Machine Learning Project Report

AMPM: A Motion Prediction Model.

Ghizzo Samuele, Guéguen Juliette, Madaus Mason, Puchas Simon

University of Klagenfurt

February 6, 2025

Contents

1 Introduction

The main objective of this project is to implement a machine learning model for motion forecasting. Motion forecasting refers to the task of predicting the future position of a tracked object. This is a fundamental research problem in multiple fields. For instance, in autonomous driving, vehicles must anticipate the movement of surrounding traffic to ensure safe navigation. Another example is a robotic waiter in a restaurant who must avoid collisions while carrying plates.

The movement of objects, such as vehicles on the road or pedestrians in a restaurant, can be represented as time series data. Time series data consists of sequential records collected at fixed time intervals. Each time series comprises multiple time steps, where each step contains relevant information such as the object's position (x, y, z coordinates), orientation (roll, pitch, yaw), and other features. The goal of time-series prediction is to use the first n time steps to predict the object's position at future time steps $n+1, n+2, n+3$, and so forth.

Various machine learning models can handle this task; in this project, it was chosen to implement a Long-Short-Term Memory (LSTM) network due to its ability to process sequential data effectively. A key feature of LSTMs is the memory cell, which enables the model to capture and retain information from previous time steps, making it particularly well suited for time series forecasting.

The LSTM is trained on a custom dataset that was created using ROS (Robot Operating System), which is a framework used to simulate robot environments. More information about this will come in the following section.

In this project the goal was to use the past 10 time steps to predict the 11th. As simple as a task this may sound, it is still a very good way to understand and practice with the LSTM architecture and other important areas of machine learning. Additionally, this implementation can be further expanded, as it will be discussed in section 6 of this report.

2 Dataset

The dataset was created by Simon Puchas.

As already mentioned, the data was created using ROS. ROS is a framework for writing robot software. It provides tools and libraries for simulating, visualizing and controlling robotic systems. It was used in combination with Gazebo, a high-fidelity physics simulator, to simulate many robot movements and get the coordinates of the position of the robot in each time step of these movements.

The reason it was chosen to create a self-made dataset from scratch is that the few existing datasets available online did not have the best shape to pass to the model, or some key features were missing. More time would have been consumed cleaning and reshaping the data than the time invested in creating a dataset.

2.1 Data collection

In order to create and collect the data needed for the proposed project idea, the beginning point is the ROS workspace. This workspace is a specific folder structure used for ROS, as it allows to launch multiple scripts, Gazebo and different applications used to work with robots at the same time.

The scripts used to create the dataset are *robot_controller.py* and *data_recorder.py*.

2.1.1 Robot Controller

The controller script implements the logic for the movement of the robot. First the waypoints the robot will go to are specified, with a tolerance as Gazebo has some simulation unavoidable errors. Then the linear and angular velocity are set, where the linear velocity stays constant, but the angular velocity is dynamically changed based on the angle difference. The angle difference is computed from the robot heading and the target angle to reach the next waypoint.

Then the publishers and the subscriber are set. One publisher is used for passing the linear and angular velocity at which the robot needs to drive. The second publisher is less important as it is only used to shut down all publishers and subscribers once the last waypoint is reached. The subscriber constantly obtains the 6D-pose of the robot to calculate the angle difference and the distance to the next waypoint.

2.1.2 Data Recorder

In order to record and store the positional and velocity data of the robot, a data recorder script was created. This script has 3 subscribers, one for the 6D-pose, one for the velocity, and one that receives the shutdown message to stop recording.

Then the noise from z, roll, pitch and yaw is filtered, as it could negatively affect the model's performance. After that, the data is stored as tensors in a numpy array. The data is additionally stored in a csv file, to help with debugging and understanding how the data looks.

2.2 Dataset creation

After its collection, the data needs to be rendered in form of a dataset such that it can be exploited by a machine learning model. This is handled by the *dataset6_creator.py* script, which as the name suggests was the 6th data collection.

This script first loads the data recorded with the Data Recorder. Then, as each array contains one movement, movements that do not have at least 50 time steps are filtered out and all movements longer than 50 steps are cut such that they have exactly 50 steps. If a movement has 100 or more time steps, it will be split into 2 separate movements. This way all movement sequences are of the same length, which makes processing them and feeding them into the LSTM model much easier.

