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 main 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 an SVM with Gaussian RBF Kernel and optimized hyperparameters to achieve 86.3% test accuracy. We then trained an LSTM for the same task and achieved 80.4% test accuracy. Finally, we experimented with pedestrian motion prediction, to predict the future pedestrian trajectory, which is more complicated than binary classification, and achieved $\sqrt{0.11}$m deviation when predicting the pedestrian's next position.**

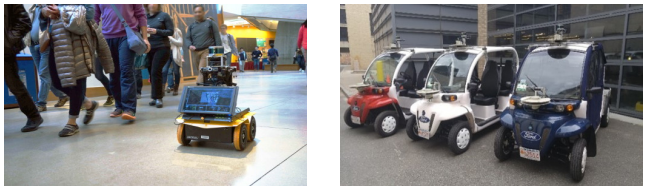
I. Introduction

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.

We omit a thorough literature review section for the sake of brevity – relevant works that are specifically used in the project are cited throughout.



(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.

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) an SVM classifier for predicting when pedestrians will step in front of a vehicle, (ii) a classifier using LSTMs for that same objective, and (iii) initial results for pedestrian motion prediction using LSTMs.

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 an explanation of our findings and methods to process the data.

A. Raw Data

Our data comes from golf carts equipped with Lidar and cameras [1], [2]. 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.

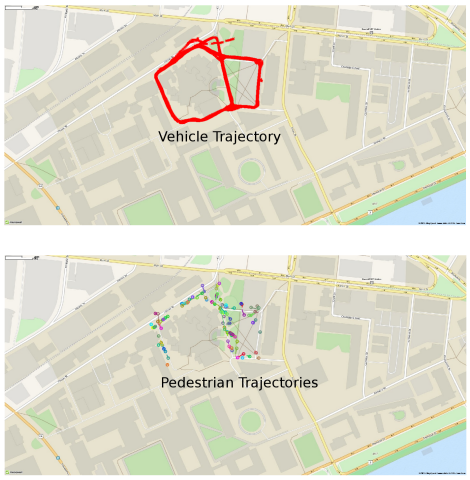


Fig. 2: The raw trajectories from one day of data collection visualized on a campus map. All trajectories are recorded in the global map coordinate frame.

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

One issue is that the vehicle's (x,y) position sometimes jumps, because the vehicle's localization system does not use 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 global vehicle trajectory and global pedestrian trajectory into a classifier, but it seemed much more straightforward for us to rotate the data ahead of time rather than hoping the algorithm would learn how the two data streams interact. Further, the global coordinates span 100s of meters, so we constrain the size of the input space by transforming into a local frame.

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 [3].

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 exist in the raw dataset. The global-to-local transformation projects the vehicle-to-pedestrian relative position vector onto the coordinate frame that points straight out of the vehicle's front and left side. We omit a detailed description for the procedure for filtering, transforming, and labeling the raw dataset, since the focus of the project is the learning algorithms.

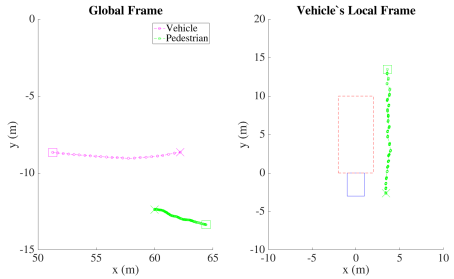


Fig. 3: On the left, the vehicle trajectory (magenta) and pedestrian trajectory (green) in the global frame. Both start from the square and move to the X. On the right is the local vehicle frame: the vehicle is the blue rectangle, and the red dashed rectangle is the "cross" zone. The green pedestrian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

The transformation is visualized in Fig. 3. On the left, the global frame shows the vehicle (magenta) moving from left to right, and the pedestrian (green) moving from bottom right toward the left. Therefore, the pedestrian is moving along the vehicle's right side. This is exactly what we see in the vehicle's local frame on the right plot. The blue rectangle represents the vehicle, and the dashed red rectangle is the "cross zone". This pedestrian does not enter the "cross zone", so it is given 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 show 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 avoided if we had access to raw sensor measurements, as these provide local coordinates by default.

