Trajectory prediction for autonomous vehicle using CNN-RNN architecture

Giulio Bazzanti, Niccolò Biondi
Dipartimento di Ingegneria dell'Informazione (DINFO)
University of Florence - Florence, Italy
giulio.bazzanti@stud.unifi.it, niccolo.biondi1@stud.unifi.it

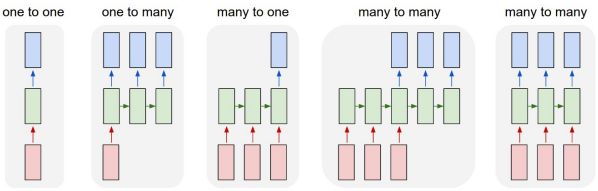
***Abstract*—Today autonomous driving is a challenge, multiple automakers have in development or in production automated driving systems that offer freeway-pilot functions. In this paper we propose a machine learning approach able to generate trajectories starting from video frames acquired from a video camera placed over a remote-controlled car. In particular, we study an encoder-decoder architecture that processes the videos frames and generate a sequence of points, with the use of a LSTM. We develop three different models: *single-frame* model, with one RGB frame in input; *multiple-frame* model with five RGB frames concatenated each others; *depth* model with the RGB and depth frame as the input of the network. We show that the *multiple-frame* model reports the best results both in the evaluation metric (Euclidean Distance) and in the quality of the predicted trajectories.**

I. Introduction

The autonomous drive is a topic that is becoming very important today. An automatic car is a automatic vehicle equipped by a variety of sensors to perceive their surroundings, such as radar, lidar, sonar or GPS, that can satisfy the main transport capabilities of a traditional car with or without the human intervention [1][2]. During the development of this type of technology five different autonomous driving levels were identified: from level 0 that identifies "no autonomous driving" to level 5 where the "complete car automation" is obtained. Today autonomous vehicles are part of level 2-3. In fact, this type of technology is used for "assistant parking" or for assisted braking in the event of a sudden crossing of pedestrians. The continuous development and improvement in this area will allow the driver to leave the driving completely to the car, making everything safer. Just think of the cases of accidents caused by distraction or sleep strokes that could be foiled thanks to this type of technology [3].

The purpose of this project was to emulate a sort of small-scale autonomous guide. Through the use of a remote-controlled machine and a camera, it was possible to find the data that was used in the experiments described in the next part of this paper. At this scope was used a Deep Network with which, through the processing of frames, we are able to predict sequences of points indicating the trajectory that the remote-controlled machine should follow. The Deep Network used is composed of two parts: a part of Encoder realized through a Convolutional Neural Network (CNN) with which the features are extracted from the various frames obtained by the camera on the remote-controlled machine and a part of Decoder. This second part was realized through a Long-Short-Term-Memory (LSTM) which, thanks to the particular memory maintenance feature of past iterations, allowed the prediction of sequences of future points starting from a present point.

Figure 1: LSTM models



This type of the LSTM model used in this project is called One-to-many: through a single input, provided by the CNN, the LSTM can predict a sequence of points which represent the trajectory to follow.

In Figure 1 is possible to see the various *LSTM* possible models.

The network was build using *PyTorch* [4]. This is an open source machine learning library based on the library of *Torch*, used for applications like computer vision and natural language processing.

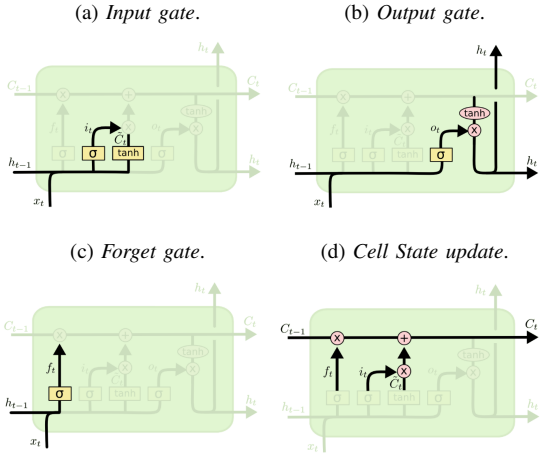
The *Torch* library, used in *PyTorch*, uses data structures called *tensor*, objects similar to arrays, but which make it possible to use *GPU* for the purposes of normal mathematical operations [5].

