Learning and Predicting Animations from Crowdsourcing

Anil Poriya
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14623
arp6763@rit.edu

Abstract— Software prototyping plays a vital role, especially when designers are designing interactive user interfaces and need to present their ideas in real time with visual elements and behavior. Applications of recurrent neural networks are widespread owing to their capabilities of learning long term dependencies. Their application can be extended to software prototyping. Apparition enables Wizard-of-Oz experience that allows for prototyping in real time because human computations are used to perform intelligent tasks that are currently beyond the realm of AI systems. It does this through crowd sourcing where remote workers collaborate to instantly create prototypes from verbal description or sketches provided by the designer. A major application of crowd sourcing is to perform certain intelligent tasks which the current AI systems are unable to perform using human computation. In software prototyping, crowds perform the task of generating working prototypes and visual behaviors just from the description provided by the designer to whom they are connected via the Internet. An excellent use case of machine learning would be to develop a system or model that can learn these action sequences or animations and predict them as required to be replayed automatically by Apparition. Learning these action sequences basically translates to learning a sequence of image frames and utilizing information from previous frames, possibly involving long-term dependencies. A Long Short Term Memory recurrent neural network works best for such cases owing to their USP of information persistence.

I. INTRODUCTION

Consider designing of user interfaces or presentation of software prototypes with some visual elements and behaviors. It involves speakers, prominently software designers presenting their ideas in real time and using some animations or graphics to better illustrate their designs to people. An ideal use case would be that designers focus only on presenting their ideas while the task of creating and presenting animations/graphics is delegated to other people. Apparition is a web tool that aims to serve this purpose. Apparition is a meteor based collaborative tool that involves remote workers connected to designers/presenters via the Internet and a microphone. The tasks of creating visual elements and behaviors of the objects that designers describe are distributed among these remote workers, called “wizards”. These workers then collaborate to complete the pending tasks using a commercial platform to generate instant prototypes that enable designers to better explain their ideas. However, the process of Crowdsourcing can get cumbersome and repetitive, especially when the designers tend to repeat their ideas to break down some complex designs. The workers would repeatedly have to create visual elements and behaviors each time designers illustrate their ideas to people. Additionally, the expense of paying these remote workers for completing the tasks can also be on the higher side. Thus it becomes vital somehow to eliminate the need for remote workers and all the expenses associate with it, to get the tasks completed. Using machine learning, it would be possible to create an intelligent model that can learn these animation sequences and recreate them automatically when required by the designers. The process of learning animation sequences from existing data and recreating these sequences given an input sequence is not a tricky one. A powerful model with information persistence can easily achieve this. The real challenge for these systems lies in predicting when to end the animation sequence or determining the length of predicted animation sequence.

The existing intelligent model in Apparition is that of Long Short Term Memory (LSTM) recurrent neural networks. These networks were chosen primarily due to their ability to remember information over long periods of time. Animation sequences involve a sequence of consecutive image frames. Image features from one frame may be required to predict the next frame in the sequence or even the future frames. Thus a LSTM network that can handle such time dependencies and remember important features from every image frame, is considered for predicting animation sequences in Apparition. The model is first fed with data extracted from the Apparition tool and trained to learn these sequences. It is then fed with an initial sequence of unseen data to predict the next sequences in the animation until end of animation is predicted. The predicted sequences are then transformed and replayed again as animation on the Apparition tool. The proposed model aims at successfully learning the end of animation given an input sequence. That is, if a model is trained with an animation sequence of 40 steps, the model should ideally predict the end of animation after 40 steps if it is given the first step of the animation sequence as an input.

Previous work on this project involves augmenting a session’s data using features of the animation objects, adding jitter to this data, training the model and subsequently predicting an animation sequence for an unseen input sequence. Although the model was successfully able to predict animation sequences for unseen data, it could not learn when to stop predicting the steps for current animation sequence and was thus constrained...
to only predict a fixed number of animation steps. The crux of
the project is really to enable the model in learning the stopping
step of the animation based on trained animation sequences.
We describe the data augmentation techniques used to train
the existing model and the need to switch to a sequence-
to-sequence based learning. The sequence-to-sequence based
model for this project is developed in Keras, a neural networks
API written in Python.

