Pedestrian Motion Classification for Autonomous Vehicles (6.867 Final Project)

Michael Everett & Björn Lütjens

***Abstract*— In this project, we applied techniques from the course to a dataset of pedestrian trajectories recorded on a golf cart around MIT's campus. Our objective was to classify whether a pedestrian's trajectory would enter a 4x10m rectangle in front of the vehicle, which could be used as an active safety feature on various types of vehicles that interact with pedestrians. We trained a SVM with Gaussian RBF Kernel and optimized hyperparameters to achieve 74% test accuracy. We then trained an RNN for the same task and achieved M. E. RNN accuracy? accuracy. Finally, we experimented with pedestrian motion prediction, to predict the future pedestrian trajectory.**

(a) Ground robot in Stata (b) Golf carts (MIT MoD fleet)

Fig. 1: Many types of autonomous vehicles operate among pedestrians and must account for pedestrian motion in the vehicle motion planning algorithms.

I. Introduction

Autonomous vehicles use onboard sensors to perceive their surroundings, and use the data to make decisions about where it is safe to drive. Today's research vehicles typically rely on Lidar and cameras as the main sensors, as these provide information about where/what obstacles are, and can help the vehicle maintain safe position in its lane and with respect to obstacles. In addition to static obstacles, vehicles interact with pedestrians in a variety of environments: people in crosswalks, jaywalkers, kids running across neighborhood streets. A campus shuttle could even drive on sidewalks among pedestrians, so the vehicle would be interacting with pedestrians almost all the time. These regular interactions mean the vehicle must be able to predict pedestrians' next moves in order to maintain safety.

Pedestrian motion prediction is difficult because people often behave very unexpectedly. Humans use many strategies to predict pedestrian intent while driving, such as body language, eye contact, and other non-verbal cues between human driver and human pedestrian, but autonomous vehicles can't easily communicate or read these signals. The information from sensors is typically fused: Lidar is used to measure the pedestrian position, and the camera is used to label the particular obstacle as a pedestrian. Therefore the only measurements collected are position over time (from which velocity can also be computed).

This project uses a dataset collected on MIT's campus over several months by three golf cart shuttles providing Mobility on Demand service to students, while simultaneously collecting pedestrian trajectory data to optimize vehicle routing strategy. The dataset contains about 65,000 pedestrian trajectories as well as the vehicles' trajectories, all in a global frame across the MIT campus.

M. E. **related works**

The objective of this project is to develop a classifier that can determine whether a person will step in front of a vehicle, based on a few seconds of their trajectory. We use a portion of our data set to train different classifiers, optimize hyperparameters with a separate portion, and evaluate performance with yet another portion. This classifier could be a useful component of an autonomous vehicle, or part of an active safety feature on a human-driven vehicle that could take over in case the driver does not see a pedestrian in time.

The main contributions of this work are (i) a pedestrian trajectory dataset with 65,000 trajectories over three months, (ii) an SVM classifier for predicting when pedestrians will step in front of a vehicle, (iii) a classifier using Deep RNNs for that same objective, and (iv) a pedestrian motion predictor using Deep RNNs.

II. Dataset

An important realization we made during this project is the challenge of working with a real dataset. Understanding the raw data became a significant portion of the project, so this chapter provides a thorough explanation of our findings and methods to process the data.

A. Raw Data

Our data comes from golf carts equipped with Lidar and cameras [?], [?]. Fortunately, it is relatively straightforward to visualize trajectory data, as opposed to some high-dimensional datasets that exist for other applications.

The raw dataset's fields are shown in Table I, where (easting,northing) are the latitude/longitude global coordinates, and (x,y) are the coordinates in our global campus map. Veh id indicates which of the three vehicles corresponds to that data point, or in the pedestrian case, which vehicle sensed that pedestrian. Ped id is a unique id given to each pedestrian seen.