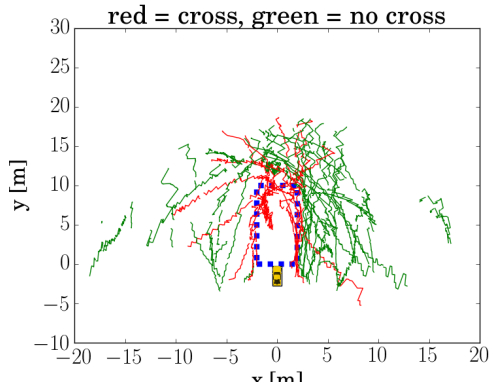


Fig. 4: A small portion of the training set for the binary classifier. The vehicle (yellow taxi) has a blue rectangle representing the "cross" zone. Green trajectories are local pedestrian trajectories that do not cross into the blue zone, and red ones do.

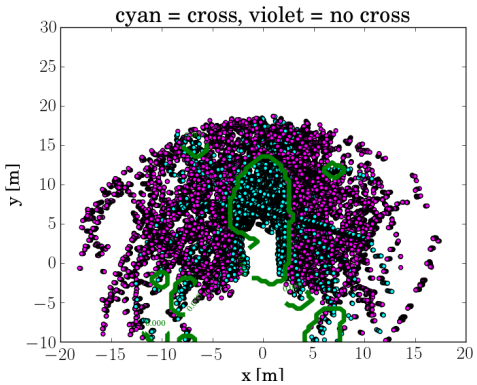


Fig. 5: SVM trained on snippets with $T = 1$ allow visualization of decision boundary. The important part of this plot is the green decision boundary in the center that resembles our arbitrary "cross zone". The SVM has roughly learned the location/shape of zone for simple input data. There is some overfitting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

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° region. 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 T. 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 T-long lists of (x,y) points are fed into the classifier as the feature vector of length $2T$, as: $(x_i = [x_1, y_1, x_2, y_2, ..., x_T, y_T]; y_i = 0/1)$.

We used the SKLearn implementation of SVM [4] with Gaussian RBF Kernel.

A. Simple Example

We first tried training with $T = 1$, 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 ($T = 1$). 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 T, but lose the ability to easily visualize the decision boundary.

B. Hyperparameters

There are several hyperparameters that affect the behavior of the classifier. The main ones that we experimented with were T (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 T increases, more time steps of pedestrian positions are used in the feature vector, so there is more information on which to classify. Intuitively, large T would seem preferable than small, but if T 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 $T = 25$ and $C = 10$, so we choose those as the optimal hyperparameters.

The test accuracy with these values was 74.4%.

C. Training Set Size

Another parameter that affects classifier accuracy is size of training set. We show in Fig. 6 that increasing the training set size increases the test accuracy. The curve levels off around 20,000 trajectory snippets, which corresponds to about 2 weeks of data collection.

Dataset	$T=2$							$T=10$						
	$C=10^{-3}$	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	$T=25$							$T=50$						
	$C=10^{-3}$	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, T.

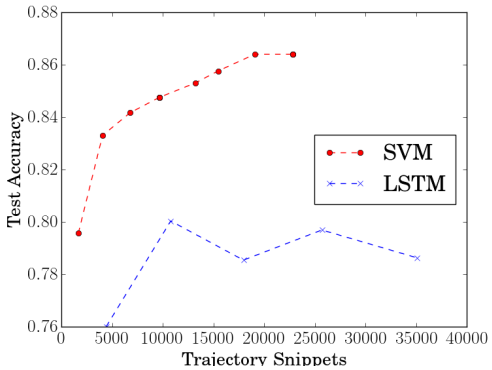
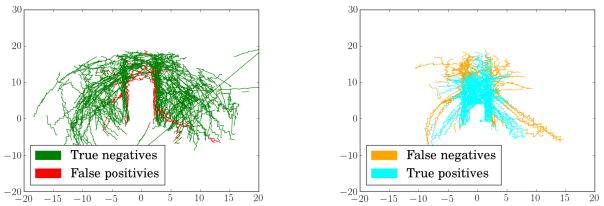


Fig. 6: Effect of training set size on SVM and LSTM binary classifiers. As expected, test accuracy increases with more training data, but levels off after about 20k trajectory snippets for SVM, and 10k for LSTM.

