Engineering
Shiv Nadar University
Greater Noida, Uttar Pradesh
nk832@snu.edu.in

May 20, 2021

Abstract

This document details the authors' efforts on understanding and creating a Long Short-Term Memory Model that can predict future video frames based on previous frames being provided. This document explores Recurrent Neural Networks, Long Short-Term Memory Models, Convolutional Long Short-Term Memory Models, Sequence-to-Sequence Models, and discusses the flaws in the authors' chosen method to predict future video frames.

Table of Contents

Table 1: Table of Contents

1 Introduction

Video is a collection of frames appearing one after another. In this project, we have tried to predict future frames in a video by taking previous n frames as input. This prediction is built on understanding the information of the historical images that have occurred so far. Given a sequence of video frames, it refers to constructing a network that can accurately generate subsequent frames.

For this purpose, we have taken the assistance of tools such as Long Short-Term Memory to build the neural net and convolution to work on the frames. We have taken the help of the library Keras which is a very powerful tool and has most of the functionalities pre-built avoiding the hassle of coding from scratch.

For this project, we studied many papers and thought this implementation to be optimal. We have implemented seq2one architecture, which is a special case of the seq2seq model, by building convLSTM layers to work as an autoencoder.

In this report, we will go through our model in a step-by-step fashion covering all concepts and explaining the model in detail. We will discuss the concepts used and go through the dataset used. This report will explain the model in greater detail with all the helper functions that have been added. We will start by discussing the basis of the basic neural net we have used and then we will build our knowledge to understand the model built to predict the future frames.

2 Recurrent Neural Network

Recurrent Neural Network remembers the past and it's decisions are influenced by what it has learnt from the past. Basic feed forward networks *remember* things too, but they remember things they learnt during training. For example, an image classifier learns what a "1" looks like during training and then uses that knowledge to classify things in production.

While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s). It's part of the network. RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular Neural Network, but also by a *hidden* state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.

In summary, in a vanilla neural network, a fixed size input vector is transformed into a fixed size output vector. Such a network becomes *recurrent* when you repeatedly apply the transformations to a series of given input and produce a series of output vectors. There is no pre-set limitation to the size of the vector. And, in addition to generating the output which is a function of the input and hidden state, we update the hidden sate itself based on the input and use it in processing the next input.

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based

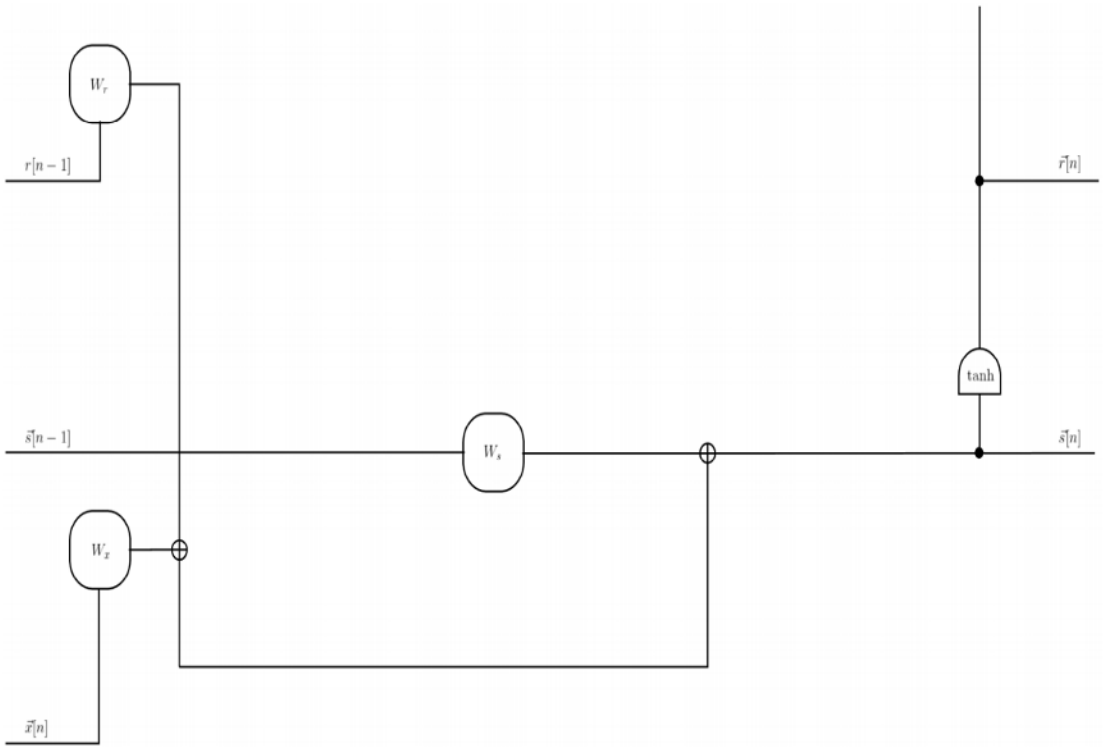


Figure 1: Canonical RNN Cell [10]

on the previous ones. If we are trying to predict the last word in "the Sun rises in the East," we don't need any further context – it's pretty obvious the next word is going to be East. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France. I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

3 Long Short-Term Memory

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by [9]. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through." An LSTM has three of these gates, to protect and control the cell state.

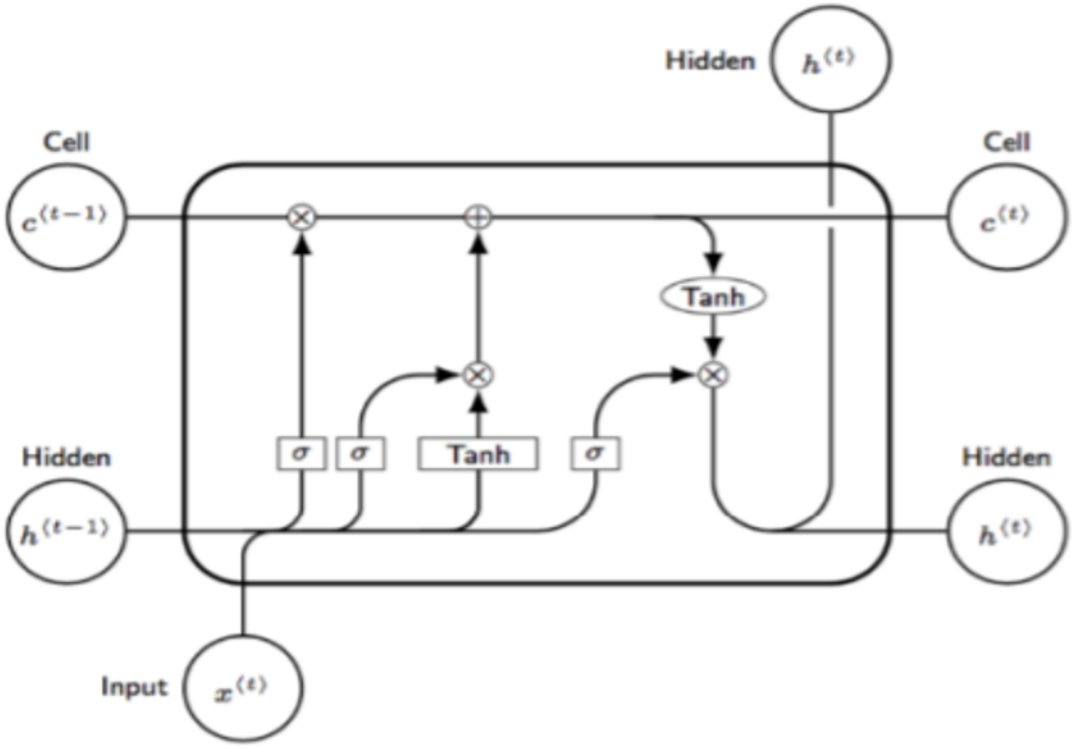


Figure 2: Basic LSTM Cell [7]

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_f)$$

$$\tilde{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C)$$

It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add i_t * $\tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between 1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

The advantages of LSTMs are [9]:

- The constant error backpropagation within memory cells results in LSTM's ability to bridge very long time lags in case of problems similar to those discussed in the previous section.