There are some noise-related issues with the raw data, as it was collected on a research vehicle under development. One issue is that the vehicle's (x,y) position sometimes jumps, because the vehicle's localization system does not use

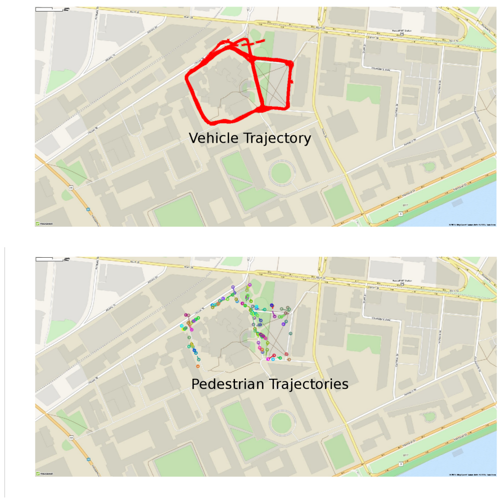


Fig. 2: The raw trajectories from one day of data collection visualized on a campus map. All trajectories are recorded in the global map coordinate frame.

Type	Fields						
Vechicle	time	easting	northing	x	y	veh id	
Pedestrian	time	easting	northing	x	y	veh id	ped id

TABLE I: Raw data fields

GPS and is imperfect. Other minor issues include pedestrian trajectories that incorrectly split/merge or are too short to be useful for this project.

In addition to addressing noise, we also pre-process the data by converting pedestrian trajectories into the vehicle's local frame. Our objective is to classify whether pedestrians cross in front of the vehicle, so the pedestrian trajectories must be converted into the vehicle's local frame. That is, our dataset is initially unlabeled, and we must generate the ground truth label that we wish to learn to classify later.

It might be possible to somehow feed both the vehicle trajectory and pedestrian trajectory into a classifier, and have it learn the transformation, but it seemed much more straightforward for us to rotate the data ahead of time. Using global coordinates could potentially allow the classifier to learn a global understanding of the map (e.g. where sidewalks/intersections are), but we chose to focus on a more structured problem for this project. Since global coordinates should have an impact on pedestrian motion, we could in the future try a hybrid approach where we feed local coordinates along with some context features (e.g. distance to curb, traffic light state) as in [**?**].

B. Global-to-Local Transformation

The global-to-local transformation relies on knowledge of vehicle orientation (heading angle) and smooth vehicle trajectories, neither of which we have by default. Line 1 describes the procedure for filtering, transforming, and labeling the raw dataset. M. E.**explain fig 3** M. E.**explain how to do global-to-local transform**

Algorithm 1: Algorithm for extracting local trajectories

```
Input: $V_g$, $P_g$: global vehicle, pedestrain trajectories (Table I)
Output: $P_l$: pedestrian trajectories in local vehicle frame
1: for each vehicle do
2:     $I_{posjump} = \{i \in [1, len(V_g) - 1] \mid euclid\_dist(V_g(i) - V_g(i+1)) > 1.0\}$
3:     $I_{timejump} = \{i \in [1, len(V_g) - 1] \mid time(V_g(i) - V_g(i+1)) > 0.5\}$
4:     J = $I_{posjump} \cup I_{timejump}$
5:     $T_{valid} = \{[J(i), J(i+1)] \mid time(J(i+1) - J(i)) > 5.0\}$
       M. E.finish
6: return pedestrian trajectories
```

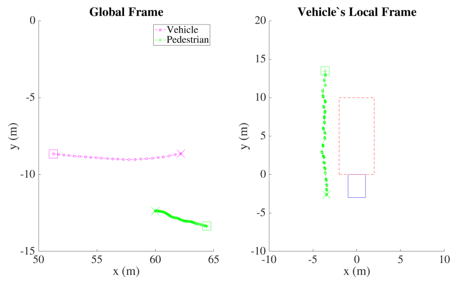


Fig. 3: On the left, the vehicle trajectory (magenta) and pedestrian trajectory (green) in the global frame. Both start from the square and move to the X. On the right is the local vehicle frame: the vehicle is the blue rectangle, and the red dashed rectangle is the "cross" zone. The green pedestrian trajectory doesn't cross into the red rectangle, so this trajectory is given binary label 0.

C. Processed Dataset

The processed dataset is visualized in Fig. 4. The vehicle (yellow taxi) has a blue rectangle representing the "cross" zone (similar to Fig. 3). Green trajectories are local pedestrian trajectories that do not cross into the blue zone, and red trajectories are ones that do enter the blue zone.

