Recurrent Neural Networks to predict translation behavior of objects in space

Shrishail Baligar
EECS, School of Engineering

Karnamohit Ranka
CCB, School of Natural Sciences

University of California, Merced

Introduction

Understanding behavior of any object, living or non-living is the key behind manipulating and making decisions about/around them.

Thus, to traverse and make sense of the surroundings, the first and the most obvious action to undertake is figuring patterns. Finding patterns in the surroundings has proven to be the essence of survival in evolution. Identifying patterns helps one manipulate and interact with their environment. Having explicit reasoning or some underlying hierarchical abstract understanding of patterns is essential to any inference tasks. Furthermore, these patterns are often time-dependent and sequential in nature. The data describing such patterns also depends on previous states of the system being considered, making it memory-dependent.

Recurrent neural network (RNN) and its derivatives have the inherent property of encoding history (albeit, in practice, for short time-periods) and its capability to work on sequential data makes it an excellent candidate model to encode behaviors which show sequential properties.

This project aims to train a RNN model which predicts the motion of points moving along a virtually 2-dimensional plane, while analyzing effects of memory-inclusion using Pearson correlation.

Motivation

The project aims at and expects equal contributions from both the authors, but for simplicity we'll use the terms *first* and *second* author.

The *first* author has research interests ingrained in an ambiguous but a core idea of understanding, predicting and accurately inferring behavior of objects. In this project, the goal is to predict the future positions of people walking in a passage. We call it the Canadian Passage of Humans. It is a video containing people walking in the Left or Right direction with some instances of having random walks too. The project will also accomplish a small module (Scene Behavior Understanding) of the *first* author's startup project.

The following serves as motivation for the *second* author’s focus on RNN methods: an important problem in the field of quantum chemistry is related to the time-dependent Schrodinger equation (TDSE). The quantum-mechanical wave function used to describe dynamics of the electrons in a molecular system can be propagated in time using TDSE to generate the electron density of that system at each time-step of propagation. Thus, the (time-dependent) electron density evolution is a temporal sequence. An alternative formulation, that propagates the electron density itself in time instead of the system’s electronic wave function, assumes that the Hamiltonian used in the Schrodinger equation is a functional of this density. Thus, a self-consistent solution of the TDSE in this formulation, where the density is also assumed to be

independent of its history of evolution (has no memory), is prone to erroneous calculation of the time-dependent density in the future. This formulation is known as the adiabatic approximation (adiabatic referring to the absence of memory). This method can break down for certain systems; however, RNNs can be useful in this case by helping to reconcile the calculations of this method with the more accurate, wave function-propagating methods (accurate because they account for memory, and the Hamiltonian is independent of the density). The functionals within the adiabatic approximation could be re-parametrized and modified to include memory effects *via* RNN models trained with data obtained from the accurate methods.

In the context of this project, to traverse and make sense of our surroundings, the first and the most obvious action to undertake is figuring patterns. Finding patterns in the surroundings has proven to be the essence of survival in evolution. Identifying patterns helps one manipulate and interact with their environment. Thus, having explicit reasoning or some underlying hierarchical abstract understanding of patterns is essential to any inference tasks.

This project involves training an RNN model which predicts the future positions of persons moving along a virtually 2-dimensional plane in the video[6].

Approach towards the problem

Initially, the project sought to use ideas from some recent papers[2][4] mentioned in the project proposal, but it was decided that the project would continue with a basic RNN architecture, to help establish a proof of concept regarding the performance of RNN models on customized sequential data.

The reasoning behind this decision is two-fold:

1. Complex RNN architectures and training techniques mentioned in some recent papers are hard to train and replicate.
2. A bottom-up approach: founding the architecture on the most basic forms of RNN and LSTM, building it up with just the necessary augmentations that the data summons.

The gist of the idea is to avoid building a Neural Turing Machine (NTM)[4] which is clumsy while testing and unwieldy during training; while this may not always be true, testing the flexibility of the basic hyperparameters of an RNN model in accommodating unexpected patterns can lead to a better architecture that makes the memory-calls utilized in a NTM computationally more efficient.

Two critical components of project

A) The Data a.k.a. Spaghetti Monster: The Fig.1 below gives an idea why we call the data a spaghetti monster. It is unstable and has stochastic patterns. However, there are some repetitive behavior patterns, on which our RNN model relies.