Classification	SVM		LSTM	
	Number	Percent	Number	Percent
False Positives	100	4.5	349	6.0
False Negatives	199	9.1	791	13.5
True Positives	822	37.4	1840	31.6
True Negatives	1077	49.0	2850	48.9
Total	2198	100.0	5830	100.0

TABLE III: Classification matrix



(a) Trajs. labeled "No Cross" (b) Trajs. labeled "Cross"

Fig. 7: 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.

D. Results

With these optimized hyperparameters and training set size, we then trained on 22,481 trajectory snippets. The test accuracy was 86.3%. Note that 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. 7, with numerical listings in Table III. On the left (Fig. 7a), 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. 7b), 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.

E. Kernel Choice

According to [5], 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.

IV. Recurrent Neural Network: Binary Classification

We first briefly describe RNNs and LSTMs, as these were not covered in homework assignments. Then, we analyze the performance of an LSTM for the same binary classification task as before, and compare with the SVM implementation.

A. Brief Overview of RNNs

Recurrent Neural Networks (RNNs) have been shown to extract sequential information and generate highly complex

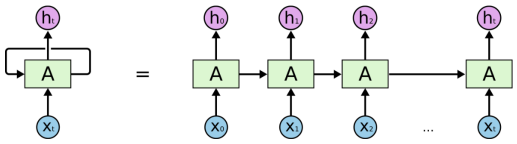


Fig. 8: Illustration of a RNN in it's enrolled (left) and unrolled (right) form. Each sequential input x_t is fed in at timestep t, transformed by the activation cell A and output and fed into the next cell as hidden state h_t. From [6]

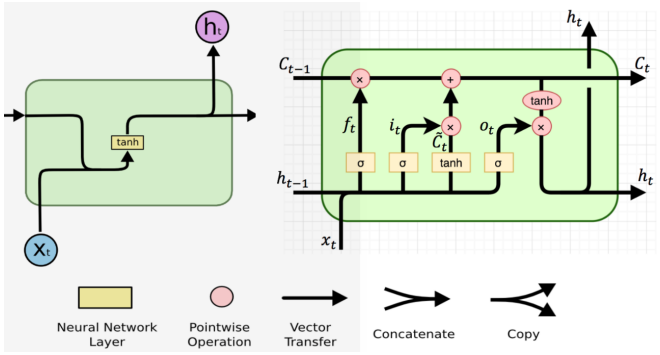


Fig. 9: Activation cell of a traditional RNN (top) and a LSTM (bottom) [6]. LSTM propagates the cell c_t and hidden h_t state. With ignoring the forget gate f_t, c_t is only updated by sum operations. The gradient along a sum-operation splits up equally and thereby allows to propagate until the beginning of the sequence without vanishing. ResNets have introduced a similar change in the CNN architecture (resnet paper).

sequences as documents, translations, music and more.

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/vanishing gradient problem is often addressed by a specific architecture of RNNs, called *Long Short-term Memory* (LSTM) networks. LSTMs split up the RNN's hidden state into a hidden and a cell state. The cell state, similar to in ResNets, is updated by a sum-operation, rather than a whole transformation as depicted in Fig. 9. The gradient along a sum-operation of LSTMs does not vanish in the same way as it does during the multiplication of RNNs.