A few important observations about the processed dataset are the jaggedness and locations of trajectories. The jaggedness is due to the transformation and sensor rates: the Lidar provides pedestrian trajectory at high rate (50 Hz), but the vehicle position is updated at a slower rate (10 Hz), so the pedestrian trajectory jumps a bit each time vehicle time step. Since the rotation is dependent on vehicle heading angle, and vehicle heading angle is computed from consecutive vehicle positions, this process is subject to some noise. It is also somewhat difficult to imagine what local trajectories *should* look like, because these trajectories are showing how the vehicle and pedestrian move relative to one another, and an observer doesn't know what the vehicle's velocity was for any particular trajectory. For example, a trajectory that arcs along the front of the vehicle could be the vehicle turning as a pedestrian walks straight, or a pedestrian walking in a curve while the vehicle is parked. All of this could have been

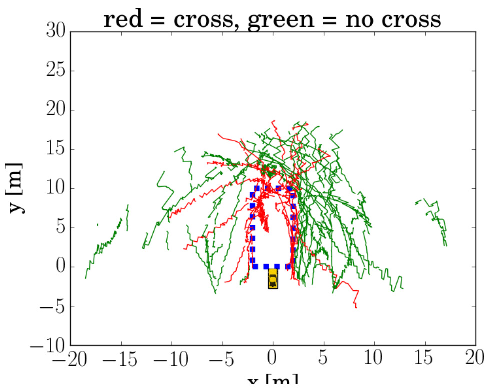


Fig. 4: A small portion of the training set for the binary classifier. The vehicle (yellow taxi) has a blue rectangle representing the "cross" zone. Green trajectories are local pedestrian trajectories that do not cross into the blue zone, and red ones do.

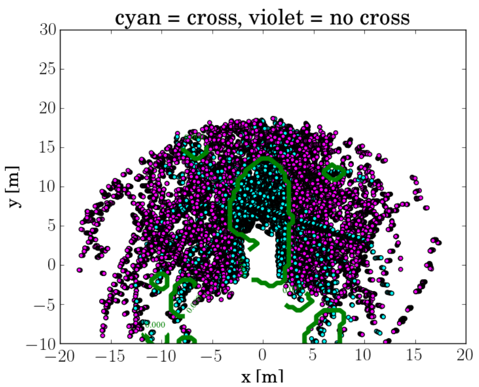


Fig. 5: SVM trained on snippets with $l = 1$ allow visualization of decision boundary. The important part of this plot is the green decision boundary in the center that resembles our arbitrary "cross zone". The SVM has roughly learned the location/shape of zone for simple input data. There is some overfitting evident by other green regions, but this is likely more of an artifact of $l = 1$ hyperparameter.

avoided if we had access to raw sensor measurements, as these provide local coordinates by default.

The second observation is that the trajectories are in front/along the sides of the vehicle, and have a limited range of about 20m. This makes sense given our sensor: it is mounted on the vehicle's front and sees a 270° cone. Its maximum range is more than 20m, but tracking pedestrians reliably is difficult beyond this distance.

III. SVM Binary Classifier

We trained a Support Vector Machine (SVM) to do binary classification on local pedestrian trajectories, to predict whether the trajectory will/will not enter a 4x10m rectangle in front of the vehicle. A portion of the pedestrian trajectory is fed into the classifier, and it outputs a binary label. This could be a useful safety feature on a car to identify which pedestrians might cross in front of the vehicle.

The trajectories are all of different length, so we split them up into equal "snippets" of length l. Each trajectory has a single 0/1 label, and this same label is applied to each snippet. This allows us to learn some notion of direction from (x,y) sequences, because a snippet that never crosses in front of the vehicle can still be labeled as such, if a later portion of its trajectory eventually does cross in front. These l-long lists of (x,y) points are fed into the classifier as the feature vector of length $2l$, as: $(x_i = [x_1, y_1, x_2, y_2, ..., x_l, y_l]; y_i = 0/1)$.

We used the SKLearn implementation of SVM [**?**] with Gaussian RBF Kernel.

A. *Simple Example*

We first tried training with $l = 1$, so that we could easily visualize the decision boundary in Fig. 5. The data (magenta/cyan) aren't particularly meaningful, because it's hard to learn anything from a single (x,y) position of a pedestrian ($l = 1$). But, we can observe that the green decision boundary roughly resembles our arbitrarily-created rectangular "cross zone". This figure indicates the SVM is correctly learning that there is a region in space that causes trajectories to be labeled a certain way. After this quick sanity check, we can proceed to larger values of l, but lose the ability to easily visualize the results.