II. Approach

A. LSTM

Long-Short-Term-Memory (LSTM) take part of a neural network family that takes name of *Recurrent Neural Network* (*RNNs*). The RNNs are a family of neural network for processing sequential values $x^{(1)}, \ldots, x^{(\tau)}$, for a certain time value τ. To realize this type of networks, was introduced one interesting idea: sharing parameters across different parts of a model. Parameters sharing makes it possible to extends and apply the model to examples of different forms and generalize across them. The parameters sharing principle is used in RNNs in the following way: each member of the output is a function of the previous member of the output. Each member of the output is produced using the same update rule applied to the previous outputs. This recurrent formulation results in

Figure 2: The Description of the three gates that compose the *LSTM*: in figure *a* is represent the input gate, in figure *b* the forget gate and in figure *c* the output gate. In the last figure, figure *d*, is described the combination of forget and input gate.



the sharing of parameters through a very deep computational graph.

As of this writing, we focus on another type of model called *gatedRNN*. This type of model include the LSTM network and there is resolved the vanishing gradient problem, typical in RNNs [6].

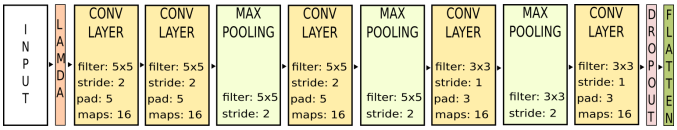
The main characteristic of the LSTM is remembering the information for long time period. This type of neural networks have four layer that interact among them. The key idea behind this type of model is the *cell state*: the LSTM can add or remove information from this cell state according to particular structures called *gates*. This gates are the best method to let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation [7]. The outputs of the sigmoid between zero and one determinate how much of each component should be let through. We describe now the three different type of gates with reference to the Figure 2.

The *forget gate* has the aim to decide what information throw away from the cell state according to the new input, indicated with x_t, and the previous output of the LSTM Cell called *hidden state* and indicate with the symbol h_{t-1}. This decision is made with a sigmoid layer called *"forget gate layer"*. The output of this layer is a number between 0 and 1 for each number in the cell state at the previous time step $t-1$, indicated with C_{t-1}. The representation of this operation is describe in Figure 2c and with the following formula:

$$f_t = \sigma(W_\mathrm{f} \times [h_{t-1}, x_t] + b_f) \tag{1}$$

The *input gate* is shown in Figure 2a, and decide which information we're going to store in the new long-term memory C_t (cell state). Like the forget gate, the input gate works with the current input x_t and the previous output at time $t-1$ of

Figure 3: The model architecture. Different transformations are depicted in different colors.



the LSTM Cell h_{t-1} and use a sigmoid layer, called *"input gate layer"*, to decide witch value we'll be update.

$$i_t = \sigma(W_\mathrm{i} \times [h_{t-1}, x_t] + b_i) \tag{2}$$

Next, it calculate a vector that is the output of a tanh layer. This vector, indicated with $\tilde{C}_t$, represent the candidate value that will be added to the new cell state C_t.

$$\tilde{C}_t = \tanh(W_\mathrm{C} \times [h_{t-1}, x_t] + b_C) \tag{3}$$

At the end, as we can see in Figure 2d, these two values, i_t (2) and $\tilde{C}_t$ (3), are combined into the new cell state C_t. The old cell state C_{t-1} is multiply with the forget gate f_t (1), then is added $i_t \odot \tilde{C}_t$.

$$C_t = f_t \odot C_t + i_t \odot \tilde{C}_t \tag{4}$$

The last gate, described in Figure 2b and called *output gate*, decide the new output h_t wich will be based from the cell state. First of all a sigmoid layer compute, like in equation 2, the output o_t. Then, this vector, is combined with the result of a tanh layer. This layer filter the cell state C_t to push the value to be between -1 and 1.