One of the major issues that we address in this report is that
of learning the end of animation sequence. The current working
model fails to learn, and hence predict this end of animation
sequence. This could occur due to inadequate amount of
training data that is provided to the model for learning which
again reiterates the need for substantial synthetic data for
the model to accurately learn the end of animation sequence.
Thus, the report focuses on working with various techniques
to generate synthetic data to better improve its learning of the
end sequence. Additionally, we try to limit our LSTM model
to the domain of single object straight line movements before
moving to more complex movements and multiple objects.
The latter half of the report focuses on building a sequence-
to-sequence deep learning model in Keras and the results of
running the same experiments with this Keras model. The
accuracy of LSTM depends on a multitude of factors. Some
of the important factors that can have a major impact on its
accuracy include size of the training data, loss functions used
in back propagation, number of hidden layers, and number of
epochs used to train the model. Apart from size of the training
data, rest of the factors can be worked with or tweaked to
obtain a trade off between accuracy and the time it takes to
train the model. However, one of the biggest caveats that we
try to address in this report is the size of the training data.
In this report we focus on building a system where the model
could be run entirely on synthetic data. We especially focus
on the problem of generating synthetic training data for straight
line movements, where each training example is strictly the
straight line movement of one object, possibly with a small
amount of perturbation or jitter.

II. RELATED WORK

To allow fluidity in software prototyping, more specifically
for interfaces that involve complicated behavior and interactive
elements, Walter S.Lasecki et al. [1] introduce “Apparition”,
a collaborative crowdsourcing tool for prototyping such inter-
active systems. The paper describes how Apparition enables
prototyping in near instant time using automatic sketch recog-
nition algorithms and paid microtask crowds. Apparition tries
to overcome the shortcomings of other similar systems that
cannot fully capture intricate details or interactive behavior of
various elements that a designer may envision. With sketching
and Wizard-of-Oz control functionalities, Apparition enables
easier prototyping as compared to other similar tools. As
designers or Apparition users describe their designs using
verbal descriptions or sketches, the self-coordinated crowd,
known as “workers”, and the automatic sketch algorithm help
visualize these designs using higher-fidelity representations.
The sketches are transformed into desired prototype elements
which allows to create a “living spec” and impart designer’s
intentions more clearly. The paper also describes the multi-
modality feature of Apparition that allows designers to verbally
express their desired behaviors which can expedite the process
of elements creation by the crowd. Additionally, the authors
describe the write-lock mechanism of Apparition that allows
different workers to coordinate with each other for doing
various tasks.

Haim Sak et al. [2] first reiterate the importance of Long
Short Term Memory recurrent neural networks over RNNs and
DNNs in acoustic modeling for speech recognition. Whilst
DNNs can provide only limited temporal modeling, RNNs are
unsuitable to handle longer term dependencies. Additionally,
the authors propose an alternative to the standard architecture
of LSTM, the Long Short-Term Memory Projected architecture
that outperforms these networks by efficient use of model
the long short term memory recurrent networks in detail. It
describes how LSTM models are superior to RNNs due to
their ability to handle/use long-term dependency. It also does
a step-by-step walk through for LSTM which enlists the details
of the “forget gate layer”, “input gate layer”, and the “output
layer”.

Joseph Lemley et al. [4] address the issues faced by neural
networks in low-data settings which reduce the generalization
capability of deep neural networks. They introduce the method
of smart augmentation that can generate augmented data
during the training process of a target network and help in
reducing the network loss. This smart augmentation process
is targeted to improve regularization and reduce over fitting of
data when training deep neural networks. Unlike manual
augmentation techniques where model attempts to learn simple
transformations, smart augmentation tries to learn the best
augmentation strategy. The proposed method [4] uses two
networks; Network A that generates data and Network B that
performs the desired task with the core idea being to learn
the best augmentation technique using A to train B. Joseph
Lemley et al. [4] also compare networks that use single or
multiple A networks to train network B versus training network
B alone. The results demonstrated that smart augmentation to
train network B worked better than training network B alone
(without smart augmentation). Additionally, using multiple A
networks to train network B gave slightly better results.

Terrance DeVries et al. [5] describe yet another approach to
data set augmentation irrespective of the domain change. The
approach adds noise, interpolation or extrapolation between
existing data points but in a learned feature space as opposed
to the input space. The approach uses a Long Short Term
Memory based sequence encoder to generate a feature space
thus making it effective for both static and sequential data
like time series. The transformations such as interpolation or
extrapolation are then applied in a feature space where
every example in the data set is projected. The experiments
performed demonstrated that the performance of learning al-
gorithms(supervised) can be improved through augmentation
by extrapolation between examples in feature space since
extrapolation generated useful synthetic data set as opposed
to noise and interpolation.