B. *Hyperparameters*

There are several hyperparameters that affect the behavior of the classifier. The main ones that we experimented with were l (snippet length) and C (SVM regularizer). A grid search for optimal parameter values is shown in Table II (optimal w.r.t. high validation accuracy). Beacuse of the time required to train so many combinations (especially affected by Gaussian RBF Kernel), we trained on a smaller subset of the training set during hyperparameter optimization.

As C increases, training accuracy increases for all scenarios, which is expected because this corresponds to more penalty on mis-classified points. We can tell when C is too high, because training accuracy is much higher than validation accuracy, a symptom of overfitting.

As l increases, more time steps of pedestrian positions are used in the feature vector, so there is more information on which to classify. Intuitively, large l would seem preferable than small, but if l is too large, the noise in the data may outweigh the value of the information. To be more concrete, it seems unlikely that one would need 50 points worth of trajectory data to do the binary classification (especially given the noise seen in Fig. 4).

This intuition aligns with our observed validation accuracies. The highest validation accuracy of 79.8% occurs with $l = 25$ and $C = 10$, so we choose those as the optimal hyperparameters.

The test accuracy with these values was 74.4%.

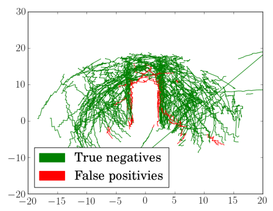
C. *Results*

With these optimized hyperparameters, we then trained on 1 week of data (9700 trajectory snippets). The test accuracy was 84.8%. This is higher than the accuracy from hyperparameter optimization because there was more training data.

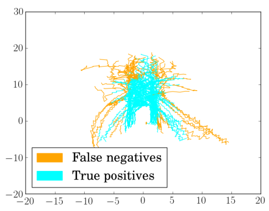
We show the various combinations of true/predicted labels in Fig. 6, with numerical listings in Table III. On the left

Dataset	$l = 2$							$l = 10$						
	$C = 10^{-3}$	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}
Train	70.8	83.2	84.8	85.9	87.1	89.1	91.9	63.8	76.1	84.6	86.8	92.6	97.2	99.6
Val	63.1	77.4	79.1	78.9	78.0	77.8	77.6	60.3	67.9	78.4	79.4	79.3	77.7	76.1
Dataset	$l = 25$							$l = 50$						
	$C = 10^{-3}$	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}
Train	62.9	64.9	82.4	88.7	96.8	99.9	100	60.4	60.4	77.6	92.2	99.3	100	100
Val	59.2	59.3	75.0	79.7	79.8	78.8	78.8	52.3	52.3	64.8	75.0	74.8	74.5	74.5

TABLE II: Accuracy of SVM for various values of hyperparameters C, l.



(a) Trajs. labeled "No Cross"



(b) Trajs. labeled "Cross"

Fig. 6: Predicted labels on test set with SVM classifier. On the left, trajectory snippets that were predicted label 0 (green is correct, red is incorrect). Notice the large rectangular gap in the center, corresponding to learned "cross zone". On the right, trajectory snippets that were predicted label 1 (cyan correct, orange incorrect). Across the whole test set, 84% accuracy was achieved.

(Fig. 6a), we show the trajectory snippets for which our classifier predicted a label of 0. Our accuracy on trajectories that don't cross in front of the vehicle are pretty high. It is interesting to note the rectangular gap in the center; this demonstrates that our classifier learned that trajectories that enter that region should never be classified as 1. There are some red trajectories around the edge of the rectangle which are particularly challenging.

On the right (Fig. 6b), the trajectories our classifier labeled as 1 are relatively somewhat less accurate. Recall that the trajectories are split into snippets, so for long trajectories that are split into many snippets, this classifier seems to perform much worse on the snippets that are far away from the vehicle. It is expected that the performance would be worse further away from the vehicle. Some of the long trajectories are still classified correctly, indicating the classifier is learning some notion of direction, since we are only feeding in pedestrian positions over time.