$$o_t = \sigma(W_\mathrm{o} \times [h_{t-1}, x_t] + b_o) \tag{5}$$

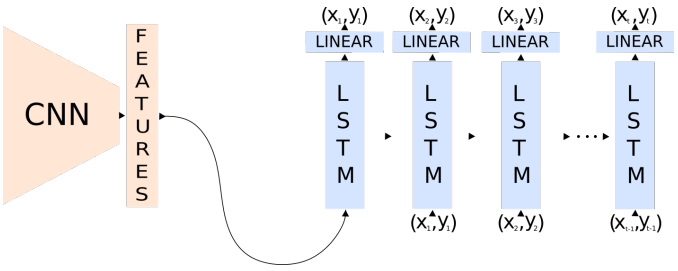
$$h_t = o_t \odot \tanh C_t \tag{6}$$

where b, and W are respectively the bias and recurrent weight into the LSTM cell, different for each gate. The symbol σ represents the non-linear sigmoid function and the symbol $\odot$ represents the product with the gate value [8][9].

B. CNN

The *Convolutional Neural Network* (*CNN*) is a simple network dedicated to extract the features from the video frame in input. The extracted features will be used to the LSTM to predict the coordinates that will compose the entire sequence. This encoder is composed by three convolution layers with kernel 5×5 stride 2 and two convolution layer with kernel 3x3 with stride 1. Also the second and the third convolution layers are followed by max-pooling layer with kernel 5×5 and stride 2 and the fourth convolution layer is followed by max-pooling with kernel 3×3 and stride 2. In the end we have a dropout layer followed by a fully-connected layer witch brings the features vector to have the same dimension of the hidden cell used in the LSTM. All the layers use Exponential Linear

Figure 4: The Deep Network architecture used in the project.



Units (ELUs) as activation function. Other information related to the network structure are in Figure 3.

C. CNN-LSTM structure

We use a two layer LSTM model for generating a sequence of points like is describe in [9]. We employ the LSTM to "decode" a visual feature vector representing the frame video with the aim to generate a sequence of coordinates.

The first step is to create a features vector that describe the frame video. For this is used a CNN, describe in Section II-B, and, unlike the methods read in [9], the net is not pre-trained, but train with the LSTM together. The aim of this choice was to check if the LSTM would have made good predictions without a trained CNN. We extract the output of the linear layer, Figure 3, from every frame video. The resulting features vector forms the input to the LSTM layer. Notice that we use two LSTM layers, so the hidden state created from the first layer became the input of the second LSTM layer. The entire structure is described in Figure 4.

III. Model Configuration

A. Loss Functions

We use the function *Mean-Square-Error (MSE)* as the model's loss function. The aim of our model is to generate a sequence of coordinates similar to those coming from the ground truth. In this scenario, thanks to MSE, the predicted sequences' values converge to the values of the ground truth labels.

Alternatively, we try to use the following loss functions: *Mean-Absolute-Erorr (MAE)*, *Root Mean Square Error (RMSE)* and a weighted version of the MSE in order to point out long-distance errors of the model, specially the errors that were made in the curve, the most difficult pattern to learn. The results obtained from these various metrics have been almost comparable to those obtained with MSE. This is the reason for use an human metric to observe the real effectiveness of our model.

B. Evaluation Metrics

An additional metric used in the experiment was the calculation of the Euclidean distance through the depth information. The frame compute from the Network described in Section II, is associate with its predicted trajectory and with its ground truth trajectory. Then the depth is used to turn the pixel coordinate (x_p, y_p) in the image plane into the real word point (x_r, y_r). This transformation allowed us to compute the distance in centimeters between ground truth points and predicted points. This transformation is made in the following way:

$$x_r = \frac{(x_p - cx) \cdot depth(x_p, y_p)}{fx} \tag{7}$$

$$y_r = \frac{(y_p - cy) \cdot depth(x_p, y_p)}{fy} \tag{8}$$

$$depth(x_p, y_p) = \frac{1}{9} \cdot \sum_{i=x_p-1}^{x_p+1} \sum_{j=y_p-1}^{y_p+1} depth[i, j] \tag{9}$$