The biggest challenge which was initially perceived as trivial was to trace the position of people in the video using traditional computer vision (CV) techniques like Optical Flow and Gaussian Mixture Models. The detection performance was miserable. Hence, we ended up using Python's built-in mouse pointer-tracing library - Tkinter, to trace the position of people/objects in the video through mouse pointer.

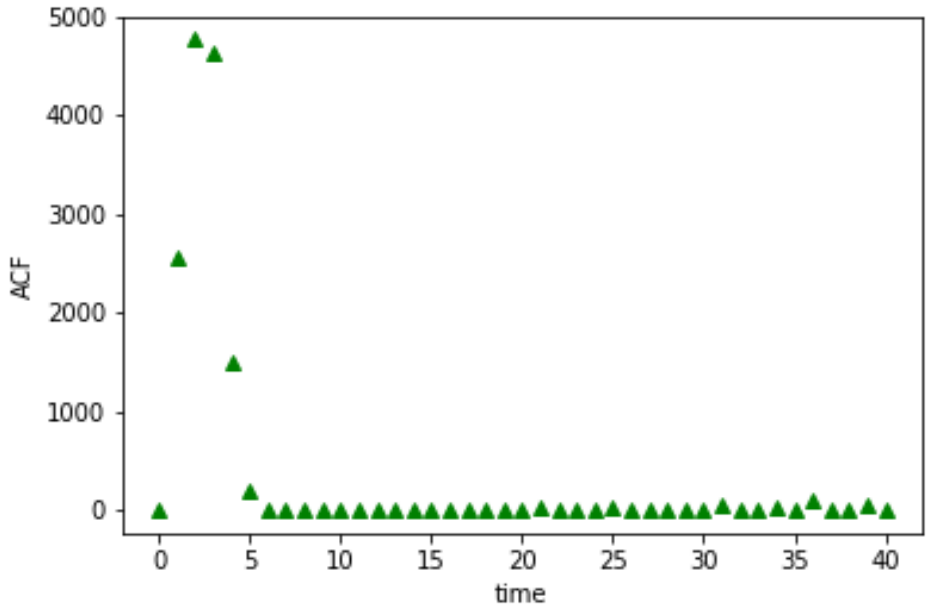

Fig. 1: ACF values of 5 old and new sequences (Ignore the initial ACF values which induced due to noise, they are for demonstrating how noisy ACF could be with bad data).

Since the data points were the x and y-coordinates of the video-screen window, there was no rationale in using these data points as inputs to the RNN directly because it would not be helpful in the downstream operations like working with the correlation coefficient and RNN cells. Therefore, it was decided to subtract each point by the previous point in the sequence so that we get the necessary number bounded in reasonable limit. This number is proportional to the number of steps the person takes in x- and y-coordinate directions at a time step t.

The advantage of this system for generating data-points is that the data is not expected to have anomalies of excessively significant differences/steps, unless there is an impossible outlier of a person making improbable maneuvers in the region of our interest. Because we are using hand traced data the human error is bound to be induced in the dataset.

An important point to note: even with the shortcomings, we expect our model to perform well on getting/predicting the trajectory of the persons' movements accurately, even if it performs badly in finer maneuver/position predictions. The evidence for the same can be seen in Fig.2

B) <u>The RNN:</u>

The coordinate-input set, X, has been defined as follows:

$$X = \{\vec{\chi}^1, \vec{\chi}^2, \vec{\chi}^3, \dots, \vec{\chi}^t, \dots, \vec{\chi}^\tau\} \quad \dots (1)$$

where,

$\vec{\chi}^t$ is the input/data vector in the sequence at time-step t;
τ is the total time-period for which the training input is provided.

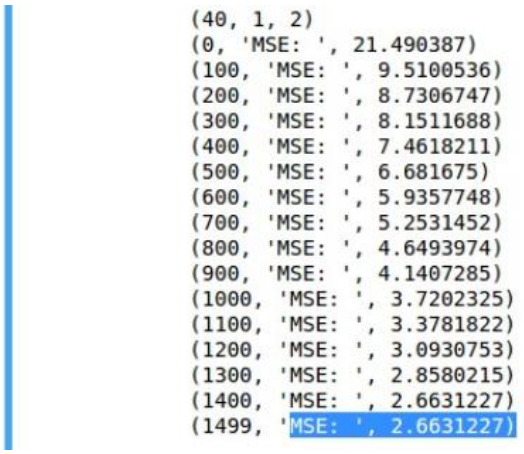

Fig. 2: Mean square error after 1500 iterations with minimalist architecture setting (without ACF).