Classification	Number	Percent
False Positives	101	4.6
False Negatives	234	10.6
True Positives	787	35.8
True Negatives	1076	49.0
Total	2198	100.0

TABLE III: Classification matrix

D. *Kernel Choice*

According to [?], time series data can be classified with a SVM with many different types of kernels, but the most broadly applicable one is Gaussian RBF. Therefore, we used a Gaussian RBF kernel with γ set automatically by SKLearn's implementation. This kernel allows non-linear combinations of feature elements, which could be useful for this task. Ideally, we would use some kernel that was designed specifically for our time-series trajectory data, but we didn't focus much on this aspect of the problem. Instead, we considered a different learning method (RNN) to try to better capture the time dependence of our data.

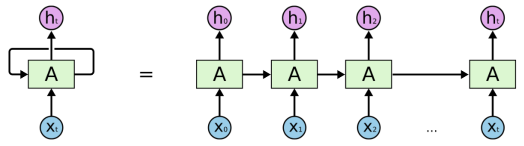


Fig. 7: Illustration of a RNN in it's enrolled (left) and unrolled (right) form. Each sequential input x_t is fed in at timestep t, transformed by the activation cell A and output and fed into the next cell as hidden state h_t. (colah)

IV. Recurrent Neural Network: Binary Classification

A. *Introduction*

Recurrent Neural Networks (RNNs) have been shown to extract sequential information and generate highly complex sequences as documents, translations, music and more. In comparison to n-gram models, which have been used in language model for speech recognition, a RNN makes a prediction by a high-dimensional interpolation between training samples. N-gram models predict by creating a distribution, based on the number of exact matches between the recent history and the training set (graves).

In theory RNNs can be trained by backpropagating the error along the chain of cells. However, every step involves a multiplication with the weight matrix W, which, in many cases, results in either an exploding or vanishing gradient (see class exercise). The exploding gradient problem is, in practice, leveraged by gradient clipping (cs232 lecture). The *vanishing gradient problem* is leveraged by a specific architecture of RNNs, called *Long Short-term Memory* (LSTM) networks. They are splitting up the RNNs hidden state into a hidden and a cell state, whereas the cell state, like ResNets, is updated by a sum-operation, rather than a whole transformation (SEE FIGURE ... for the comparison of RNN to LSTM cell). The gradient along a sum-operation splits up equally and thereby allows to propagate until the beginning of the sequence without vanishing.

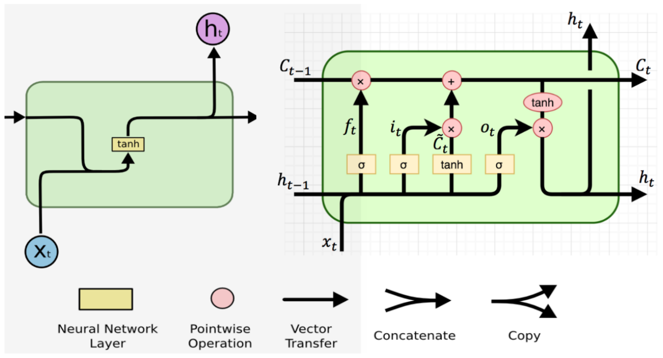


Fig. 8: Activation cell of a traditional RNN (top) and a LSTM (bottom). LSTM propagates the cell c_t and hidden h_t state. With ignoring the forget gate f_t, c_t is only updated by sum operations. The gradient along a sum-operation splits up equally and thereby allows to propagate until the beginning of the sequence without vanishing. ResNets have introduced a similar change in the CNN architecture (resnet paper). (changhau)

The LSTM is computed according to the following composite function (graves):

$$\begin{aligned} i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\ c_t &= f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \\ h_t &= o_t tanh(c_t) \end{aligned} \tag{1}$$

where σ is the sigmoid function, i_t, f_t, o_t and c_t are the input gate, forget gate, output gate and cell activation vectors. All weight matrices have the same meaning, for example W_{hf} is the hidden-forget weight matrix. For clarity, bias terms are omitted. Figuratively speaking, the input gate regulates how much information is taken from the new input x_t. The forget gate can erase the cell memory and thereby shorten or lengthen the time dependency of a sequence prediction. The cell state is the propagated memory. The output gate regulates how much information from the hidden cell state is propagated into the output hidden state.