The LSTM is computed according to the following composite function [7]:

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

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. The weight matrices are subscripted according to gate names, (e.g. W_{hf} is the hidden-forget weight matrix). 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 controls the duration of time dependency during sequence prediction. The cell state is the part of memory that propogates between hidden nodes. 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 used LSTM networks for the same binary classification task from Section III. LSTMs are able to extract the sequential information more naturally than SVM, so we expected to see better classification accuracy in the train and test datasets. We used Python and the *Tensorflow* library to implement the LSTMs.

Again, the full trajectories have been precompiled into snippets 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:

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

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

C. Hyperparameters

Again, there are many hyperparameters to optimize, including the number of hidden units, batch size, learning rate, and number of training epochs. It is very time intensive to search over all hyperparamters. We briefly summarize the hyperparameters' effects on performance, using validation accuracy as the performance metric. The output of our full grid search is in our posted code (file: rnn_classifier_hyperparameters.txt).

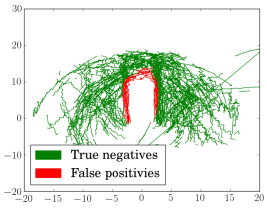
The best hyperparameters were batch size of 5, $n_{hidden} = 256$, $T = 10$, and 3 training epochs, and we used a learning rate of 0.001.

Performance generally levels off after 3-5 training epochs. For large T, more hidden units are required to handle the higher dimension of the input data. Specifically, when $T > 10$, $n_{hidden} = 8$ got $< 50\%$ accuracy, but performance improved with $n_{hidden} = 64$ (we ranged T from 2-50, and n_{hidden} from 8-256). Small batch size tended to improve the performance, but it was not a clear trend (we ranged batch size from 1-10).

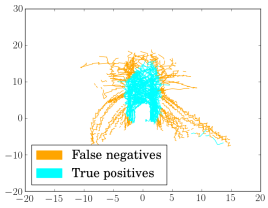
We also investigate the effect of training set size on performance in Fig. 6. The blue line corresponds to the LSTM classifier, which is not greatly affected by training set size (close to 80% throughout). The LSTM classifier is not as sensitive as SVM to training set size.

D. Results

After optimizing hyperparameters and confirming we have a sufficient amount of training data, we trained the LSTM binary classifier. The overall test accuracy was 80.4%.



(a) Trajs. labeled “No Cross”



(b) Trajs. labeled “Cross”

Fig. 10: Predicted labels on test set with LSTM classifier. On the left, trajectory snippets that were predicted label 0 (green is correct, red is incorrect). On the right, trajectory snippets that were predicted label 1 (cyan correct, orange incorrect). Across the whole test set, 80% accuracy was achieved. Compare with Fig. 7

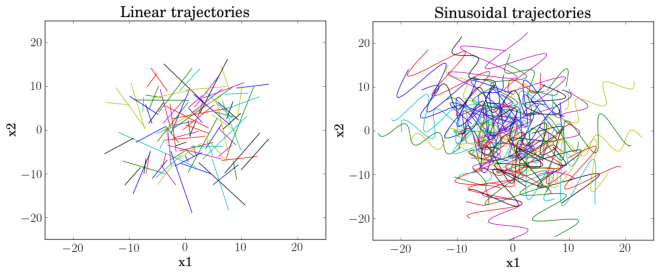


Fig. 11: Randomly generated trajectory data in linear (left) shape and sinusoidal shape (right). The trajectories have been generated by sampling the start point, range, scale, translation and rotation at random.

The results of the two classifiers (Fig. 7 & Fig. 10) look pretty similar. The LSTM classifier performed worse than the SVM classifier (84 vs. 80% test accuracy), which we did not expect. Both classifiers learned about the “cross zone”, perform relatively well on data that is predicted to be class 0, and struggle with false negatives far from the vehicle. The fact that both classifiers performed well/poorly on the same types of data indicates that the data itself was particularly challenging. Based on our observations of the training data set and classifier performance, it is likely that for many of the snippets that were far from the vehicle, there were similar snippets with each label, which is highly confusing to the classifiers.