Hamid Palangi et al. [6] propose and develop a model to
address sentence embedding using Long Short Term Memory
recurrent neural networks. The authors propose LSTM due to
its memory capabilities that can capture and store long term
dependencies. For the purpose of sentence embedding, the
authors compare and contrast the LSTM approach with the
Paragraph Vector which is one of the well known methods. The proposed method of LSTM-RNN works on words within the given sequence of sentences and simultaneously maintains information about the sequence of these sentences, ultimately providing a semantic representation of the entire sequence of the sentence. For applications involving document retrieval or document matching, where the Paragraph Vector is generally used, the proposed method was seen to significantly outperform the Paragraph Vector.

The paper [7] explores the concept of Crowd memory to demonstrate that crowds can remember information over periods of time and can even get better with time.

III. TOOL: APPARITION

A. Crowdsourcing and the need for an Intelligent Model

For the LSTM recurrent neural networks that will be built to achieve the goals of this project, the data about object, its orientation and other features are obtained through an Apparition tool. Thus, it is important to have some background information about this meteor based tool. Apparition enables Wizard-of-Oz experience through crowdsourcing. A major application of crowdsourcing is to perform tasks that are currently beyond the reach of the state of the art AI systems. Lasecki et al. [1] describe how Apparition can enable designers to create interactive prototypes in only the time it takes to simply visualize these ideas. Apparition provides Wizard-of-Oz experience that allows for prototyping in real time because human computations are used to perform intelligent tasks that are currently beyond the realm of AI systems.

The Wizard-of-Oz experience is provided by remote workers who share the tasks on the same canvas of Apparition. These workers can be termed as wizards as they co-ordinate with each other to generate working prototypes and visual behaviors just from the description provided or from some of the strokes provided by a designer [1]. Additionally, these wizards provide this functionality in real-time making instant prototyping possible. This is useful especially during software prototyping of products that involve interactive user interfaces or animations. However, the designs initially developed by the designer are subjected to a lot of changes based on the feedback received from the developers. Additionally, over the course of presentation, the designer is bound to repeat some critical aspects of the design. Thus, it becomes necessary that the workers who coordinate to complete this task learn and remember some of the tasks that were previously described by the designer. Lasecki et al. [7] discuss crowd memory, an intelligent system approach where crowd agents can learn and remember over time. The workers who provide the Wizard-of-Oz experience are actually remote workers who are connected to the designers via the Internet. These workers then use the Apparition’s canvas to provide instant prototyping of the ideas that designer speaks by drawing animations on this canvas. To draw these animations, the crowd or the worker uses Amazon Mechanical Turk [8]. A lot of research is being done in learning the behavior of the crowds and the animations that are drawn on the canvas as the designer speaks. Since the designers are bound to repeat their ideas over the course of time, the task of recreating the same animations on the canvas becomes an exhaustive one. Additionally, the cost associated with it also increases. This necessitates an intelligent learning model that can learn the drawn animations and replay them as required. The problem for replaying the animation boils down to the prediction of correct image frames in sequence to recreate the animation [8]. Furthermore, the key challenge for replaying the animation is for the model to determine how many image frames does it need to predict before it can successfully say that the animation has ended. We build a learning model in Keras using Long Short Term Memory networks.

B. Apparition Architecture

Apparition as a tool is hosted on URL http://localhost:3000/ ?role=worker&workerId=some-worker&session=sessionId. It uses a JavaScript framework of Meteor for sharing information between clients [1]. It consists of a canvas with some already predefined interface elements and behaviors. Figure 1 shows a screenshot of Apparition’s canvas. Apart from Meteor, the LSTM model implemented using Keras and Apparition includes Python 2.7, MongoDB and Node JS.

C. Apparition Architecture

A general architecture of Apparition is described in Figure 2 [8]. Apparition’s general architecture involves recording training and testing animations on the canvas and then using this data in training the learning model. However, for the use case of predicting the end of animation, we take a slightly different approach in generating training and testing animations from Apparition.
• **Step 1: Record Animation**
  Based on the functionality provided on the canvas, two animations are recorded: train and test. For recording animation, an object is placed on the canvas and moved from left to right in forward direction. For train animation, entire length of the animation is recorded whereas for test animation, only partial forward movement is recorded.

• **Step 2: Feature Extraction**
  A FeatureSpace.py file is used to extract the animation features from the canvas and map it to numerical ones which will be used by the LSTM model to learn these animations. There are 45 features in total which are sub-grouped into four different types based on their functionality [8]. These include:
  1) 6 Geometrical Features
  2) 11 Object Shape Features
  3) 8 Movement Type Features
  4) 20 Single/Multiple Object Features

• **Step 3: Data Augmentation**
  The first two steps are used in our case only to extract the .pkl file which is the feature dictionary file created by FeatureSpace.py script [8]. This procedure is required only once to obtain the feature file since we will be using the same object of “rectangle” for all our experiments. The train and test animations that were recorded are discarded and we use data augmentation techniques to generate synthetic data. One of the major caveats for creating a learning model for such use case is the lack of training data that is available. Thus, it becomes important to generate enough training data, either through the tool or synthetically, to train the model for better results. The data augmentation techniques focus on generating translatory motions of the object from left to right at different y co-ordinates on the canvas. The process is detailed below in the Methodology section. The lstm_synthetic.py file is used for data augmentation.