B. *LSTMs for binary classification*

We have used LSTM networks for the same binary classification task, as for SVMs. LSTMs are able to extract the sequential information and should thereby provide better classification accuracy in the train and test dataset. All calculations have been done in python and the network has been set up with the *tensorflow* library.

The full trajectories have been precompiled into sniplets of static sequence length T. At each time step, the input vector is the relative pedestrian position at time t: $x_t \in \Re^{1x2}$. For binary classification, we use a many-to-one architecture. The output is calculated at the output layer by using a logistic sigmoid for binary classification:

$$y_t = \sigma(W_{hout}h_t + b_out) \tag{2}$$

where $y_t \in [0,1]$ and $W_{hout}\Re^(n_{hidden}x1)$, where n_{hidden} is the number of hidden units. The weight matrices and biases are initialized with random noise $N(0,1)$. The loss is computed with least mean squared error (LMSE). The system is optimized by a stochastic gradient descent (SGD) method with the goal to minimize the loss over the weight matrices and their biases.

1) Hyperparams in bin class: - Optimize number of hidden units - batchsize - learning rate - maximum epochs

- challenge: predict real-valued sequence - low dimensionality and ease of visualization - no sophisticated preprocessing or feature-extraction techniques (graves p.18) - reduce variation in data (normalize character size, slant, skew,) - Compare our dataset to handwritten dataset size (graves p.18) - handwriting has 25 timesteps per character and 700 timesteps per line - 5000 training, two val of 1500 lines, test 4000 lines (each line 700 tsteps)

V. Recurrent Neural Network: Prediction

A. *Introduction*

Previous work has been conducted in predicting trajectories in a 2D space. The most relevant work has used LSTMs to generate human handwriting from the IAM online handwriting database (graves).

LSTMs have the possibility to predict sequences, by feeding themselves their output as an input. To update from binary classification to trajectory predicting LSTM, we change the network architecture from many-to-one to many-to-many. Our output $y_t\Re^{Tx2}$ is computed from every time step as given in **??**.

Many work is found on predicting text (hinton,2011). However, our application domain is real-valued in comparison to the discrete valued domain of predicted words.

The literature proposes several transformations, which could be applied instead to the output y_t. For example, (Graves) uses mixture density networks (MDNs) and calculates x_{t+1} by the conditional probability $P(x_{t+1}|y_t,)$ and using a Gaussian mixture prior on the probability distribution. The loss is the negative-log likelihood of the conditional probability and SGD is applied to update the weights along with the parameters of the distribution: mean μ_m, standard deviation σ_m, correlation ρ_m, mixture weights π_m for M mixture components. However, the complexity of the proposed transition function is obvious and results in longer training time. Due to our limited CPU capacity and corresponding time-consumption of hyperparameter tuning, this approach does not seem promising.

Additionally, (graves) has used the relative position $\delta x = x_{t+1} - x_t$ as input vector. In our case, we cannot use the relative position, because we would lose the information of proximity to the car. In other words, we would ignore, that a pedestrian might avoid a car, when it approaches him. For the MDN approach our means μ_m would be further spread among the full absolute position space, which would presumably consume even longer training time with graves approach. Therefore, we have decided to linearly compute the output, as stated in (2).

As we cannot use the loss function proposed by (graves), we have used our loss function, which is the mean squared

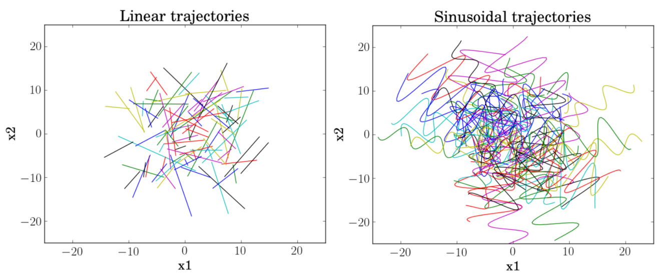


Fig. 9: Randomly generated trajectory data in linear (left) shape and sinusodial shape (right). The trajectories have been generated by sampling the start point, range, scale, translation and rotation at random.

error over the full output vector and the next ground truth input vector. We have used the L_2 norm instead of the L_1 norm to be more robust against outliers.

$$Loss_{pred}(x_t) = \frac{1}{T}\sum_{t=0}^{T}(y_t - x_{t+1})^2 \tag{3}$$

The loss function is computed for every batch and updates the parameters in the activation cells via SGD. We randomly draw batches from our training dataset for each SGD parameter update. Thereby we brake apart the dependency of previously concatenated trajectories, that we have split apart into snippets. This is necessary, as a test trajectory will be independent of the training trajectories.