where cx, cy, fx, fy are the camera parameters and $depth[i, j]$ returns the value of the image depth corresponding to the one being analyzed. To avoid also possible measurement noises is made the average, in the neighborhood of (x_p, y_p), of the value obtained by $depth[i, j]$. After this, are computed the tuple (x_r, y_r), that represent the Euclidean distance between the ground truth and the predicted point. At the end we focus our attention only in few point to understand the model's efficiency. In fact we analyzed the average error on the entire test only in the following predicted points: $[1, 5, 10, 15, 20, 25, 30]$. This technique help us to know how the model generalize on new data.

C. Human Metrics

Another important metric used to analyze the predicted trajectories was the visual metric. This type of metric is used to better understand how much the predicted trajectories differ from the ground truth. So the predicted trajectories obtained from the model are plotted in the corresponding frame with also the ground truth. With this kind of method it was easier to evaluate the goodness of the trained model and of the predicted trajectory. Thanks to this type of metric it was possible to understand that the MSE loss function, instead of other loss function cited in Section III-A, can give a better stability and continuity to the predicted trajectory specially in the prediction of curvilinear trajectories.

D. Proposed Models

In our project we have developed three different models based on the same architecture described in Section II. The main difference in these three approach is based on network's input. In the first case we have an RGB input image, in the second case we have a sequence of RGB image and in the last case we consider the RGB image and the relative Depth image. In the following part of the article we refer to these three kind of approach in the following way: *single-frame* for the first case, *multiple-frame* for the second case and *depth* for the last case.

Parameter optimization is performed with mini-batch Stochastic gradient descent (SGD) with batch size 20 and momentum 0.9. We choose and initial learning rate of $\eta_0 = 0.1$, and the learning rate is updated with decay rate $\rho = 0.1$ every

20 epochs. To reduce the effects of "gradient exploding", we use a gradient clipping of 5.0 [10]. We explored other more sophisticated optimization algorithms such as *Adam* [11] or *RMSProp* [12], but none of them meaningfully improve upon *SGD* with momentum and gradient clipping in our preliminary experiments.

Another important aspect was the initialization of the hidden and cell state in LSTM. The default approach to initializing the state of an RNN is to use a zero state [13] or with a noisy ones [14]. Instead in our case, these two type of initialization make a lot of noise in the first predicted points which carried to a bad trajectory. For this reason we have chose to initialize the LSTM state with the feature vector (*fc7 vector*) extracted by the *CNN* from the input frame. This type of initialization give more information to the LSTM and it allow better prediction in the train and test phase too. There we have a different hidden and cell state size: in the *single-input* model the hidden and cell state have size 128, in the *depth* model the states have size 256 and in the *multiple-input* model the states have size 512. We have to notice that the hidden and cell state have the same dimension of the feature vector extracted by the *CNN*. This give us the intuition of the reason of the different dimensions: the internal *LSTM* state size, like the fc7 vector, has to be directly proportional with the network input information.

IV. Experiments

A. Dataset

To conduct our experiments we used a dataset provided by the University of Florence obtained from a remote-controlled machine equipped, among other components, of an RGB camera and of a depth one. The dataset is overall composed of 16 video sequences each consisting of multiple RGBA and depth frames. Each sequence has a json file associated where are stored information for each video's frame. In particular we focus on the following fields: *future*, that is a list of coordinates indicating the trajectory of the machine from each frame until the last frame of the video; *present* that is an identical field for all dataset's frames, its value is the point from which starts the calculation of the trajectory contained in the future field.

We split the sequences to get the training set, the validation set, and the test set, consisting of 11 sequences for the training set, 3 for the validation set and 2 for the test set. The acquired frames are originally 1280x720 both in the RGBA format and depth. In order to achieve better results, we do some preprocessing stuff on these images. In particular, we crop the upper part on every image, that is the sky, and the lower one, where there is the road; then we convert every image from RGBA format to RGB and, finally, we resize the images, both RGB and depth ones, to 160x90. In order to have the same scale both for RGB and depth images, we crop some depth values, in essence every value grater than 10000 we fix to 10000, and we normalize every value in $[-1, 1]$ dividing it by the average value (127.5 for RGB images and 5000 for depth ones) and subtracting 1.