After this, sliding windows of size 10 are applied on all the movements, thereby creating 40 X windows and 40 y windows for each movement. Each X window contains 10 time steps and the corresponding y window contains only one time step, which is the time step following the last time step of the X window.

As the data is now given as X and y, normalization/standardization techniques are applied. NaN values are handled by a conditional that checks if the standard deviation is 0; this is needed as for example the z, roll and pitch data is always 0, because the robot is only moving horizontally, but in this way the project can be potentially expanded with other type of movements easily.

For the last step the data is split into 80% training dataset, 10% validation dataset and 10% test dataset. These are stored in a PyTorch dictionary:
$X\_train, X\_val, X\_test, y\_train, y\_val, y\_test$

3 Model

The model architecture was created by Samuele Ghizzo.

3.1 LSTM

The model chosen for this project is the LSTM. The LSTM is a model able to capture long-term dependencies, as the name suggests (Long-Short Term Memory), making it ideal for sequence prediction and, in this case, motion forecasting. The principle of the LSTM is to deal with the vanishing / exploding gradient problem of RNNs by saving the long-term memory and the short-term memory into two different states; the long-term memory is saved in the cell state c_t, while the short-term memory is saved in the hidden state h_t.

3.2 The LSTM cell

The LSTM cell is called a "gated structure"; this is because each cell has three gates, called the forget gate, the input gate and the output gate, each of them with a different purpose

3.2.1 The forget gate

The forget gate controls how much of the long-term memory is kept by passing the current input through a sigmoid function together with the hidden layer and bias, and then multiplying it by the previous cell state c_{t-1}. A smaller value means less relevance of the past memory, while a bigger value means more relevance. In other words, the forget gate decides what to delete from the past.

3.2.2 The input gate

The input gate decides how much of the current input and hidden state will influence the future long-term memory; in other words, decides what to add to the memory from the present.

3.2.3 The output gate

The output gate takes the updated cell state C_t, applies a $tanh$ function to scale its values between -1 and 1, and then multiplies this by o_t (the output gate's activation vector). This produces the hidden state. The hidden state h_t represents the short-term memory that is immediately relevant for making predictions or decisions at the current time step. In other words, the output gate decides how much of the current memory should be shared as hidden state.

3.3 Custom LSTM

The theoretical LSTM model was implemented practically in the CustomLSTM class, which comprises three functions: the constructor, the forward function,

and the weights initializer function.

3.3.1 __init__

The __init__ function initializes the LSTM model by defining the input and hidden sizes. A linear projection layer to adjust the input size to the hidden size is initialized, in order to handle also variable input sizes.

The weight parameters W, U, and biases are created: they are then called by the function *init_weights*() to be initialized. Finally, a linear layer to convert the projected predictions back into the original form is initialized.

3.3.2 forward

The forward function processes an input sequence through the LSTM. It takes an input tensor x with dimensions [batch_size, sequence_length, input_features]. If no initial hidden states or cell states are provided, they are set to zero.

The input is projected to match the hidden size, and the function iterates over the sequence length, computing the new cell state and hidden state. The final hidden state is passed through the fc layer which converts it back in a suitable form to generate a prediction. The function returns this prediction along with the last hidden and cell states.

3.3.3 init_weights

The *init_weights*() function initializes the model parameters by assigning values from a uniform distribution. The standard deviation is computed as $\frac{1}{\sqrt{\text{hidden\_sz}}}$, and all weight tensors are assigned values within [-stdv, stdv]

4 Training

The training was done by Mason Madaus, Juliette Guéguen and Simon Puchas.

4.1 Training loop

The DataLoader from PyTorch is used to load the data from the created dataset and divide it in batches.

The training loop iterates through the epochs and in each epoch the model is first set to training mode; the batches are moved to the GPU (if possible) and reshaped to match the expected input dimension of the LSTM by multiplying the batch size with the window size (since the LSTM expects an input of three dimensions). After that the reshaped batches go through the forward-pass which computes the prediction. The prediction is then used to compute the training loss. The gradient is set to 0 to remove previous gradients and the loss is used in the backward propagation to update the model parameters. Once the training for the current epoch is over, the accumulated loss is used to compute the average loss over all batches in the epoch.