For prediction, the parameter-sharing property of RNNs and LSTMs is essential. The activation units' parameters have been trained to predict the next state based on the current state. Therefore, for prediction, where the ground truth input vector is unkown, the LSTM can use the output y_{t-1} as input for x_t. This gives us the predicted pedestrian position y_t.

B. Generated Data

As seen in the previous sections, the given pedestrian trajectories are highly complex nonlinear functions. To validate our LSTM prediction model, we have trained on the prediction of simpler functions. Therefore, we have chosen to generate linear and sinusodial shaped trajectories, displayed in Fig. 9.

For our hyperparameter tuning, we have considered the batch size b, number of training samples N, snippet length T, learning rate lr, maximum epochs e_{max} and number of hidden units n_h. As the possible combinations rise exponentially with the number of parameters and our computational resources are very limited, we have evaluated most of the parameters, while holding the other parameters constant. Optimal tuning would run a search over all combinations, which would result in better performance.

Each model is evaluated by the training and validation loss, whereas both datasets are generated independently.

In the search over the number of hidden units we have made the following insights. As seen in Fig. 10, the number of hidden units $n_h = 1$ is obviously not sufficient to represent the complexity of sinusodial curves. We can also see, that

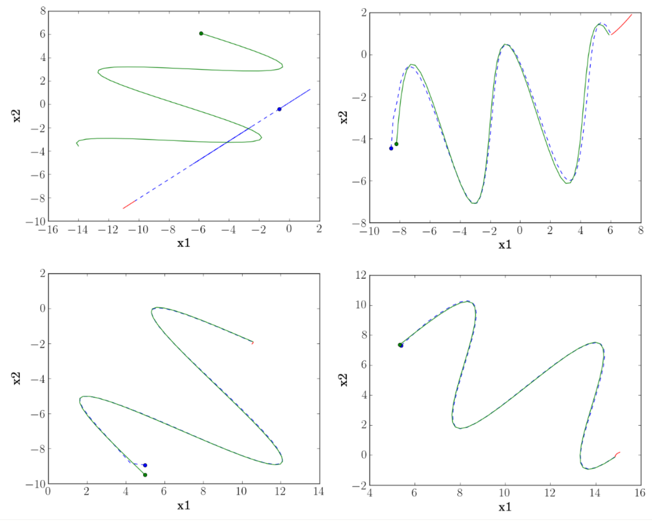


Fig. 10: Search over the number of hidden units: $n_h = 1$(top-left), $n_h = 4$(top-right), $n_h = 32$(bottom-left), $n_h = 128$(bottom-right). The plot displays the input trajectory (green) x_t with total trajectory snippet length $T = 50$ an its starting point (green-dot). The LSTM predicts the next pedestrian position y_t (blue-dotted) with its starting point (blue-dot), based on the past sequence $x_{0:t}$ and the learned parameters. After the input trajectory (green) is finished, the LSTM tries to predict the future trajectory (red). All trajectories are test trajectories and have not been included in the training set.

	$n_h = 1$	$n_h = 4$	$n_h = 32$	$n_h = 128$
$L_e = 0$	58.5	49.7	**6.22**	2.62
$L_e = 1$	54.2	36.0	**0.50**	0.12
$L_e = 2$	51.1	24.4	**0.23**	0.05
$L_e = 3$	49.2	17.8	**0.14**	0.03
e_c	25	85	35	10
L_{e_c}	0.30	0.03	0.0058	0.004
L_{test,e_c}	0.34	0.04	0.003	0.003

TABLE IV: Search over the number of hidden units n_h. The average batch loss ($b = 100$) over epochs e and test loss is displayed. Number of training samples $N = 5000$ with snippet length $T = 50$ and learning rate $lr = 0.01$.

the complexity of $n_h = 4$ is sufficient to fit the sinusodial shape approximately and $n_h = 32$ is sufficient to visually exactly fit the data.