- For long time lag problems such as those discussed in this paper, LSTM can handle noise, distributed representations, and continuous values. In contrast to finite state automata or hidden Markov models LSTM does not require an *a priori* choice of a finite number of states. In principle it can deal with unlimited state numbers.

- There appears to be no need for parameter fine tuning. LSTM works well over a broad range of parameters such as learning rate, input gate bias and output gate bias. A large learning rate pushes the output gates towards zero, thus automatically countering its own negative effects.

4 Convolutional Long Short-Term Memory

For general-purpose sequence modeling, LSTM as a special RNN structure has proven stable and powerful for modeling long-range dependencies in various previous studies [1,3,8,9]. The major innovation of LSTM is its memory cell c_t which essentially acts as an accumulator of the state information. The cell is accessed, written and cleared by several self-parameterized controlling gates. Every time a new input comes, its information will be accumulated to the cell if the input gate it is activated. Also, the past cell status c_{t1} could be "forgotten" in this process if the forget gate f_t is on. Whether the latest cell output c_t will be propagated to the final state h_t is further controlled by the output gate o_t. One advantage of using the memory cell and gates to control information flow is that the gradient will be trapped in the cell (also known as constant error carousels [9]) and be prevented from vanishing too quickly, which is a critical problem for the vanilla RNN model [8,9,13]. FC-LSTM may be seen as a multivariate version of LSTM where the input, cell output and states are all 1D vectors. In this paper, we follow the formulation of FC-LSTM as in [1]. The key equations are shown in below, where 'Θ' denotes the Hadamard product:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}\Theta c_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}\Theta c_{t-1} + b_f)$$

$$c_t = f_t\Theta c_{t-1} + i_t\Theta tanh(W_{xc}x_t + W_{hc}h_{t-1}) + b_c$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}\Theta c_t + b_o)$$

$$h_t = o_t\Theta tanh(c_t)$$

The major drawback of FC-LSTM [11] in handling spatiotemporal data is its usage of full connections in input-to-state and state-to-state transitions in which no spatial information is encoded. To overcome this problem, a distinguishing feature of our design is that all the inputs $X_1, \ldots, X_t$, cell outputs $C_1, \ldots, C_t$, hidden states $H_1, \ldots, H_t$, and gates i_t, f_t, o_t of the ConvLSTM are 3D tensors whose last two dimensions are spatial dimensions (rows and columns). To get a better picture of the inputs and states, we may imagine them as vectors standing on a spatial grid. The ConvLSTM determines the future state of a certain cell in the grid by the inputs and past states of its local neighbors. This can easily be achieved by using a convolution operator in the state-to-state and input-to-state transitions (see Fig. 3). The key equations of ConvLSTM are shown below, where '$*$' denotes the convolution operator and 'Θ', as before, denotes the Hadamard product:

$$i_t = \sigma(W_{xi}X_t + W_{hi}H_{t-1} + W_{ci}\Theta C_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}X_t + W_{hf}H_{t-1} + W_{cf}\Theta C_{t-1} + b_f)$$

$$C_t = f_t\Theta C_{t-1} + i_t\Theta tanh(W_{xc} * X_t + W_{hc}H_{t-1}) + b_c$$

$$o_t = \sigma(W_{xo}X_t + W_{ho}H_{t-1} + W_{co}\Theta C_t + b_o)$$

$$h_t = o_t\Theta tanh(C_t)$$

If we view the states as the hidden representations of moving objects, a ConvLSTM with a larger transitional kernel should be able to capture faster motions while

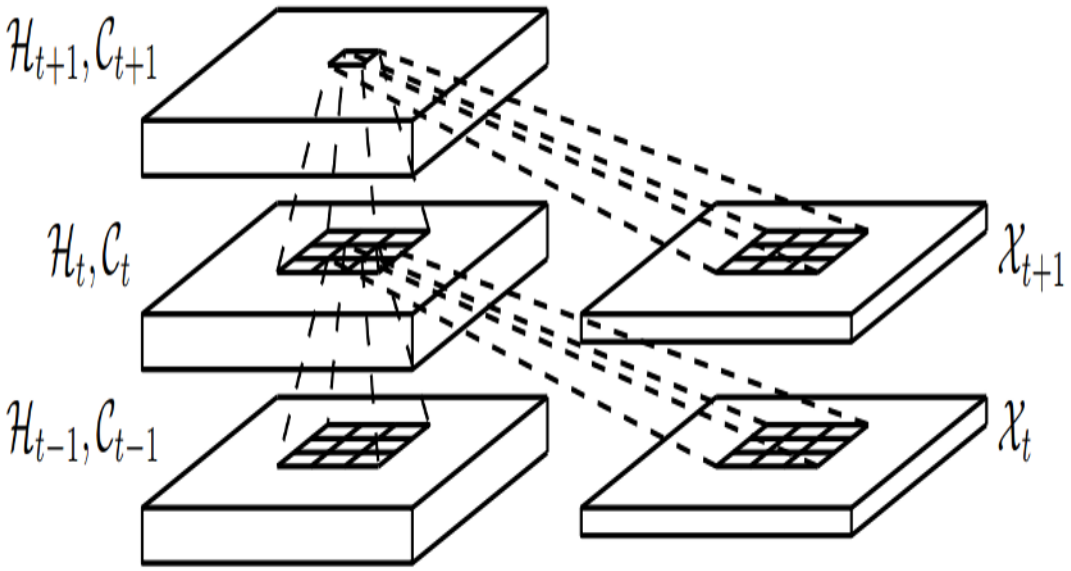


Figure 3: Inner Structure of ConvLSTM [11]

one with a smaller kernel can capture slower motions. Also, if we adopt a similar view as [4], the inputs, cell outputs and hidden states of the traditional FC-LSTM represented by equations above may also be seen as 3D tensors with the last two dimensions being 1. In this sense, FC-LSTM is actually a special case of ConvLSTM with all features standing on a single cell.