• **Step 4: Training the Encoder-Decoder Model**
  Once the train and test data have been synthetically generated, they are fed to the training model in Keras using (N-1) examples of each animation where N is the number of steps in it. The goal of the model involves two tasks. One is to learn the actual animation movement. That is, it needs to predict the next step or the next frame in the animation sequence. If it receives animation frame at $T = t$, it needs to predict what would be the animation frame at $T = t + 1$. Additionally, the crux of the problem that we focus on in this report is the learning of end of animation. If the model is given an animation sequence of 10 steps and is given the first two steps as input, it needs to predict 8 additional frames before determining that the end of animation has reached. However, in reality, the problem of predicting the end of animation is a challenging one since the training data contains only a single step or sequence that indicates end of animation. The encoder-decoder model in Keras API is used to train the model. A single loss function is used throughout and a batch size of 1 is chosen. Although the batch size should be more in order to generalize the model, the small size of the train and test sets propelled us to settle to a batch size of 1. As the number of steps in train and test data and the number of animations increase, the batch size can be increased. Also, the model performs better as the epochs increase. For a animation of N steps and (N-1) examples, the end of animation step is predicted for each example and compared to where the end of animation actually should have been. When the end of animation has been predicted, the output data file is generated for each example which contains the predicted steps of the animation.

• **Step 5: Feature Reversal**
  The output file that is generated by the training model is then fed back to the Apparition tool. This is done through a FeaturesSpaceWithReversal.py file which creates the animation format from the output file using the feature dictionary file created in Step 2 [8]. The animation, generated with the name “anitron-”, can be directly seen in Apparition’s local host.

## IV. **Long Short Term Memory Networks**

Christopher Olah [3] in his paper iterates the importance of LSTM models in situations where information persistence is required. A lot of applications require information from the past in order to make current decisions. Traditional neural networks are feed forward networks with independent inputs and outputs. Thus, they do not have the capability of storing and using information from the past events. Recurrent neural networks are used to address this issue of information persistence. The reason for using RNNs is that it is able to use previous information that has transpired in the current task that is being performed. According to Olah [3], recurrent neural networks can be seen as a chain-like structure which is composed of same neural network in a sequential manner. It essentially performs the same tasks on every element of the sequence where the outputs are dependent on the previous outputs and the current inputs. Olah [3] also mentions that although recurrent neural networks have gained remarkable success in the various domains such as speech recognition, it has its own limitations when long term dependencies are involved. He gives a simple example of predicting the next word in a sentence. When the sentence is small, that is, when the information required from the past and the place where it is required is small, RNNs tend to perform good. However, when the gap is large, RNNs perform poorly as they cannot learn such long term dependencies. Olah [3] also introduces LSTM networks that are specifically designed to address these long term dependencies. LSTMs were introduced by Hochreiter and Schmidhuber [9] in 1997. LSTMs are basically special kind of RNNs that perform an excellent job in maintaining long term dependencies.

### A. **LSTM Working**

Olah [3] mentions that a key difference between RNNs and LSTMs is in the structure of their repeating modules. Figure [5] shows the internal structure of LSTM [3]. While RNNs have a single neural network layer such as a tanh layer, LSTMs generally have four layers. All these four layers
interact with other through some special means. The forget gate layer which is the sigmoid layer, is responsible for determining what information will be discarded from the LSTM cell. This is based on the output which is given by the sigmoid layer between 0 and 1. The input gate layer which is also a sigmoid layer, is responsible for determining the values to be updated in the existing LSTM cell. This is usually combined with a tanh layer to update the existing state. A vector of candidate values to be added to the state, is generated by the tanh layer. The values to be shown as output are finally decided using the output gate layer which performs this task in two steps. A sigmoid layer first determines parts of the cell state that will be used as output. Secondly, the values obtained from the tanh layer when the cell state is passed through it, is multiplied with sigmoid gate’s output to only output the parts that were decided upon in the first step. The ability of the LSTM model to store long term dependencies comes through a horizontal line which carries the data at the top of the model. At this level, the gates also provide their inputs based on what they have processed.