V. Recurrent Neural Network: Prediction

In addition to classification, we also investigated applying LSTMs to the task of pedestrian motion prediction. That is, rather than a binary label, we try to predict *where* the pedestrian will walk next. This is much more challenging, but also would be more insightful to an end-user in vehicle applications.

A. Introduction

Existing research has been conducted in predicting trajectories in a 2D space. One relevant work used LSTMs to generate human handwriting from the IAM online handwriting database [7] [8].

LSTMs can be used to predict sequences, by feeding themselves their output as an input. To switch from binary classification to trajectory predicting LSTM, we change the network architecture from many-to-one to many-to-many. The output $y_t \in \Re^{T\text{x}2}$ is computed from every time step as given in (2).

Many works can be found for the task of text prediction (e.g. [9]). However, our application domain is real-valued, as opposed to the discrete-valued domain of word prediction.

The literature proposes several transformations, which could be applied to the output y_t. For example, [7] uses mixture density networks (MDNs) to calculate x_{t+1} with the conditional probability $P(x_{t+1}|y_t)$, with parameters of the distribution: mean μ_m, standard deviation σ_m, correlation ρ_m, mixture weights π_m for M mixture components.

Additionally, [7] used the relative position between timesteps $\delta x = x_{t+1} - x_t$ 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 an approaching car. For the MDN approach, our means μ_m would be further spread among the full absolute position space, which would presumably require even longer training time than the earlier approach. Therefore, we have decided to linearly compute the output, as stated in (2).

Our chosen loss function is the mean squared error of each predicted output and next ground truth input, over the whole time sequence. 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 we update the parameters in the activation cells via AdamOptimizer. We randomly draw batches from our training dataset for each parameter update to break dependencies of neighboring trajectory snippets in the training set.

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, pedestrian trajectories are highly complex nonlinear functions. To validate our LSTM prediction model, we have instead tested on simpler functions; we generated linear and sinusoidal shaped trajectories, displayed in Fig. 11.

For our hyperparameter tuning, we 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 finite, we have evaluated most of the parameters, while holding the other parameters constant. Optimal tuning would

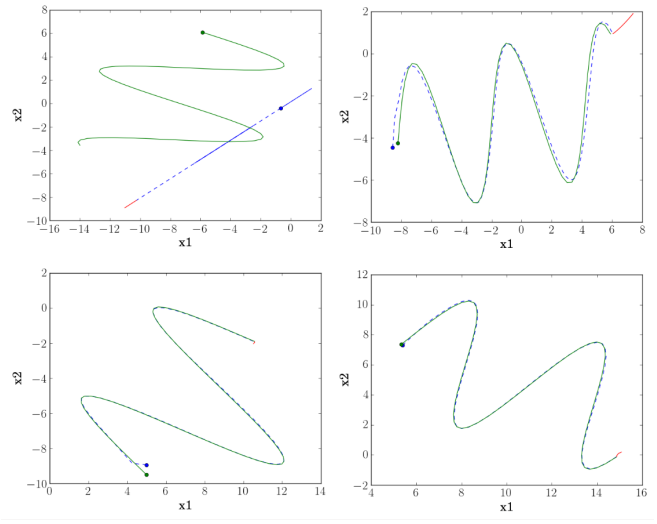


Fig. 12: Search over the number of hidden units: $n_h = 1$(top-left), $n_h = 4$(top-right), $n_h = 32$(bottom-left), $n_h = 128$(bottom-right). The plot displays the input trajectory (green) x_t with total trajectory snippet length $T = 50$ an its starting point (green-dot). The LSTM predicts the next 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) for $T_p = 10$ timesteps. All trajectories are test trajectories and have not been included in the training set. LSTM was also able to predict for linear trajectories – we have chosen to show the more interesting sinusoidal curves. The first blue dot y_0 purely based on the input x_0, and therefore deviates from the green curve.

run a search over all combinations, which would result in better performance.