To ensure that the states have the same number of rows and same number of columns as the inputs, padding is needed before applying the convolution operation. Here, padding of the hidden states on the boundary points can be viewed as using the state of the outside world for calculation. Usually, before the first input comes, we initialize all the states of the LSTM to zero which corresponds to "total ignorance" of the future. Similarly, if we perform zero-padding on the hidden states, we are actually setting the state of the outside world to zero and assume no prior knowledge about the outside. By padding on the states, we can treat the boundary points differently, which is helpful in many cases. For example, imagine that the system we are observing is a moving ball surrounded by walls. Although we cannot see these walls, we can infer their existence by finding the ball bouncing over them again and again, which can hardly be done if the boundary points have the same state transition dynamics as the inner points.

Like FC-LSTM, ConvLSTM can also be adopted as a building block for more complex structures. For our spatiotemporal sequence forecasting problem, we use the structure shown in Fig. 4 which consists of two networks, an encoding network and a forecasting network. Like in [6], the initial states and cell outputs of the forecasting network are copied from the last state of the encoding network. Both networks are formed by stacking several ConvLSTM layers. As our prediction target has the same dimensionality as the input, we concatenate all the states in the

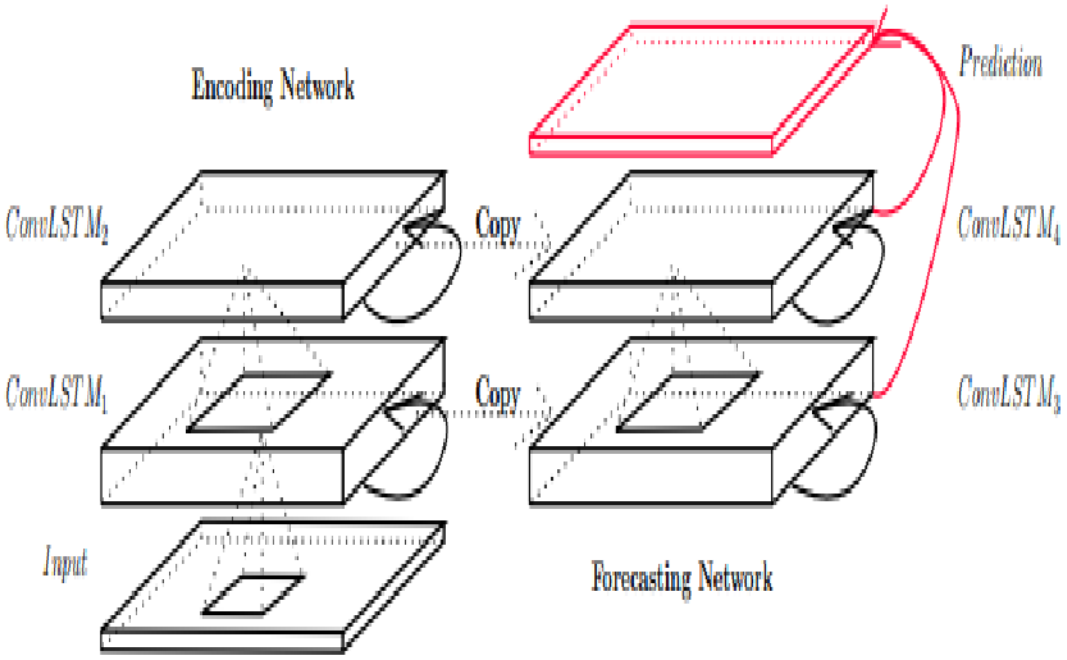


Figure 4: Encoding-forecasting ConvLSTM Network [11]

forecasting network and feed them into a 11 convolutional layer to generate the final prediction.

We can interpret this structure using a similar viewpoint as [3]. The encoding LSTM compresses the whole input sequence into a hidden state tensor and the forecasting LSTM unfolds this hidden state to give the final prediction:

$$\tilde{X}_{t+1}, ..., \tilde{X}_{t+K} = argmax_{X_{t+1},...,X_{t+K}} p(X_{t+1}, ..., X_{t+K} | \hat{X}_{t-J+1}, \hat{X}_{t-J+2}, ..., \hat{X}_t)$$

$$\tilde{X}_{t+1}, ..., \tilde{X}_{t+K} = argmax_{X_{t+1},...,X_{t+K}} p(X_{t+1}, ..., X_{t+K} | f_{encoding}(\hat{X}_{t-J+1}, \hat{X}_{t-J+2}, ..., \hat{X}_t))$$

$$\tilde{X}_{t+1}, ..., \tilde{X}_{t+K} = g_{forecasting}(f_{encoding}(\hat{X}_{t-J+1}, \hat{X}_{t-J+2}, ..., \hat{X}_t))$$

This structure is also similar to the LSTM future predictor model in [6] except that our input and output elements are all 3D tensors which preserve all the spatial information. Since the network has multiple stacked ConvLSTM layers, it has strong representational power.

5 Sequence-to-Sequence Model

A typical sequence to sequence model has two parts – an encoder and a decoder. Both the parts are practically two different neural network models combined into one giant network. Broadly, the task of an encoder network is to understand the input sequence, and create a smaller dimensional representation of it. This representation is then forwarded to a decoder network which generates a sequence of its own that represents the output.

A sequence to sequence model lies behind numerous systems which you face on a daily basis. For instance, seq2seq model powers applications like Google Translate, voice-enabled devices and online chatbots. Generally speaking, these applications are composed of:

- Machine Translation
- Speech Recognition
- Video Captioning

These are only some applications where seq2seq is seen as the best solution. This model can be used as a solution to any sequence-based problem, especially ones where the inputs and outputs have different sizes and categories. We will talk more about the model structure below.

Introduced for the first time in 2014 by Google [12], a sequence to sequence model aims to map a fixed-length input with a fixed-length output where the length of the input and output may differ.

For example, translating "Hello" from English to Mandarin has input of 1 word and output of 2 symbols "你好". Clearly, we can't use a regular LSTM network to map each word from the English sentence to the Chinese sentence. This is why the sequence to sequence model is used to address problems like that one.

The model consists of 3 parts: encoder, intermediate (encoder) vector and decoder.

5.1 Encoder

- A stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward.
- In question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as x_i where i is the order of that word.
- The hidden states h_i are computed using the formula:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$$

This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previous hidden state h_{t-1} and the input vector x_t.