B. LSTM within Apparition

The structure of the LSTM model which was previously used in Apparition can be seen in Figure 3. It basically consists of LSTM cells, inputs, outputs and the targets. The experiments in this report are focused on the encoder-decoder model which has been implemented in Keras. Thus, we only focus on the input part of the model since Keras internally uses a LSTM model to implement its encoder and decoder network.

- Input: The animation data that is extracted from the Apparition session or which is synthetically generated is in the form of a sequence of animation steps where each step is offset from the previous step by some offset of time. Each animation step is represented as a vector containing 45 features. This sequence of animation steps basically illustrates the motion of the object along x-axis from left to right with constant y-axis value. The 45 features of each animation step are further subdivided into four sub-groups. The sub-groups and their descriptions are given below:
  - Geometrical Features: The first six features of any animation step are the geometrical features which describe the basic configurations of the object in terms of height, width and position along with the time offset.
    1) x co-ordinate
    2) y co-ordinate
    3) width
    4) height
    5) rotation angle
    6) time
  - Object Shape Features: The next 11 features of any animation step are the object shape features which describe the shape of the object that is being used. The currently supported shapes include line, circle, square, pentagon, octagon, triangle, rhombus, hexagon, ellipse and path.
  - Movement Type Features: The next 8 features include the movement type features which describe the current state of the object with respect to its movement. The 8 features include move, matrix, transpose, translate, restore, create, matrix and end.
  - Number of Object Features: These features basically indicate the number of objects that are contained in the animation. The number of object features is 25 meaning there can be a minimum of 1 and maximum of 25 object(s) in the animation.

C. Flow of LSTM Model

Figure 4 describes the LSTM workflow which was undertaken to implement the model in Keras API.
V. METHODOLOGY

A. Data Augmentation

The existing LSTM model uses a single animation drawn on the Apparition canvas for data augmentation by adding jitters/perturbations to it. The model is then trained on this augmented data to predict animation sequence for a given test sequence. Although real data is best to train the model with, recent works have shown that training a model with synthetic data can prove beneficial in improving the accuracy of the model. The proposed system intends to improve the existing learning model of LSTM by generating synthetic straight line movements for objects drawn on the Apparition canvas and then augmenting it by adding jitters/perturbations as in the previous model. The system currently focuses on single object animations before exploring the dimension of multiple object animations. To focus on generating synthetic straight line movements for single object, we first generate a dummy session with single object that would be used for all the future experiments. The single object under consideration in our experiments will be of type “rectangle”. To do this, we perform the following steps:-

1) Reset Meteor
2) Generate a dummy session “foo” in Apparition
3) Create a single “rectangle” object on the canvas using the pre-defined shapes provided by Apparition
4) Create a “train-” animation
5) Create a “test-” animation

After a dummy session has been generated, we then create a backup of this session using MongoDB’s mongodump utility. This backup can then be used to restore this session of single “rectangle” object with “train-” and “test-” animations on any MongoDB instance using “mongorestore” utility. The next phase in Data Augmentation involves generating a script that can reset the object on the canvas to its starting x co-ordinate and a y co-ordinate that is provided as a parameter to the script. The idea behind this script is to reset the object to different y positions on the canvas and then generate synthetic straight line movements along x axis at these y positions. Apart from resetting the object on canvas with appropriate x and y co-ordinate values, the script also uses parameters to generate n steps of synthetic data in t amount of time. This allows the script to generate synthetic translation motion for the object using different parameter values of y co-ordinate, number of steps and time. The parameters needed to run this script are illustrated below:

Run Command : python lstm_synthetic.py --which_data @val1 --x_cor @val2 --y_cor @val3 --t @val4 --n @val5 --session @val6

1) --which_data :- specify whether to run this script on train or test data. Defaults to train data
2) --x_cor :- specify the desired end of animation x co-ordinate used to calculate delta_x for each step
3) --y_cor :- specify the y co-coordinate of the object that will remain constant throughout synthetic data.
4) --t :- specify the desired time for completion of synthetic animation
5) --n :- specify the number of steps for completion of synthetic animation

B. Keras’s Encoder-Decoder Model

The experiments with single training and single testing sets did not yield the desired results as the model was able to learn the end of animation sequence only for initial combinations of m and n. The failure of the model could possibly be due to a limited training set which consisted of only a single animation. Thus another set of experiments were carried out with multiple training animations. However, the model was able to learn end of animation sequences only for the initial combinations just like in the previous experiments. This necessitated a move from PyTorch’s LSTM model to a deep learning based model that could better suit the purpose. The switch to a deep learning based model was made based on the recent trends that have shown the power of deep neural networks in achieving excellent results on difficult tasks. The project focuses on the encoder-decoder architecture that is used for sequence-to-sequence(Seq2Seq) mappings [10]. The general idea behind Seq2Seq learning to train models for inter-domain conversion of sequences. F. Chollet[10] describes an approach where a sequence, a simple sentence in English, is converted to a sequence in French. The task of learning the input-output sequences and performing inter-domain conversions can be done in several ways. The model described in the paper uses recurrent neural networks to achieve this task. The approach uses Keras’s encoder decoder architecture for cases where input sequences and output sequences differ in their lengths. This is an example of a general case which is more prominent while working with real data. Thus the model developed for our animation sequence closely follows the Seq2Seq model used in this approach. It consists of two layers of LSTM recurrent neural networks, “encoder” and “decoder”.