Each of these input data-vectors represents a pair of displacements for all tagged agents in the space of the video at time-step t:

$$\chi_J^t = \{\Delta x_J^t, \Delta y_J^t\}$$

$$\vec{\chi}^t = \{\chi_1^t, \chi_2^t, \dots, \chi_J^t, \dots, \chi_N^t\} \quad \dots (2)$$

where,

Δx_J^t is the displacement of agent J along axis x at time t;
N is the total number of tagged agents.

The memory-accounting expression being used for this RNN model calculates the auto-correlation function (ACF) between any two sequences of the same length at time-step t_4 within a time-series associated with an agent, $\vec{A}^{t_4}(\tilde{\vec{\chi}}_{t_2}^{t_1}, \tilde{\vec{\chi}}_{t_4}^{t_3})$; it is a vector of length $2N$ at each "output" time-step (time-step at which the prediction is made) and also serves as input along with the displacements. The individual components of the ACF vector are defined in terms of short sequences of the input vectors between time-steps t and s ($t > s$), $\tilde{\vec{\chi}}_s^t$:

$$\tilde{\vec{\chi}}_s^t = \{\vec{\chi}^s, \vec{\chi}^{s+1}, \vec{\chi}^{s+2}, \dots, \vec{\chi}^{t-2}, \vec{\chi}^{t-1}, \vec{\chi}^t\}$$

$$A_{\Delta x,J}^{t_2}(\tilde{\vec{\chi}}_{t_1}^{t_1+b}, \tilde{\vec{\chi}}_{t_2}^{t_2+b}) = \left(\sum_{j\in[t_1,t_1+b], i\in[t_2,t_2+b]} \left(\Delta x_J^j - \mu_{\Delta x}(t_1; b, J)\right) \cdot \left(\Delta x_J^i - \mu_{\Delta x}(t_2; b, J)\right) \right) \quad \dots (3)$$

$$\vec{A}^{t_2}(\tilde{\vec{\chi}}_{t_1}^{t_1+b}, \tilde{\vec{\chi}}_{t_2}^{t_2+b}) = \left\{A_{\Delta x,1}^{t_2}, A_{\Delta y,1}^{t_2}, \dots, A_{\Delta x,J}^{t_2}, A_{\Delta y,J}^{t_2}, \dots, A_{\Delta x,N}^{t_2}, A_{\Delta y,N}^{t_2}\right\}$$

where,

$A^{t_2}_{\Delta x,J}\left(\tilde{\vec{\chi}}^{t_1+b}_{t_1}, \tilde{\vec{\chi}}^{t_2+b}_{t_2}\right)$ is the auto-correlation function of the inputs along the x-axis, Δx, for the tagged agent J at time-step t_2;
b is the batch-size, or the length of the sequences for which the ACF is being calculated;
$\mu_{\Delta x}(t)$ is the mean (displacement, Δx) of the batch sequence;
$\sigma_{\Delta x}(t)$ is the variance of the batch sequence.

The prediction of the RNN model is represented by the set Y as follows:

$$Y = \{\vec{v}^{b+1}, \vec{v}^{2b+1}, \dots, \vec{v}^{sb+1}, \vec{v}^{(s+1)b+1}, \dots, \vec{v}^{(\lfloor \tau/b \rfloor - 1)b+1}, \vec{v}^{\lfloor \tau/b \rfloor b+1}\}$$

$$\vec{v}^{t} = \{\mathrm{v}^{t}_{1}, \mathrm{v}^{t}_{2}, \dots, \mathrm{v}^{t}_{J}, \dots, \mathrm{v}^{t}_{N}\}$$

$$\mathrm{v}^{t}_{J} = \left\{\Delta x^{t,pred.}_{J}, \Delta y^{t,pred.}_{J}\right\}$$

where,

$\vec{v}^{t}$ is the prediction vector in the sequence at time t, corresponding to the data-point $\vec{\chi}^{t}$;
$\Delta x^{t,pred.}_{J}$ is the *predicted* displacement of agent J along axis x at time t;

$\vec{v}^{t}$ is defined in the same manner as $\vec{\chi}^{t}$ above, except now it is the RNN model that produces this quantity as the coordinate-output/displacement prediction.