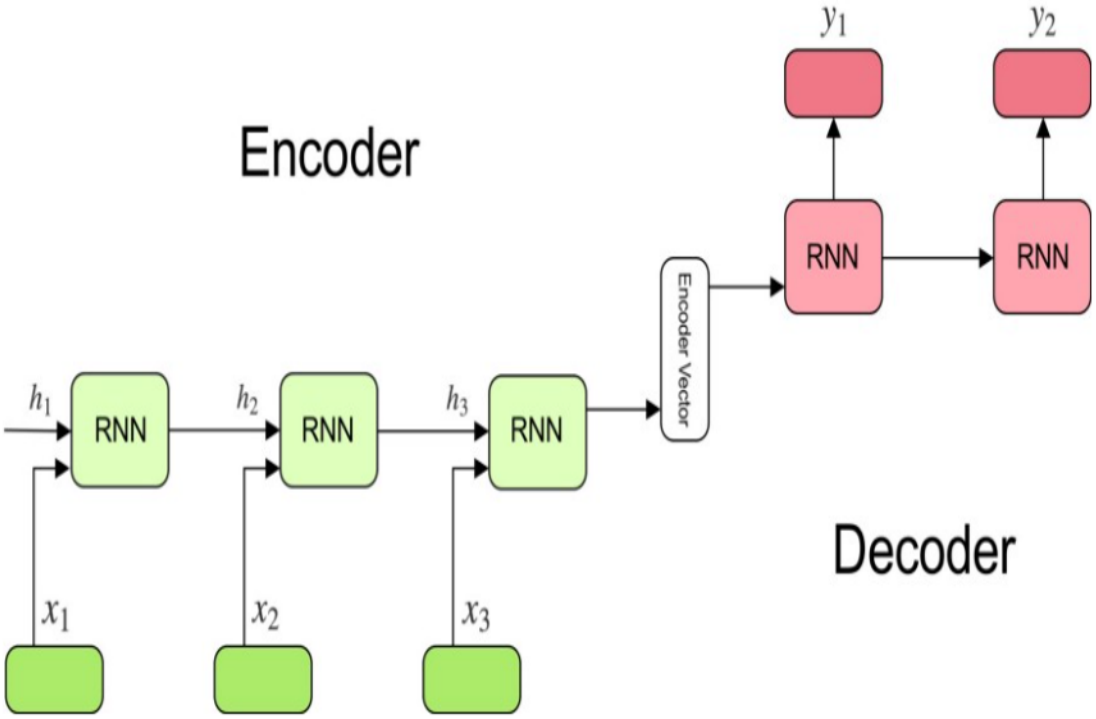


Figure 5: Encoder-decoder Sequence to Sequence Model [12]

5.2 Encoder Vector

- This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above.
- This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.
- It acts as the initial hidden state of the decoder part of the model.

5.3 Decoder

- A stack of several recurrent units where each predicts an output y_t at a time step t.
- Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state.
- In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as y_i where i is the order of that word.
- Any hidden state h_i is computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1})$$

- The output y_t at time step t is computed using the formula:

$$y_t = softmax(W^S h_t)$$

We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help us determine the final output.

The power of this model lies in the fact that it can map sequences of different lengths to each other. As you can see the inputs and outputs are not correlated and their lengths can differ. This opens a whole new range of problems which can now be solved using such architecture.

6 Our Architecture

While researching we came across many architectures such as Prednet, Sequence2Sequence, a simple layer of convLSTM [14]. We read about it all and decided to go ahead with seq2one architecture using ConvLSTM as our encoder and decoder layers [5].

It is imperative to understand the Seq2Seq (Sequence-to-Sequence) architecture in order to understand Seq2One (Sequence-to-One) architecture as the latter is simply a special case of the former.

Sequence2Sequence model was first proposed in 2014 in a paper Sequence to Sequence Learning with Neural Networks by Ilya Sutskever, Oriol Vinyals, and Quoc V. Le in Google [12] and was implemented in 2016 for Google Translate. Due to its origin, the model is typically used for NLP or time-series tasks.

In the most basic terms, a seq2seq architecture is one where one takes a sequence of inputs and produces a sequence of outputs. For example, the input could be a sequence of Spanish words and the output can be a sequence of the English translation.

As mentioned earlier, Seq2Seq consists of 3 components.

1. *Encoder* which encodes the input list. In our example, the encoder takes the Spanish sequence as input by processing each word sequentially.

2. *Encoder Embedding Vector* which is the final embedding of the entire input sequence. In our example, the encoder outputs an embedding vector as the final representation of our input.

3. *Decoder* decodes an embedding vector into an output sequence. In our example, the decoder takes the embedding vector as input and then outputs the English translation sequence.

The technique of autoencoding is very useful especially when dealing with huge data. Now, the Sequence-to-One model is simply a special case of Seq2Seq where the input is a sequence of input but there is only a single output.

For our model, we decided to use convLSTM layers as the encoder and decoder layers as LSTM layers are useful in holding the past information and convolution provides a very powerful tool for working with images. The convLSTM layer provided by Keras has been used as it provides a direct implementation of the layer.

7 Implementation

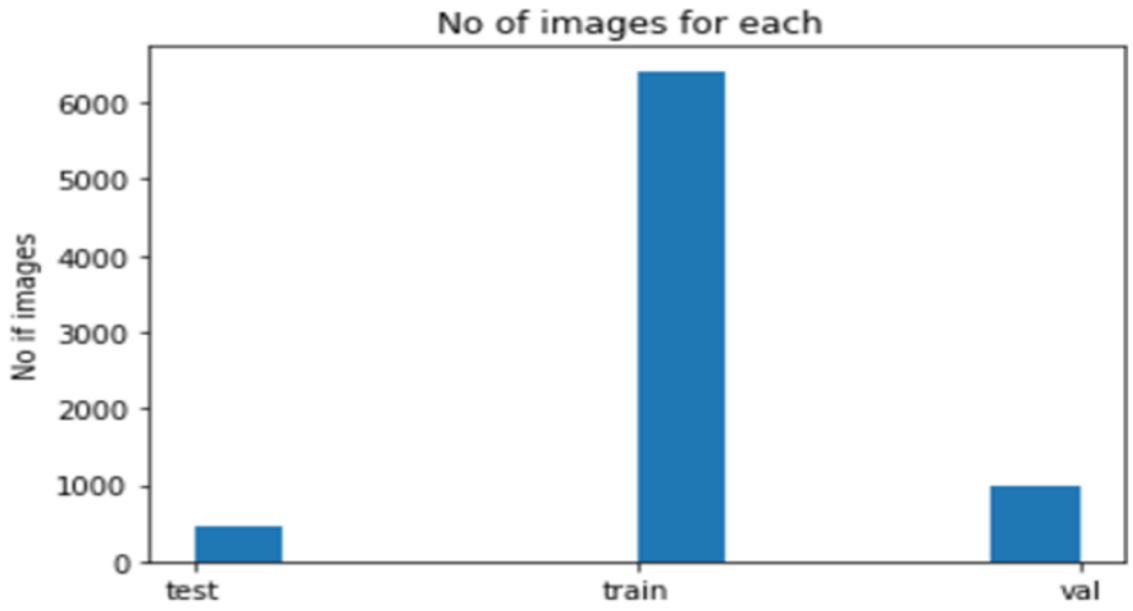
The Jupyter Notebook containing the code can be found at:
https://github.com/jg504/Video_Frame_Prediction/blob/main/Video

7.1 Our Dataset

For this project, we took the use of "Catz" dataset provided by wandb. It comprises of a sequence of images extracted from GIFs of cats and has been prepared by Giphy. Each cat GIF has its own directory containing a sequence of 6 images that constitute the frames of the GIF or the video.

Initially, there were 6421 training sequences and 1475 testing sequences. We had to split them to get a validation dataset as well, and in our model we created three sub-directories inside the Catz directory named Train, Test, and Validation. The train contained 6418 samples with each GIF having its own directory. Validation and Test contained 1000 and 472 samples respectively in a similar format.