1) Encoder: An encoder is a layer of recurrent neural network that processes the input sequence. The “internal state” returned by the encoder is used by the decoder in the next step. The output generated by the encoder RNN is discarded and only the internal state, termed as the “context” for the decoder, is utilized.

2) Decoder: A decoder is also a layer of recurrent neural network that processes previous target sequence to predict the next target sequence. These target sequences are offset by one time step.

C. Seq2Seq model for Inter-Domain sequence conversion

Figure 5[10] shows the working of Keras’s encoder-decoder model on a simple use case of converting English sentences to French. F. Chollet[10] enlists the steps utilized by this model for conversion of sentences:

1) Input sequence is encoded into state vectors. Each state vector is a 3D vector.
2) A “start-of-sequence” character is used as a target sequence. This target sequence is of size 1.
3) Decoder is fed with this target sequence along with state vectors.
4) Next character in the sequence is predicted by the decoder as a vector.
5) The next character is then extracted using the max value.
6) Proceed to step 3 and repeat until end of sequence is encountered or character limit is reached.

D. Seq2Seq model for Predicting End of Animation

Input 1: Example E1 with first step
Input 2: Example E2 with first 2 steps
Input N-2: Example E(N-2) with first N-2 steps

Injecting Predictions until we predict "End of Animation"

Fig. 5: Sequence to Sequence Structure for Sentence Conversion

Fig. 6: Sequence to Sequence Structure for Predicting End of Animation

Figure 6 shows the working of Keras’s encoder-decoder model for our use case of predicting end of animation for a given input sequence of animation. The model closely follows the one that was used for English to French translation of sentences except for a few functional differences. The model was used to predict characters as a categorical class using the loss function of "categorical cross-entropy". However, due to the nature of our problem which is essentially a regression problem, we developed a model to predict a real-value output. The loss function used was that of "mean squared error" and the optimizer used was “adam” which is an algorithm for stochastic optimization. The aim of the model is to learn the "end" feature of the animation sequence which is encoded as a vector. The basic steps for a single animation are listed below:

1) Input animation examples are encoded into state vectors. Each state vector is a 3D vector.
2) An initial input animation sequence is used as a target sequence. This target sequence is of size 1.
3) Decoder is fed with this target sequence along with state vectors.
4) Next animation sequence for the animation is predicted by the decoder as a vector.
5) The index with the maximum value is extracted
6) Check if the index with the maximum value is the "end" feature index. If not, proceed to step 3 and repeat.

E. Data for Experiments with Keras

The data used for the encoder decoder model was generated from the synthetic scripts. The experiments were performed with single as well as multiple training animations. However, to use the animation data with Keras’s model, it was first converted to appropriate state vectors before feeding it to the model. We describe the data representation for a single animation of N steps. A single animation of N steps is split up into (N-K) examples where K can be determined by the user. Each example has increasing length of input sequences to be fed to the encoder and decoder. Thus, if first step which is encoded as state vector is fed to the encoder, the remaining (N-1) steps are fed to the decoder and if first two steps which are encoded as state vectors are fed to the encoder, remaining (N-2) steps are fed the decoder. This process is continued for the desired number of examples.

1) Numpy Array Representation: Similar to the approach described by F. Chollet, the input examples for a single animation were converted to three Numpy arrays, “encoder input data" (enc_inp_data), “decoder input data” (dec_inp_data), “decoder target data” (dec_tar_data):

- **enc_inp_data**: It is a three dimensional array which consists of animation examples fed to the encoder.
- **dec_inp_data**: It is a three dimensional array which consists of corresponding examples fed to the decoder.
- **dec_tar_data**: It is same as dec_inp_data but with a slight difference. It is offset by one timestep.

2) Dimensions of Input Examples: All the three Numpy arrays are of the shape(dimension_1, dimension_2, dimension_3). These dimensions are described below with respect to a single animation of N steps and (N-1) input examples:

- **dimension_1**: Number of input examples in a single animation, (N-1).
- **dimension_2**: Number of steps in the animation, N.
- **dimension_3**: Number of features in each input example. This dimension remains constant since there are 45 features in our animation examples.