As seen in Fig. 12, the number of hidden units $n_h = 1$ is insufficient to represent the complexity of sinusoidal curves. We can also see that the complexity of $n_h = 4$ is sufficient to approximately fit the sinusoidal shape, and $n_h = 32$ is sufficient to fit the data accurately upon visual inspection.

In Table IV we see that the loss shrinks with an increasing number of hidden nodes n_h. The training time dramatically increases with the number of hidden units. A high number of hidden units corresponds to high model complexity (and high bias), which could lead to overfitting on the training data. However, we do not observe symptoms of overfitting, as accuracy is similar on training and test data. The more complex the model, the fewer epochs it took to reach an acceptable loss value. Moving forward, we refer to the models with the hyperparameters from Table IV as 1-, 4-, 32- and 128-unit model.

We used early stopping (based on a validation dataset) to prevent overfitting. The training was stopped when the difference in average batch loss between two consecutive epochs $\delta L_b < 0.001$.

The model needs a minimum snippet length to be able to estimate the curve. For a low snippet length ($T < 10$) our loss increased ($Loss_b = 0.02$ for model $n_h = 4$ from Table IV). An increase in the chosen snippet length above $T = 50$ did not improve loss.

	$n_h = 1$	$n_h = 4$	$n_h = 32$	$n_h = 128$
$L_e = 0$	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 learning rate lr influences the rate of convergence. The higher lr, the faster the model should converge. However, too high values of the learning rate can cause an oscillation/divergence. In our case, all learning rates in the interval $[0.1, 0.01, 0.001, 0.0001]$ led to a reasonably low loss. $lr = 0.1$ made the loss oscillate in the interval $[0.01, 0.07]$ for our 32-unit model. A learning rate of $lr = 0.01$ for the 32-unit model showed the best tradeoff in terms of minimal loss, while maintaining fast training.

Our choice of batch size b impacted the speed of convergence (because tensorflow is able to execute efficient matrix operations), but did not impact the achievable loss given a fixed model and infinite possible epochs. Given a dataset of size $N = 5000$, we have chosen the batch size to be $b = 100$.

The LSTM has been designed to predict an unseen pedestrian trajectory for a given number of time steps. Previous work has shown that an LSTM is generally possible to predict a sinusoidal curve into the future [10]. However, they use x_1 as the input and predict the x_2 value. This is not directly possible in our case, as we have to predict in a local frame with respect to the car. Therefore, we predict both variables at the same time $x = (x_1, x_2)$, making the learning task more complex. Additionally, previous work uses more complex models ($n_h > 150$) with more training data ($N > 500000$) and lower learning rate ($lr < 0.0005$). An iteration over other hyperparameter choices with these values is not feasible for the scope of this project. However, the work suggests that further optimization of the hyperparameters would make it possible to predict trajectories better than displayed in Fig. 12.

We conclude that our model is able to predict one step into the future accurately, given our achieved average batch loss $L_b < 0.006$. However, an accurate prediction over many timesteps requires more computational power for hyperparameter tuning.

C. Real pedestrian data

Our LSTM is able continuously predict the next timestep in a domain of translated, rotated, scaled and stretched sinusoidal trajectories. The real pedestrian trajectories, however, are highly nonlinear, but should still be possible given the capabilities of LSTMs.

We have trained ($N \sim 3000$), validated ($N \sim 3000$) and tested ($N \sim 2000$) on distinct datasets. All hyperparameters have been initialized based on the results of the sinusoidal