To get the dataset you can type in the command:
curl https://storage.googleapis.com/wandb/catz.tar.gz — tar xz




```
No of Training Images: 6418
No of Validation Images: 1000
No of Test Images: 472
```

Figure 6: Distribution of the dataset

7.2 wandb.ai

Before moving any further, we would like to mention and explain the special package we have used for studying the model which is Weights and Biases or wandB. It is a python package that allows us to monitor our training in real-time. It can be easily integrated with popular deep learning frameworks like Pytorch, Tensorflow,

or Keras. Additionally, it allows us to organize our Runs into Projects where we can easily compare them and identify the best performing model. To begin working on this, one needs to sign up on wandb.ai. To know more about this library one can simply go the read the documentation provided by them at https://docs.wandb.ai/ .

7.3 Generator Function

A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

We have built a generator function to generate batches of input to the model. The function takes batch size as input and returns those many inputs for training. But in our case, the input is not just a single image but a sequence of 5 images. We can't send these images one by one to the model, as each frame is dependent on the previous frame. So those 5 input images are concatenated using numpy manipulations and an input of (96,96,15) is passed into the model.

Using the reshape and permute layers defined in the model, each of the image in the sequence is extracted and learnt by the model.

7.4 Callback Function

A callback is a function that is to be executed after another function has finished executing — hence the name 'call back'. So we are doing a callback to log the inputs and outputs on wandb page after every epoch in the training process. The 'ImageCallback' function written in the code is responsible for logging sequence of inputs and the next frame predicted by the model and also the actual frame that has to be predicted. All these are logged as soon as an epoch gets completed. By this, we can see how the model is getting to predict better and better after every epoch. We can also decide the minimum and maximum number of epochs that can be carried out according to the predicted frames after each epoch.

7.5 Metric: Perceptual Distance

Working with colored images is a tricky task since color perception is subjective and not uniform. This is because when we map the true color on a reduced quantization, every true-color pixel must be mapped to the palette entry that comes closest to the original color. Thus the questions "what is the closest color?" and "how does one measure the distance between colors?" become relevant.

If we map color in an orthogonal 3-dimensional space, the distance between two colors is given by $||C_1 - C_2||$, where $||...||$ denotes the Euclidean distance. For a three-dimensional space (with dimensions R, G, and B) the Euclidean distance between two points is calculated as follows:

$$||C_1 - C_2|| = \sqrt{(C_{1,r} - C_{2,r})^2 + (C_{1,g} - C_{2,g})^2 + (C_{1,b} - C_{2,b})^2}$$

Graphic applications for the PC usually employ the Red-Green-Blue (RGB) colour space. This model maps well to the way the common Cathode Ray Tube (CRT) display works. These displays have three kinds of phosphors that emit red,

green or blue light when they are struck by an electron beam. Another advantage of the RGB model is that it is a three-dimensional orthogonal space, precisely what we need for the Euclidean distance function.

We will refer to the red term as $C_{1,r} - C_{2,r}$ as Δ^R and similarly for the green and blue components as well.

Now in order to get to a metric to work well for colored images we we need to work for the following solutions:

- What we need is a formula that gives a "distance" between two colors. This distance will only be used in comparisons, to verify whether one color, A, is closer to color B or to color C.
- In perceptually uniform color space, the Euclidean distance function gives this distance. This is the most straightforward (and obvious) solution, but not the only solution.
- There are three well-known color models (CIE L*u*v*, YUV, and R'G'B') that all score "quite well" in perceptual uniformity (or so they say...).
- The proper test for these and other formulae is to compare their choice of "the closest color" to the color that a person would pick.

Hence, we used the following result which is a combination of both weighted Euclidean distance functions, where the weight factors depend on how big the "red" component of the colour is. First one calculates the mean level of "red" and then weights the $\Delta^{R'}$ and $\Delta^{B'}$ signals as a function of the mean red level. The distance between colours C_1 and C_2 (where each of the red, green and blue channels has a range of 0-255) is:

$$\vec{r} = \frac{C_{1,R} + C_{2,R}}{2}$$

$$\Delta^R = C_{1,R} + C_{2,R}$$
$$\Delta^G = C_{1,G} + C_{2,G}$$
$$\Delta^B = C_{1,B} + C_{2,B}$$

$$\Delta^C = \sqrt{(2 + \frac{\vec{r}}{256}) * \Delta^{R^2} + 4 * \Delta^{G^2} + (2 + \frac{255 - \vec{r}}{256}) * \Delta^{B^2}}$$

This formula has results that are very close to L*u*v* (with the modified lightness curve) and, more importantly, it is a more stable algorithm: it does not have a range of colours where it suddenly gives far from optimal results. Thus we have used the above formula as the metric for our model for measuring the similarity between two images. The metric is called perceptual distance.

7.6 Our Model

As mentioned earlier for the model we have used the seq2one architecture to build our model which works on the autoencoder mechanism. In this section, we will describe all the layers in detail and the entire model. The table below shows the model built and trained.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 96, 96, 15)]	0	
reshape (Reshape)	(None, 96, 96, 5, 3)	0	input_1[0][0]
permute (Permute)	(None, 96, 96, 3, 5)	0	reshape[0][0]
gaussian_noise (GaussianNoise)	(None, 96, 96, 3, 5)	0	permute[0][0]
permute_1 (Permute)	(None, 5, 96, 96, 3)	0	gaussian_noise[0][0]
ConvLstm1 (ConvLSTM2D)	(None, 5, 96, 96, 4)	1024	permute_1[0][0]
batch_normalization (BatchNorma	(None, 5, 96, 96, 4)	16	ConvLstm1[0][0]
time_distributed (TimeDistribut	(None, 5, 48, 48, 4)	0	batch_normalization[0][0]
ConvLstm2 (ConvLSTM2D)	(None, 5, 48, 48, 8)	3488	time_distributed[0][0]
batch_normalization_1 (BatchNor	(None, 5, 48, 48, 8)	32	ConvLstm2[0][0]
time_distributed_1 (TimeDistrib	(None, 5, 24, 24, 8)	0	batch_normalization_1[0][0]
ConvLstm3 (ConvLSTM2D)	(None, 5, 24, 24, 16	13888	time_distributed_1[0][0]
batch_normalization_2 (BatchNor	(None, 5, 24, 24, 16	64	ConvLstm3[0][0]
time_distributed_2 (TimeDistrib	(None, 5, 48, 48, 16	0	batch_normalization_2[0][0]
ConvLstm4 (ConvLSTM2D)	(None, 5, 48, 48, 16	18496	time_distributed_2[0][0]
batch_normalization_3 (BatchNor	(None, 5, 48, 48, 16	64	ConvLstm4[0][0]
ConvLstm5 (ConvLSTM2D)	(None, 5, 48, 48, 8)	6944	batch_normalization_3[0][0]
batch_normalization_4 (BatchNor	(None, 5, 48, 48, 8)	32	ConvLstm5[0][0]
time_distributed_3 (TimeDistrib	(None, 5, 96, 96, 8)	0	batch_normalization_4[0][0]
ConvLstm6 (ConvLSTM2D)	(None, 96, 96, 4)	1744	time_distributed_3[0][0]
lambda (Lambda)	(None, 96, 96, 3)	0	gaussian_noise[0][0]
batch_normalization_5 (BatchNor	(None, 96, 96, 4)	16	ConvLstm6[0][0]
concatenate (Concatenate)	(None, 96, 96, 7)	0	lambda[0][0] batch_normalization_5[0][0]
conv2d (Conv2D)	(None, 96, 96, 3)	24	concatenate[0][0]

Total params: 45,832
Trainable params: 45,720
Non-trainable params: 112

Figure 7: Our Architecture

The input to the model is an array of size (96,96,15) which coincides with the dimension of the output of the generator function which as described above works on our data and produces the image ready to be fed to our model. Then the input is reshaped using the reshape and permute layers to get a dimension of (96,96,3,5).