3) Padding Steps in Input Examples: Considering a single animation of N steps and (N-1) examples, the first dimension is (N-1) for the number of input examples. All the input examples will have same second dimension which is the number of steps in the animation. Thus all of them will have a dimensions of N. The input examples are fed iteratively to the encoder and the decoder. An input example with first step to the encoder also has a dimension of N steps. Thus the input example is padded with zero vectors for the remaining (N-1) steps before feeding to the encoder. Similarly, for the same example, (N-1)
steps are fed to the decoder and the input example is padded with zero vector for the remaining one step before feeding to the decoder. The third dimension remains constant for every example which is the number of features(45).

VI. EXPERIMENTS WITH KERAS’S ENCODER DECODER MODEL

The experiments were targeted to first check if the “end” feature of animation was predicted. Additionally, if predicted, check if the predicted step for “end” feature aligns with the training end of animation. That is, if the end of animation was at Xth step for training example, check whether the predicted end of animation step is X. If not, how much offset does the predicted end of animation step has. All the experiments were conducted with a single rectangle object with a translatory motion from left to right with constant y coordinate of the object. The translatory steps are depicted through the steps of the animation. The steps are generated synthetically through the python scripts.

A. Encoder Decoder Parameters

The experiments were tried with multiple values of parameters that were used to train the model.

- **Epoch**: An epoch is generally referred to as “a pass” over the entire training set. During this pass over the data, the loss between the predicted outputs and targets is calculated and back propagated.

- **Batch**: A batch is generally referred to a set of X examples which are processed independently. The input data distribution is approximated by a batch. For better approximation, the batch size is increased.

- **Loss**: loss functions are generally used to calculate by how much the predicted outputs deviate from the target output. The loss calculated is then back propagated all the way to the input layer of the model, going through the hidden layers as well.

VII. RESULTS

The keras model was tested on simple animations of a single ‘rectangle’ object. Python scripts were used to generate synthetic motions of the object from left to right consisting of 40 steps. We initially trained and tested the model using only one animation of the object. Later on, multiple animations of the rectangle object at different y co-ordinates on the canvas were considered.

A. Single Animation Train Data

1) **Test Description**: For the single animation, we have a rectangle shaped object and generate left to right motion of the object synthetically. The animation generated is of 40 steps with y co-ordinate of the object as 90. The x co-ordinate was taken as 50 which is basically used to determine by how much the object will move in right direction at each step. This is obtained by dividing the x value by the number of steps. The animation was then split into 39 input examples using the 40 steps. The model is then trained on these 39 examples and then tested on the same data. A batch size of one was used since the number of steps in the animation are only 40. The model was first tried with 50 epochs where it failed to learn the end of animation for any of the input example. The model was then tried with 100 epochs where it was able to learn the end of animation.

![Fig. 7: Predicted vs Training End of Animation Steps](image-url)

![Fig. 8: Predicted Animation Movement](image-url)

- **Figures**:
  1) **Figure 7**: Predicted vs Training End of Animation Steps
  2) **Figure 8**: Predicted Animation Movement

- **Analysis**:
  Figure 7 shows the step numbers where the model was able to predict the end of animation for each input sequence length. The graph shows that the model performed well when tested on the same data as it was able to predict the step numbers almost similar to the training step numbers.
  Figure 8 shows the movement of the predicted animation sequence for an input sequence of length one. That is when input sequence of length one is provided to the model, what was the motion that was predicted by the model. The graph shows that the
model was able to almost reproduce the straight line motion of the object. The predicted path showed only slight deviation from the predicted path.

2) Test Description:: For the single animation, we have a rectangle shaped object and generate left to right motion of the object synthetically. The animation generated is of 40 steps with y co-ordinate of the object as 90 and x co-ordinate as 50. The animation was then split into 39 input examples using the 40 steps. The model is then trained on these 39 examples and then tested on a different test set which consists of 40 steps and y co-ordinate as 50. With batch size of one, the model was first trained and tested with 50 epochs and then with 100 epochs. The model was able to learn end of animation at 50 epochs as well as 100 epochs but produced slightly better results with 100 epochs.