Figure 5: The model trained on 30-dim trajectory predictions (red line) generate smoother sequences of length 10 than the ones predicted from a model trained to create sequences of 10 coordinates (green line).

B. Results

Our experiments are composed by a part of hyperparameters' tuning of the architecture, and by some experimentations on the network's layers and of the input data given to the model.

Among all the hyperparameters of the model, the most important is the initial value of the learning rate of the SGD optimizer. We set that to 0.1 and the decay period every 20 epochs of 0.1. With other values, the model could not converge and the loss value remained high, so the generated trajectories were wrong. We focus also on the number of the LSTM layers, on the size of the vector corresponding to the RNN states and how these vectors were initialized. What give us a great contribution for our work was how we initialize the LSTM states. If we initialize the states with vectors of zeros or with a vector of random numbers, the model generate wrong or very noisy trajectories. This effect could be observed in the first points of the generated trajectory which were very far from the real ones, that is because the model could not learn how to behave in front of values different from those it had to learn to predict.

Each model has been trained to predict a trajectory consisting of 30 coordinates, which determinate the positions that the vehicle will assume in the next frames. Fixing the length of the trajectory at this value, we were able to have a trade-off between one trajectory that is too short or too long. In the first case, the learning of the model is complex specially in the curvilinear portions of the video since it is not the very predictable the presence of a turn. In the other, the model has to predict a trajectory with too much point in the future, that produces few errors in the last points of the trajectory and a lot of noise in the coordinates near the present point. The reason is related to our loss function, MSE, where grater numbers are more significant than the lower ones, so the errors on the first coordinates of the trajectory matter less than ones in the end of the generated trajectory. Since in a real case of use it may

Table I: Euclidean distance on test set for our methods.

Model	1 [cm]	5 [cm]	10 [cm]	15 [cm]	20 [cm]	25 [cm]	30 [cm]
SF	1.47	4.28	8.72	12.93	18.03	23.75	31.46
MF	2.65	4.22	7.91	12.69	19.18	24.21	28.94
D	2.76	4.42	9.09	12.53	18.66	22.44	29.28

be required to predict short trajectories, we have verified that the model trained to generate 30-dimensional trajectories, it is more effective at predicting shorter sequences rather than use a model trained to predict trajectories of shorter values. This can be observed in Figure 5.

We have also experimented the use of additional information deriving from the depth images. We concatenate each one depth frame with the corresponding RGB and we put in input to the network, that is the depth model. With those additional input frames the obtained results were very similar to that of the single-frame model. We can see few improvements in maintaining a line more continuous during a turn, but with the defect that the first predicted points are more far than the other model. We agreed that to use such images only create a more complex model without increasing its performances.

Further experiments focused on the importance of use a decoder composed of a recurrent neural network, so we concatenated multiple RGB frames and we put them in input to network. In particular, for each RGB frame, we link together its 4 previous frames, in such way we provide to the network a memory of the position taken in its past. In this scenario the multiple-frame model reported significant improvements over single-frame and depth model. These improvements are observable both in the curved lines of the video, where the trajectory is more similar to the ground truth one than the other models, and in the stability of the generated trajectory. If we watch a video we can observe those effects: the other models got much noisy sequences, that is suddenly, for just one or two frames, the trajectory became wrong and immediately returns true. This could be due to that memory that we provide to the network of the vehicle's positions previously to the situation in which it must predict the trajectory. In Figure 6 we can see how the multiple-frame model (the red line) follows the ground truth trajectory (the white one) more than the single-frame (green) and the depth (purple) models. In this example those models cut the curve too early, while the multiple-frame, correctly, after few frames it'll curve as the ground truth does. Regarding the Table I, reporting the Euclidean Distance for our proposed methods, we don't find the visual aspects described above. The single-frame model (SF) reports best average values than the multiple-frame (MF) for the starting point of the trajectory and for point 20 and 25, while the depth model (D) has the best values at point 15 and 25.