After this, we get to a crucial step of adding the Gaussian noise. This is useful as it provides diversity to the dataset and hence makes our model more robust. Validation loss and the perceptual distance for the validation is in harmony with the training loss and perceptual distance for the training set when this Gaussian noise is added. We tried to build a model without it there was a significant difference between the two. More details of the difference in performance is explained later on. This gaussian noise is generally present in the real-life images and hence it can diversify our dataset. This helps in tackling overfitting during the training phase because of which the model performs well in the testing phase as well.

After this, another permute layer is added to get the desired input which is of (5,96,96,3) corresponding to the 5 input images each of size 96x96x3.

After this, we begin our seq2one architecture where we use the convLSTM layers to extract the information from every frame, and the LSTM part stores and processes this information for predicting our future frame. These convLSTM layers build our neural network and provide easy implementation of integrating LSTM and convolution layers as explained earlier.

Between each convLSTM layer, we have added a BatchNormalization layer which is used to normalize the input in between the neural net layers. This speeds up the training as well as predicting process. Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

One thing to note is that batch normalization works differently in the training and inference or testing phase. During training, the layer normalizes its output using the mean and standard deviation of the current batch of inputs. That is to say, for each channel being normalized, the layer returns gamma * (batch - mean(batch)) / sqrt(var(batch) + epsilon) + beta while in the testing phase the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. That is to say, it returns gamma * (batch - self.moving_mean) / sqrt(self.moving_var + epsilon) + beta.

Now, coming to the actual encoding and decoding part, we have used the MaxPooling2D and UpSampling2D layers inside the TimeDistributed layers to implement the autoencoder mechanism. TimeDistributed layer is a wrapper layer provided by Keras which allows applying a layer to every temporal slice of an input. In this, Every input should be at least 3D, and the dimension of index one of the first input will be considered to be the temporal dimension. MaxPooling2D is used for downsampling the input along with its spatial domain by taking the maximum value over a given window, in our case (2,2) for each channel of the input which is provided by the conLSTM layer to it. This works as the encoder part of the mechanism and we have used 2 encoding layers for this purpose. UpSampling2D is also a Keras provided layer and as an upsampling layer for 2D inputs.

It repeats the rows and columns of the data by size[0] and size[1] respectively. This is used as a decoding layer and provides its output to the convLSTM layers for processing. We have used two upsampling layers in our model. The need to use multiple convLSTM layers is to extract all the finer details of the image to work on

them as every layer can to get more and more information about the image.

Now as we know the autoencoder is a lossy mechanism and hence we lose some of the finer features and details in the background, therefore, in the end, we concatenate the last frame to the predicted frame to get those finer features and the lost details. After that, we perform a final convolution process on it and get an output of (96,96,3) which is a single colored image and is our predicted frame.

8 Performance

We have trained our model on the Catz dataset on 6418 training samples. Each input has 5 image frames of size 96x96x3. A single next frame is predicted from the previous frames. We have run 10 epochs in the part of training the model. For the first few epochs, the features of the predicted frames were noisy and blur. The color of the predicted frames is also quite different from the actual frame. Slowly as the epochs increase, the features of the predicted frame come to be accurate and also the color of the predicted frame starts to improve. By the end of 10^{th} epoch, we see a pretty decent result in terms of features and color of the image. (This model can be trained for more than 10 epochs also, to get much better results. As we did not use a GPU, it took us more time to run the model. Hence we stopped at 10 epochs). We are sure that the results will improve if the model is run beyond 10 epochs.

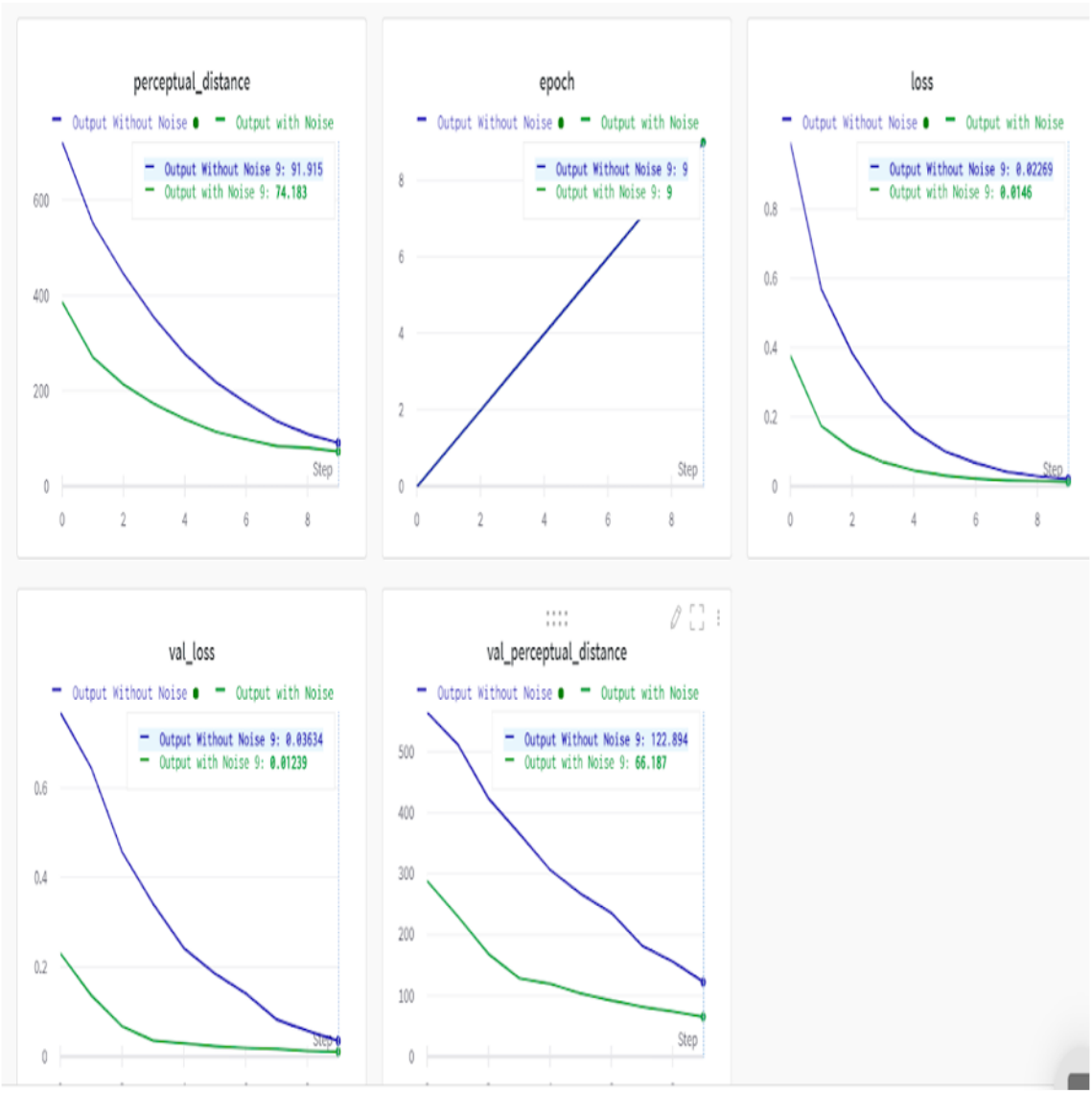


Figure 8: Loss and Perceptual Distance vs Epoch

As the epochs are passed, perceptual distance, validation perceptual distance, loss and validation loss are plotted in the above figure. We can see a gradual decrease in loss, validation loss, perceptual distance and validation perceptual distances. This

means that the model is getting better after every epoch. Remember, the less the perceptual distance, more the accuracy of the predicted frame.