The loss function used to drive optimization (towards the minimum value) is simply the mean square error (MSE), $L(X, Y, t)$, in the prediction for an input batch:

$$L(X, Y, t) = \frac{\sum_{i\in\{x,y\}, J\in[1,2,\dots,N], t\in[t_{ini}, t_{fin}]} \left(\Delta i^{t,pred.}_{J} - \Delta i^{t}_{J}\right)^2}{2 \cdot N \cdot \left(t_{ini} - t_{fin}\right)}$$

Given the above definitions, the input for the RNN model at time t is a vector of length $4N$— the first $2N$ components being the discplacements along the 2 axes for the tagged agents, and the following $2N$ components comprised of the ACFs of the tagged agents' histories along the two axes. The output/prediction is a vector of length $2N$.

The architecture of this RNN model aims to use TensorFlow (TF)[8] implementations of the following:

1. RNN cell with the ReLU activation unit for multiple neurons in a single hidden layer, wrapped with **OutputProjectionWrapper()**;
2. Dynamic unrolling with the TF class **dynamic_rnn()**;
3. ACF vectors passed to the RNN cell through tanh activation units;
4. Adam optimization algorithm[7].

Three-phase division of the project

Phase 1:

The first phase of the project was to understand how amenable the problem was with the minimum RNN configurations. The input of the RNN consists of a vector containing three values – the last Δx and Δy values in the sequence window, and the ACF of that sequence window. This vector of three values when fed to RNNs with a single layer, it gives an output which should be compared to the ground-truth in the next sequence window. The results shown in Fig. 2 are from the training

stage of this phase of the project. For Phase 1, $\tau = 200$ in eq. (1), $N = 1$ in eq. (2), $b = 5$, and $t_2 = (t_1 + b + 1)$ in eq. (3).

Phase 2:

In the second phase, the input will be a vector with the whole sequence along with the ACF (same as Phase 1). The second phase has been implemented with a single hidden layer within the basic RNN cell implementation of TensorFlow. The values of $b = 20$ and $N = 1$ were chosen due to the empirical results obtained through the implementation of phase 1, and convenience, respectively. Testing results were obtained for phase 2.

Phase 3:

In Phase 3, the input will be the same as the input in Phase 2, except we add a bias matrix which contains the positions of moving objects around our RNN (here, our agent/person). We will also implement a graph which will compare the ground-truth vs. predicted trajectory of the person in the video. Moreover, storing the ACFs from the past in an explicit memory, which can be used later to bias the hidden state of the RNN is expected to yield better results. Explicit memory techniques applied in past work[1][2] have been shown to work well in sequence generation tasks.

Results and Discussion

Table 1 displays the results obtained from tests of the phase 2 implementation of the current model.

Phase 2 (# of Inputs = 2*20 (window-size=20), # of outputs = 2)				
MSE Values (# of Iterations = 5000) **(without ACF)**				
	RNN (h=100)	RNN (h=256)	LSTM (h=100)	LSTM (h=256)
Training	0.0960318	0.0960326	0.0960317	0.0960319
Testing	0.43588	0.570859	0.435914	0.436828
MSE Values. (# of Iterations = 5000) **(with ACF)**				
	RNN (h=100)		RNN (h=256)	
Training	0.0832384		0.0796035	
Testing	0.251382		0.214714	

Table 1: Comparison of the current architecture with basic LSTM and basic RNN cells (RNN cells with and without ACF)

The LSTM and RNN cells have been employed as implemented in TensorFlow (RNN with a single "hidden" layer, and LSTM with a single "hidden" layer and 4 gates). The total number of parameters for either of these implementations is 10200 (for RNN cells with 100 neurons, 4x for LSTM cells with the same number of neurons) and 26122 (for RNN cells with 256 neurons, 4x for LSTM cells). The total number of input displacement vectors (of size 2 for x- and y-axes) used for training the current "ACF-RNN" model is 5000, whereas the size of the dataset on which the current model has been tested is 400.