We also tried to study the effectiveness of the use of the Dropout regularization [15] both before fully-connected levels and within the LSTM. However we achieve the same results, both in terms of loss on the test set, both visually creating a video with the trajectories predicted by a model with the

Figure 6: Different behavior between our three models: multi-frame (red), single-frame (green), depth (purple).

Figure 7: With the use of Dropout (red line) our model generate a more stable trajectory than using no regularization strategies (green trajectory).

Dropout layer and with those generated without its use. Few improvements of using Dropout can be observed through the values of the Euclidean Distance obtained using the depth, as can be seen in the Table II. Given these considerations, we have chosen to use it since the model, through the regularization layers, it's more robust and learns more general rules than a network without a Dropout. With this choice, the generated trajectory is more stable than the trajectory created without the regularization, as shown in Figure 7. That stability is useful to generalize the road holes' noise which involves random movements of the vehicle, even if in the ground truth trajectories. We also noticed that, in optical of future usage of our application on a physical device, the model with Dropout doesn't introduce so much latency times in the prediction of the trajectory. Generally in our studies, the prediction time increases from 10ms to 12ms, that is approximately 80fps.

V. Conclusions

In this work, we presented three models based on the same architecture for doing trajectory's predictions for an autonomous vehicle: *single-frame*, *multiple-frame* and *depth*

Table II: Average values on the test set at some points of the trajectory to evaluate the dropout effect on the model.

Model	**1 [cm]**	**5 [cm]**	**10 [cm]**	**15 [cm]**	**20 [cm]**	**25 [cm]**	**30 [cm]**
no drop	1.84	4.11	8.22	12.58	19.73	25.30	31.84
dropout	2.65	4.22	7.91	12.69	19.18	24.21	28.94

model. We observe that, the *single-frame* and the *depth* models achieve not so bad results related to the Euclidean Distance, but, as we can see in videos, these models generate trajectories that are not so similar to the ground truth ones. Instead, the *multiple-frame* model show the effectiveness of using RNN decoder to predict sequence of points. Not just for smaller values of distance from the real trajectories, as shown by the Euclidean Distance, but, mostly, for the quality of the predicted trajectory, that is more smooth, more stable and correct the errors of the other models in the first points of the trajectories. These improvements are due to the added informations derived from the previous frames concatenated to each video frame.

We show also the effectiveness of using both an Euclidean Distance and a visual metric, because sometimes even if there are small changed in the evaluation metric, plotting those trajectories in a video, we can evaluate better what kind of trajectory was the best.

The code and models are shared on this GitHub repository, for some videos of our experiments follow that Dropbox link.

References

[1] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Micȟele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumaȓ, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, T̃homas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466.

[2] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Julia Kindelsberger, Li Ding, Sean Seaman, Hillary Abraham, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. MIT autonomous vehicle technology study: Large-scale deep learning based analysis of driver behavior and interaction with automation. *CoRR*, abs/1711.06976, 2017.

[3] Araz Taeihagh and Hazel Si Min Lim. Governing autonomous vehicles: emerging responses for safety, liability, privacy, cybersecurity, and industry risks. *Transport Reviews*, 39(1):103–128, Jul 2018.

[4] *Pytorch*. Pytorch. https://pytorch.org.

[5] *PyTorch*. wikipedia, 2019. https://en.wikipedia.org/wiki/PyTorch.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Oinkina and Hakyll. *Understanding LSTM Networks*. colah's blog, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[9] Subhashini Venugopalan, Huijuan Xu, Jeff Donahu, Marcus Rohrbach, Raymond Mooney, and Kate Saenko. *Translating Videos to Natural Language Using Deep Recurrent Neural Networks*. 2015.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[12] Yann N. Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization, 2015.

[13] Danijar Hafner. Tips for training recurrent neural networks. Blog post, 2017.

[14] Hans-Georg Zimmermann, Christoph Tietz, and Ralph Grothmann. *Forecasting with Recurrent Neural Networks: 12 Tricks*, pages 687–707. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.