There are 2 plots in the above figure and they are based on different models with a slight change. The blue model is the model without adding any noise to the data. The green model is the model with added Gaussian Noise Layer. We can see that the model which has added noise has lesser loss and perceptual distances, resulting in a better accuracy of the predicted frame. By this, we understood that adding noise to the training data increases the diversity of the images and the model can be trained much robust and hence better. We observe that by adding noise, our model prevents overfitting and gives better results in the validation phase as well.

Metric	**Model Without Noise (M1)**	**Model With Noise (M2)**
Loss	0.02269	0.0146
Validation Loss	0.03634	0.01239
Perceptual Distance (PD)	91	74
Validation Perceptual Distance	122	66

Table 2: Values after 10^{th} epoch

We see that $M1_{loss} < M1_{validationloss}$ and $M1_{pd} < M1_{validationpd}$. This signifies that M1 (model without noise) is more biased towards training and is performing worse on validation/test data.

Now, analysing M2 values, we see that $M2_{loss} > M2_{validationloss}$ and $M2_{pd} > M2_{validationpd}$. This signifies that M2 (model with added gaussian noise layer) performs better on validation/test data, and this is a good sign.

Hence, it is clear that:

- Model created is improving after every epoch.
- The loss values and perceptual distance values come out to be very less at the end of 10^{th} epoch (it can be improved though) and hence after 10 epochs, we get a decent result.
- From the performance metrics, it is confirmed that using a Gaussian noise layer improves performance of the model by increasing the diversity of training data and also making it robust towards unseen data.

```
Epoch 1/10
50/50 [==============================] - 325s 6s/step - loss: 0.3782 - perceptual_distance: 386.8341 - val_loss:
0.2315 - val_perceptual_distance: 288.2466
Epoch 2/10
50/50 [==============================] - 316s 6s/step - loss: 0.1754 - perceptual_distance: 271.0931 - val_loss:
0.1380 - val_perceptual_distance: 230.3666
Epoch 3/10
50/50 [==============================] - 314s 6s/step - loss: 0.1091 - perceptual_distance: 214.2604 - val_loss:
0.0684 - val_perceptual_distance: 168.7642
Epoch 4/10
50/50 [==============================] - 312s 6s/step - loss: 0.0716 - perceptual_distance: 173.6301 - val_loss:
0.0371 - val_perceptual_distance: 129.1250
Epoch 5/10
50/50 [==============================] - 311s 6s/step - loss: 0.0475 - perceptual_distance: 141.5240 - val_loss:
0.0308 - val_perceptual_distance: 120.2841
Epoch 6/10
50/50 [==============================] - 324s 6s/step - loss: 0.0320 - perceptual_distance: 115.3031 - val_loss:
0.0241 - val_perceptual_distance: 104.1135
Epoch 7/10
50/50 [==============================] - 314s 6s/step - loss: 0.0241 - perceptual_distance: 99.1752 - val_loss:
0.0204 - val_perceptual_distance: 92.8330
Epoch 8/10
50/50 [==============================] - 318s 6s/step - loss: 0.0186 - perceptual_distance: 85.4624 - val_loss:
0.0184 - val_perceptual_distance: 82.4138
Epoch 9/10
50/50 [==============================] - 318s 6s/step - loss: 0.0173 - perceptual_distance: 81.3966 - val_loss:
0.0142 - val_perceptual_distance: 74.7146
Epoch 10/10
50/50 [==============================] - 313s 6s/step - loss: 0.0146 - perceptual_distance: 74.1830 - val_loss:
0.0124 - val_perceptual_distance: 66.1871
```

Figure 9: Run Data of every epoch

9 Result

The output of the Seq2Seq model created is what we are seeing now. Just to revoke about the input and output, we feed a sequence of 5 images as input to the model (using a generator function) and what we get back is the next predicted frame. The output is being logged in wandb.ai page and here are the results from the page.

Figure 10: Few Output Samples

Figure 10 shows the output achieved for the best run (lowest loss and perceptual distance for 10 epochs) out of few of them we encountered.

Figure 11 shows the output of one batch of test data which contains 15 inputs, each having a sequence of 5 images as input.

We can see that the features of the predicted frame are very close to the actual next frame, and the color accuracy also seems decent. By these results we can conclude that a Sequence to Sequence model consisting of multiple ConvLSTM layers as encoders and decoders is a quite good model that can predict future frames of a movie given the present or past frames.