It is to be noted that the testing dataset and training dataset belong to different agents with varying levels of noise in the respective trajectories.

During the experiments, some non-intuitive results were obtained initially when trying to set a benchmark (without ACF) to compare the proposed ACF-RNN. For instance, LSTM, despite of its larger hidden state, gave no significant improvement over its RNN counterpart (also without ACF).

We draw some conclusions, specific to the current problem, regarding why some combinations of the hyper-parameters work well:

The premise behind LSTM comparison –

LSTMs came into inception with the motivation to counter the vanishing/exploding gradients by adding the "+" gates in the RNN cell architecture as necessary [9]. The LSTM architecture was expected to be a suitable candidate for the current problem and would serve as benchmark for the current model, given its ability to account for memory effects and the transitory effect of input sequences. However, as reflected in Table 1, there weren't any major improvements compared to the basic RNN architecture. Underperformance of the basic LSTM architecture exposes its peculiarity, contradicting our understanding of LSTM architectures' hidden states. Another motivation was also to approach the problem from bottom-up by using the bare-bones RNN architecture and augment it with just the necessary memory indicators (e.g., ACF), and then compare it with the state-of-the-art, i.e., LSTM architectures.

LSTM's larger/longer hidden state does not guarantee success but providing variance information does –

As mentioned above, we needed a time-series model which would consider the history of sequences. A character peculiar to our data is that the future positions have dependency on roughly the following: variance of the sequence window and displacements in the recent past. LSTM's short term memory can take into account the second dependency, where recent points in the sequences come into play while predicting a future displacement. However, it does not account for the variance in the past displacements or positions, as the results indicate. We needed a model which would use variance as well as history in making the prediction (roughly, it divides the sequences in phases of "noisiness"). This is most likely the reason why an ACF-assisted RNN architecture works. ACF-RNN seems to understand and needs just enough history (hidden state) and behavior (variance-ACF) to make reliable predictions.

The model doesn't need a lot of data to train –

To clarify, we don't claim that our model needs little data to train. It's asserted in the broader context of how much data deep neural networks need these days. Hundreds of thousands to Millions of data points for is norm. But that is necessary from that particular use-case's POV. Those models need that much amount of data to approximate any unseen data, especially images. The variance in the possible inputs in *very* high in those cases. However, in our case the variance can be quantified by just looking at the training data. There is another aspect in our input, i.e., variance in the input sequence; however, the ACF seems to take care of that aspect and generalizes the data-points reasonably well.

Hence, there is a limit after which the number of data points would plateau the training and test accuracy, because of the low variance in the position and the ACF's capability to take care of the sequential variance.

Future Direction

Although this project does not (yet) provide state-of-art results, it is a stepping stone towards a spectrum of possibilities which involves complex behavior understanding and planning. It would also be quite reasonable to say that if this particular area of research on RNNs undergoes significant progress, it potentially solves the most critical problem in self-driving and other autonomous systems, i.e. Understanding the Uncertainty in the Environment.

The computation time for every time step in the RNN could be customized based on its input which gives better results, specifically to the task that's undertaken in this project. Very recent work [3] in the Adaptive Computation Time shows evidence of how RNNs internal time step could be differentiated using attention mechanism.

Dividing the maneuvers into sub-divisions and tackling the problem of inferring positions in the context of more complex motion of objects could be approached using hierarchical layers of RNNs. Examples of such problems like – the chaotic Indian traffic condition or in any other system of complex motions, are suitable candidate problems for such hierarchical RNN models. Since this write-up is an intermediary before the final project report, we write the above problem in the *Future Direction* section, however, it could be Phase-4 if we have time to implement by the final deadline.

Reinforcement earning – a branch of learning algorithms – can be applied to make decisions in chaotic environments [6] mentioned above. Agents with the capability of anticipating the positions of objects around it in the future, is far more well equipped to use its Reinforcement Learning models than the agents without it. Such agents are essentially very crude models of Driverless Vehicles. As a group we look forward to using this ideas in our respective future Research works.