- Fig. 9: Predicted vs Training End of Animation Steps

- Fig. 10: Predicted Animation Movement

- Analysis:
  Figure 9 shows that the model was able to predict end of animation for all the input sequence lengths. However, the pattern shows that the step numbers predicted by the model deviate from the training step numbers. However, the deviation in the step numbers was not huge for any of the input sequence length. When the model was trained with 100 epochs, Figure 11 shows that the deviation of the predicted step numbers from the training step numbers was reduced as opposed to the model with 50 epochs.
  Figure 10 shows the movement of the predicted animation sequence for an input sequence of length one. The graph shows that the animation movement predicted by the model initially had a straight motion in upward and right direction for few steps after which the motion went wayward and took a path in the
opposite direction, with some of the x co-ordinate values being negative. Figure [12] shows that when the model was run with 100 epochs, the predicted animation movement was quite similar to the one obtained in Figure [10]. However, with an increase in the number of epochs, the model was slightly better since it did not predict any negative values for the co-ordinates.

B. Train Data With Multiple Animations

1) Test Description:: The model was able to successfully learn the end of animation feature for all the input examples when single animation train data was considered. However, the predicted animation movements for the above experiments did not yield successful results in learning the straight line motions of the object. The model was then trained with multiple animations of the same rectangle object. For each animation, the horizontal motion of the rectangle was considered at different y co-ordinates.

1) Training Animation 1: 40 steps with x co-ordinate as 50 and y co-ordinate as 90
2) Training Animation 2: 40 steps with x co-ordinate as 50 and y co-ordinate as 130
3) Training Animation 3: 40 steps with x co-ordinate as 50 and y co-ordinate as 75
4) Testing Animation: 40 steps with x co-ordinate as 50 and y co-ordinate as 110

The model was trained with 50 as well as 100 epochs and a batch size of one.

- Figures:
  1) Figure [13] Predicted End of Animation Steps vs Training End of Animation Steps (50 epochs)
  2) Figure [14] Predicted Animation Movement (50 epochs)
  3) Figure [15] Predicted End of Animation Steps vs Training End of Animation Steps (100 epochs)

4) Figure [16] Predicted Animation Movement (100 epochs)

- Analysis:
  Figure [13] shows that although the model was able to predict end of animation for all the input sequence lengths, the step numbers predicted by the model deviate from the training step numbers by a considerable number initially. The difference subsides as the input sequence length increases. When the model was trained with 100 epochs, Figure [15] shows that the difference between the predicted step numbers and the training step numbers for end of animation is considerably low when compared to the model with 50 epochs.

Fig. 13: Predicted vs Training End of Animation Steps

Fig. 14: Predicted Animation Movement

Fig. 15: Predicted vs Training End of Animation Steps

Fig. 16: Predicted Animation Movement (100 epochs)
values being negative. Figure 16 shows that when the model was run with 100 epochs, the predicted animation movement was similar to the one obtained in Figure 14. However, with an increase in the number of epochs, the predicted animation path was slightly smoother when compared to the model with 50 epochs. It had a straight line motion initially in the downward and right direction after which the predicted path had a curvy motion towards right for few steps and then a straight and downward motion in the opposite direction towards the left.

VIII. Conclusion

The experiments were targeted at single object animations where the object follows a translatory motion from left to right. We successfully developed scripts in python that could easily generated this motion synthetically. The scripts accepted various parameters to provide flexibility for changing vertical position of the object and the total length of the input sequence. When using data augmentation techniques on previously built PyTorch’s LSTM model, the model was able to learn and predict the end of animation only for small sizes of train and test sets. The model failed to learn the end of animation for rest of the training examples and thus had to run for a predefined number of cycles to predict the animation sequence. However, we were successfully able to build a model based on the encoder-decoder architecture in Keras that could learn and predict the end of animation. The input to the model could include single animation or a set of multiple animations with horizontal motion. The model was successfully able to learn the “end of animation” behavior and performed well even on a completely different test set with horizontal motion. The model was however not able to completely reproduce the straight line motion of the object when tested on different data.

Creating this model required a thorough understanding of the LSTM model and its working. Additionally, it also required some extensive reading of the Keras, the API in which our model was built. Keras is a neural networks API which is written in python and supports both convolutional and recurrent networks.

IX. Future Work

Although we were able to address the stopping condition issue, in future, work needs to be done such that the model is able to reproduce the straight line motion of the object once trained. Additionally, the model was trained only with straight line motions of the object which were synthetically generated. The model in future would need to incorporate other non-synthetic and non-linear animations into its learning and be able to reproduce such animations. The model would also need to be extended to incorporate multiple objects having separate worker ids in Apparition.

ACKNOWLEDGMENT

We would like to extend our special gratitude to Dr. Christopher Homan for his unprecedented motivation, support and guidance throughout the project. Professor Homan provided his indispensable inputs regarding Apparition tool and the encoder-decoder architecture throughout the course of the project.

APPENDIX A

KEY TERMS FOR TENSORFLOW OR PYTORCH

REFERENCES