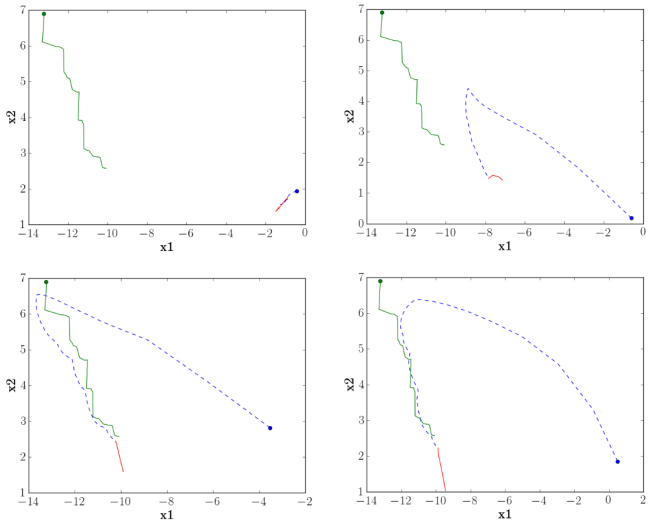


Fig. 13: Predictions on real pedestrian trajectory snippets. Colors are same as in Fig. 12. Search over the number of hidden units: $n_h = 4$(top-left), $n_h = 32$(top-right), $n_h = 128$(bottom-left), $n_h = 512$(bottom-right). The model with the highest complexity is able fit the pedestrian trajectories the best. The model is able to accurately predict the pedestrian's position one step ahead of time, given some observation time for convergence.

	$n_h = 4$	$n_h = 32$	$n_h = 128$	$n_h = 512$
L_{e_c}	50.3	21.8	4.6	1.8
L_{test,e_c}	18.46	4.43	0.81	**0.11**

TABLE V: Search over the number of hidden units n_h results in the best performance with a model with 512 units. L_{e_c} and L_{test,e_c} is the average batch loss after convergence on the training and test data, respectively.

trajectories and tuned for the new domain. Fig. 13 and Section V-C shows the performance of our LSTM on real pedestrian data.

The lowest error in the prediction of the consecutive state has been achieved with the highest complexity model with 512 hidden units. This confirms our assumption, that the real trajectories require more complex models to fit the data. The learning rate is chosen $lr = 0.001$ and the batch-size is $b = 100$.

The model is able to predict the next pedestrian timestep with an error of $\sqrt{0.11}m$ on the test dataset. As expected, our model was not able to predict pedestrian movement further than several timesteps. We expect better performance with more data and more extensive hyperparameter search. However, pedestrians behave randomly, do not follow predestined rules, and different pedestrians behave differently.

VI. Summary

In this project, we developed an SVM classifier that got 86.3% accuracy on pedestrian motion classification, and a LSTM classifier that got 80.4% accuracy on the same task. We also developed a LSTM-based motion predictor that accurately predicted future trajectories one step into the future, with a loss of $\sqrt{0.11}$m.

Division of Labor & Link to Code

Michael mostly worked on the dataset processing and SVM/RNN classifier optimization, and Björn did the RNN prediction/classifier implementations 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.

References

[1] J. Miller, A. Hasfura, S.-Y. Liu, and J. P. How, "Dynamic arrival rate estimation for campus mobility on demand network graphs," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[2] J. Miller and J. P. How, "Predictive positioning and quality of service ridesharing for campus mobility on demand systems," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.

[3] Anonymous, "CASNSC: A context-based approach for accurate pedestrian motion prediction at intersections," https://openreview.net/pdf?id=rJ26HSLRb, 2017, [Online; accessed 12-Dec-2017].

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[5] S. Rüping, "Svm kernels for time series analysis," 2001.

[6] Colah, "Understanding lstm networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015, [Online; accessed 12-Dec-2017].

[7] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[8] otoro, "Handwriting generation demo in tensorflow," http://blog.otoro.net/2015/12/12/handwriting-generation-demo-in-tensorflow/, 2015, [Online; accessed 12-Dec-2017].

[9] I. Sutskever, J. Martens, and G. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ser. ICML '11, L. Getoor and T. Scheffer, Eds. New York, NY, USA: ACM, June 2011, pp. 1017–1024.

[10] sunsided, "Tensorflow lstm sin(t) example," https://github.com/sunsided/tensorflow-lstm-sin, 2017, [Online; accessed 12-Dec-2017].