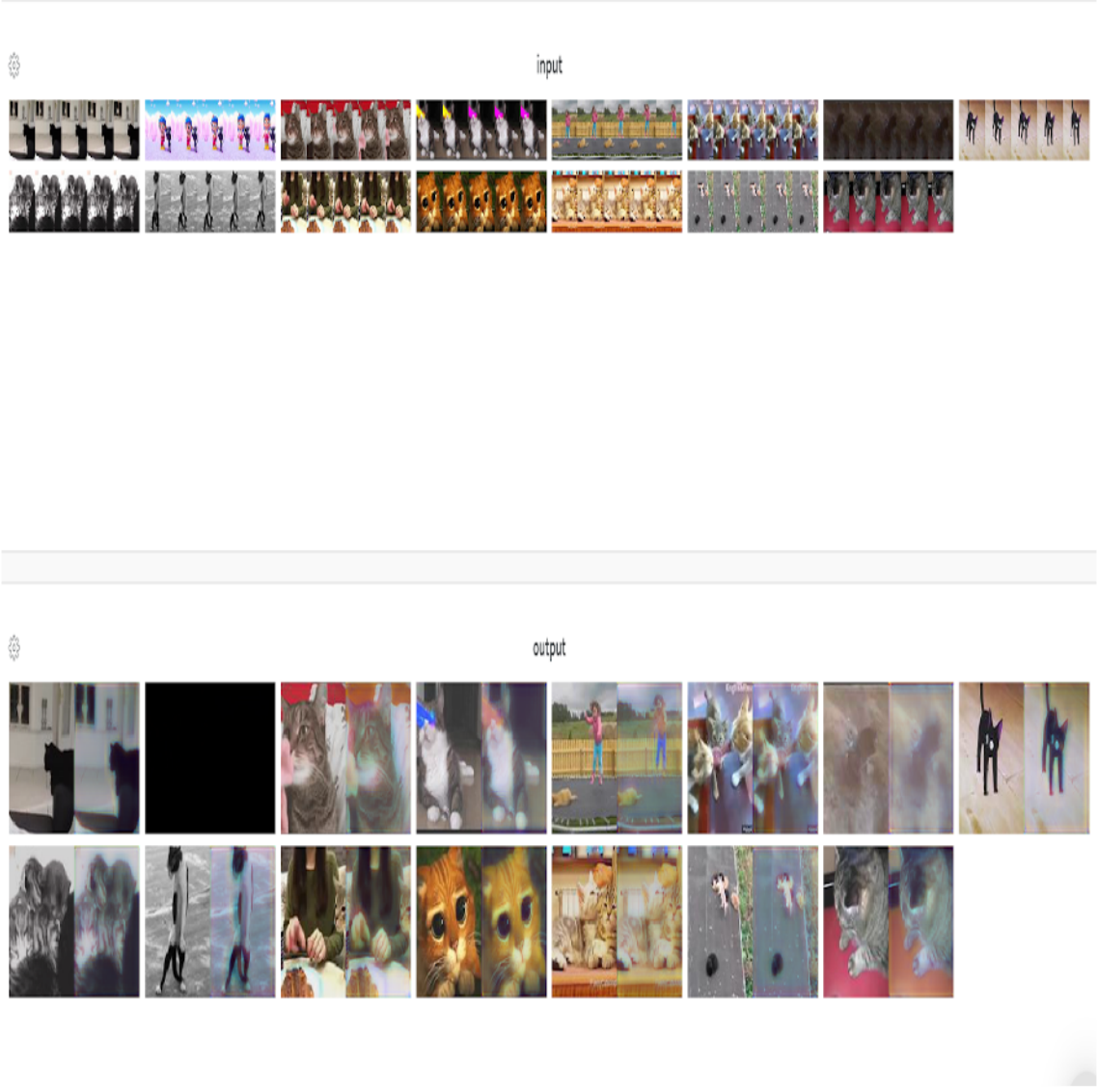


Figure 11: Output Screenshot from wandb.ai page

10 Drawbacks and Limitations

Although we have obtained a pretty decent result, there are few drawbacks in this model the way we dealt with the problem.

- The dataset itself has a limitation where we have gifs of cats, where each gif has only 6 frames in it. As the number of frames is less, we predict only the last frame (one frame) based on the first 5 frames.
- We lack a GPU and hence couldn't train for more epochs. But increasing the number of epochs would definitely increase the accuracy of the model.
- As the dataset is small where the input is just 5 frames and the output is a single frame. But the real-time data is much heavier in terms of number of input frames and output frames. So just having 7 ConvLSTM layers would be sufficient. A much more complex architecture is required for better results.
- The color, contrast and sharpness of the predicted images are somewhat bad compared to the actual images.

11 Future Work and Improvements

While the model works pretty well as we can see there are still some improvements that can be made to even improve on it.

- First of all, instead of predicting just a single frame, one can try to predict multiple frames. There can be two ways to do so.
  - One can be the autoregressive method where we predict the next frame and feed it back to the network for a number of n steps to produce n frame predictions.
  - The second could be predicting all future time steps in one go by having the number of ConvLSTM layers l be equal to the number of n steps. Thus, we can simply use the output from each decoder LSTM cell as our predictions.
- Working on a larger dataset and running the model for more epochs can result in better results as well. Provided one a decent GPU and processing capabilities, one can work on a much larger dataset and run the model for much more epochs, about 100 to 1000, to get even better results. This might also help in dealing with the color prediction problem.
- The predicted image can be processed by image processing techniques to enhance the outputs so it can be close to the actual video frame.
- Another improvement could be to include the perceptual distance metric in the loss function itself by providing weights to the current loss function and the perceptual distance. Consider the current loss function to be L and let us give it some weight α and then the perceptual loss will get the weight $1-\alpha$. Hence the loss function will become $\alpha * L + (1-\alpha) * PD$, where PD is perceptual distance.
- Another improvement can be to make a new model which would be a GAN model. Since GAN's are superion models in generating new images so it would be a much better model. One can use the model built here as the generator and then build a discriminator to identify the fake images from the real ones. Through this one can build a cycleGAN to generate the future frames.

12 Conclusion

Some key takeaways from this project are:

- Long Short-Term Memory models are used to tackle the dependency shortcoming of traditional Recurrent Neural Networks.
- Sequence-to-One architecture is a special of the Sequence-to-Sequence architecture that is used to map multiple inputs to a single output.
- Convolutional Long Short-Term Memory models which are similar to the normal Long Short-Term Memory models except that the former use convolution operations whereas the latter use matrix multiplication operations internally.

- Generative Adversarial Networks may be a better approach towards the problem of video frame predictions.
- The research in this domain may be used in Compression, Autonomous Vehicles, Unmanned Aerial Vehicles, etc.

13 Acknowledgement

We would like to thank **Dr. Snehasis Mukherjee** for the immense guidance he provided at each step of this endeavour. He provided recommendations and suggestions which were instrumental in making this project a reality in these challenging times.

References

[1] A. Graves. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

[2] Alex Sherstinsky, Deriving the recurrent neural network definition and rnn unrolling using signal processing, NeurIPS 2018, in: Critiquing and Correcting Trends in Machine Learning Workshop at Neural Information Processing Systems, vol. 31, 2018, Organizers: Benjamin Bloem-Reddy, Brooks Paige, Matt J. Kusner, Rich Caruana, Tom Rainforth, and Yee Whye Teh.

[3] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In NIPS, pages 3104–3112, 2014.

[4] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In CVPR, 2015.

[5] N. Elsayed, A. S. Maida and M. Bayoumi, "Reduced-Gate Convolutional LSTM Architecture for Next-Frame Video Prediction Using Predictive Coding," 2019 International Joint Conference on Neural Networks (IJCNN), 2019, pp. 1-9, doi: 10.1109/IJCNN.2019.8852480.

[6] N. Srivastava, E. Mansimov, and R. Salakhutdinov. Unsupervised learning of video representations using lstms. In ICML, 2015.

[7] P. Sambrekar and S. Chickerur, "Movie Frame Prediction Using Convolutional Long Short Term Memory," 2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT), 2019, pp. 1-5, doi: 10.1109/ICICICT46008.2019.8993289.

[8] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In ICML, pages 1310–1318, 2013.

[9] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," in Neural Computation, vol. 9, no. 8, pp. 1735-1780, 15 Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[10] Sherstinsky, Alex. (2020). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. Physica D: Nonlinear Phenomena. 404. 132306. 10.1016/j.physd.2019.132306.

[11] Shi, Xingjian Chen, Zhourong Wang, Hao Yeung, Dit-Yan Wong, Wai Kin WOO, Wang-chun. (2015). Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting.

[12] Sutskever, Ilya Vinyals, Oriol Le, Quoc. (2014). Sequence to Sequence Learning with Neural Networks. Advances in Neural Information Processing Systems.

[13] Y. Bengio, I. Goodfellow, and A. Courville. Deep Learning. Book in preparation for MIT Press, 2015.

[14] Y. Zhou, H. Dong and A. El Saddik, "Deep Learning in Next-Frame Prediction: A Benchmark Review," in IEEE Access, vol. 8, pp. 69273-69283, 2020, doi: 10.1109/ACCESS.2020.2987281.