After the training for the current epoch is done, the training loop continues with the evaluation by first setting the model to evaluation mode and disabling gradient computation. The validation mode works the same way as the training loop explained earlier. The only differences are that, as already mentioned, the gradient computation is disabled, because the validation should not update the model parameters, and the validation data is used instead of the training data.

At last, the training loss and validation loss of the current epoch is printed.

As the ReduceLROnPlateau scheduler is used, the learning rate is the only parameter being updated based on the validation loss.

4.2 Hyperparameter tuning

The hyperparameters of the model were tuned manually in order to improve the model's performance.

Batch sizes between 2 and 64 were tested, with the best results coming from a batch size being between 16 and 32.

For the learning rate, among the ones tried in the beginning, a value of 0.005 gave the best results. But then, to further improve the model, a scheduler was added with the goal to reduce the learning rate once the validation loss had been plateauing for five epochs.

It was experienced that the hyperparameters which introduced the most change were (as expected) the learning rate; further tuning the hyperparameters would not lead to any significant improvement of the model's performance.

5 Evaluation and discussion

The evaluation was done by Juliette Guéguen and Mason Madaus.

In order to evaluate the performance of the model, the *LSTM_evaluation_2.py* script is used. This script loads the pre-trained model and the test data from the dataset. Then the process of evaluation is the same as the training loop explained in subsection 4.1, but without computing the gradient, as the model should not be trained. Additionally to the test loss, the mean absolute error and mean squared error are computed as well.

Finally, everything is stored in a JSON file, which is divided into a field with the error metrics followed by a field for each individual movement. For each movement one can see the true values, the predicted values and then the difference, for each component of the movements. This makes the assessing of the model's performance straightforward.

Further visualization and analysis of the evaluation results, can be seen in the *Performance.ipynb* notebook.

Three metrics were calculated, the sum of squared differences, the maximum absolute error and the relative error for each component of the movements. Some notable conclusions can be drawn from the results:

- with a relative error of over 600% it can be seen that the model struggles most with predicting the angular velocity. This is most likely due to the fact that the angular velocity changes a lot depending on the movement of the robot. It can change rapidly between positive and negative values and between high and low absolute values;
- another interesting conclusion to draw from these metrics is also the fact that the sum of squared differences for z, roll and yaw is greater than zero. This means that even though the true values for those components are always zero, the model didn't remember those values, but tried to estimate them instead. It's an indicator that goes against an over-fitted model.

Lastly, using the sum of the absolute errors for every component, the best and worst predicted movements are highlighted. It was found that the overall angular velocity of the 5% best predicted movements was low. While it was much higher among the 5% of movements that were the most poorly predicted.

This corroborates what the previous findings had shown, i.e. the model tends to struggle to estimate angular velocity.

6 Conclusion and future improvements

This concludes the AMPM project done by Simon Puchas, Samuele Ghizzo, Mason Madaus and Juliette Guéguen.

As already briefly mentioned in section 1, this is a rather basic implementation of such a motion prediction model and there are lots of possible future improvements and expansions.

Following are some of them, which were partly in consideration for this project in the beginning, but due to lack of expertise and time to implement such a sophisticated system, they were discarded.

6.1 Improvements and Expansions

- one improvement and simultaneously an expansion would be to make the ROS implementation more sophisticated. This means to be able to create more complex movements bringing in also vertical motion, as well as making the logic of storing the data better, such as storing everything in tensors and leaving numpy arrays completely out.
- one could try a deeper model architecture for using more data as well as more complex data.
- a transformer architecture could be used and compared to the LSTM model, possibly delivering even better performance.
- one could also expand the prediction to not only predict one time step, but rather predict 2, 3 or more time steps, which would be much trickier for the model than just prediction one.
- finally, for the representation of the evaluation results, it would be interesting to see a ROS implementation, where the model is predicting the future position of the robot or a different object in real time and having a visualization of that.