In Table IV we can see, that increasing the number of hidden nodes n_h shrinks the loss. The training time was exponently increasing with the number of hidden units, which makes us prefer a lower number of hidden units. A high number of hidden units suggests an overfitting on the training data. However, we can see that our independently generated test data shows similar error as the training set. The complexer the model, the less epochs it took to reach acceptable loss.

From Table IV, we have seen how many epochs the model needs to converge. Every epoch repeats training on the same values and thereby increases the probability of overfitting. Therefore, we conclude that we need to train on more data ($N > 5000$), such that the model converges in a low number of epochs and does not overfit.

Include oscillating plot for convergence property. Fig. 10 also illustrates how LSTM starts by guessing y_0 purely based on the input x_0.

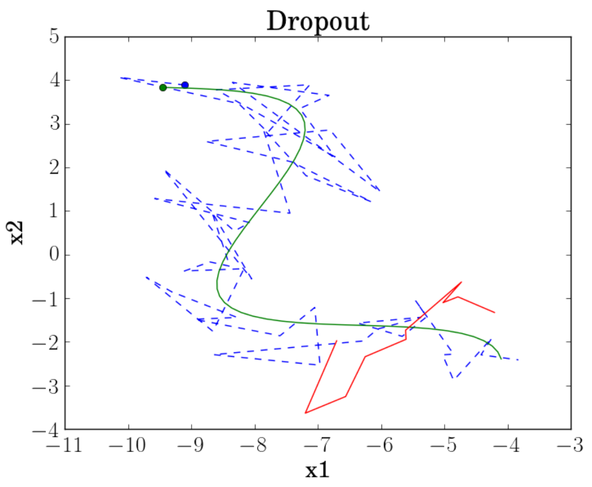


Fig. 11: Dropout for LSTMs with dropout rate $d_r = 0.5$. The predicted paths approach random guessing with $mean = x_{t-1}$, $Loss_b = 1.53$ with $b = 100$.

The use of dropout is generally not recommended for LSTMs, because it essentially erases parts of the memory. Our tests have shown similar results, where dropout caused results similar to random guessing Fig. 11

1) Linear data:

C. Results

1) Hyperparams in bin class: - Optimize number of hidden units - batchsize - learning rate - maximum epochs

- challenge: predict real-valued sequence - low dimensionality and ease of visualization - no sophisticated preprocessing or feature-extraction techniques (graves p.18) - reduce variation in data (normalize character size, slant, skew,) - Compare our dataset to handwritten dataset size (graves p.18) - handwriting has 25 timesteps per character and 700 timesteps per line - 5000 training, two val of 1500 lines, test 4000 lines (each line 700 tsteps)

D. Network architecture

- explain x_t - feeding relative x and y would not respect car position and its influence on pedestrian behavior - explain y_t

E. GRUs

From: $http : //www.jackdermody.net/brightwire/article$ $Sequence_tosequence_with_LSTM$

Long Short Term Memory (LSTM) networks are a recurrent neural network that can be used with STS neural networks. They are similar to Gated Recurrent Units (GRU) but have an extra memory state buffer and an extra gate which gives them more parameters and hence a longer training time. While performance with GRU is usually comparable, there are some tasks that each architecture outperforms the other on, so comparing each for a given learning task is usually a good idea.

F. Training

- Activation unit on output layer - Optimizer - Gradient Descent - use momentum?

1) Hyperparameters: - Dropout?! - $cell = tf.contrib.rnn.DropoutWrapper(cell, output_keep_prob = args.keep_prob)$

2) Regularization: - Random weight initialization - See graves (p.7), weight noise, adaptive weight noise - use validation set for early stopping - Gradient clipping? - could prove vital for numerical stability

G. Results

- print total loss - print avg LSE per datapoint

H. Resources

colah $http : //colah.github.io/posts/2015 - 08 - Understanding - LSTMs/$ changhau $https : //isaacchanghau.github.io/2017/07/22/LSTM - and - GRU - Formula - Summary/$ hinton 2011 generating text with recurrent neural networks icml 2011

Division of Labor & Link to code

Michael mostly worked on the dataset processing and SVM optimization, and Björn did the RNN work and the rest of the SVM work. We have been a 2-person team since Milestone 2. This project is not used for any other classes. Our code can be found at: `https://github.com/mfe7/6.867`.