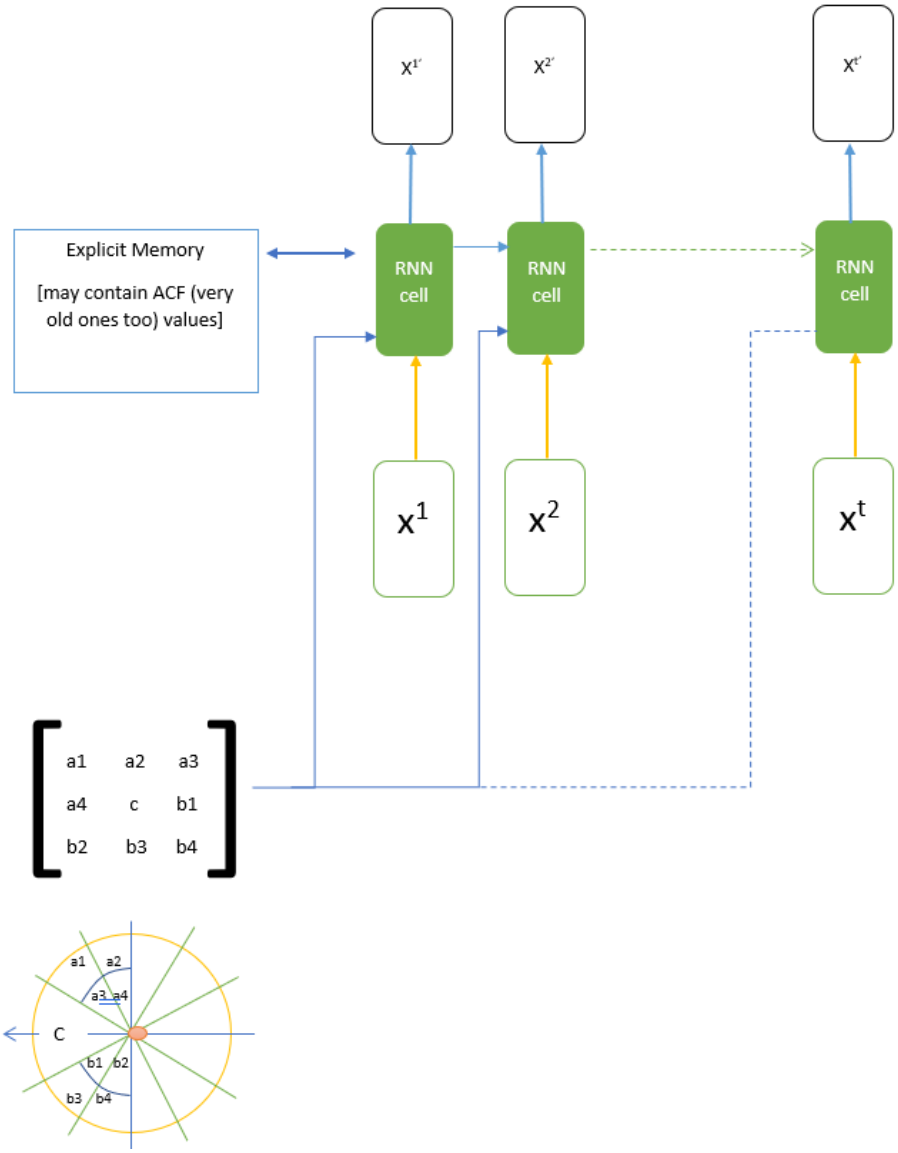

Fig. 3: Schematic of Phase 3.

References

[1] Learning Efficient Algorithms with Hierarchical Attentive Memory (Marcin, Caro; 2016)
[2] Memory Networks (Weston & Chopra *et al*; 2015)
[3] Adaptive Computation Time for Recurrent Neural Networks (Alex Graves; 2017)
[4] Neural Turing Machines (Alex Graves; 2014)
[5] Recurrent neural network regularization (Zaremba, Sutskever; 2015)
[6] https://www.dropbox.com/s/o9qn8he8urvvx7r/cph.mp4?dl=0
[7] Adam: A Method for Stochastic Optimization (Kingma, Ba *et al*; 2015)
[8] https://www.tensorflow.org/about/bib
[9] http://proceedings.mlr.press/v37/jozefowicz15.pdf