University of BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

Using Sensor Fusion and Deep Learning to Improve Activity Tracking

Harry Waugh

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Thursday 9th May, 2019

Declaration

Harry Waugh, Thursday 9th May, 2019

Acknowledgements

This work could not have been completed without the guidance and insight provided by Dr. Sion Hannuna, who has continually inspired me over my time at the University of Bristol.

I would like to thank my family, and particularly my parents for their support and confidence in me throughout my life. They have provided me with opportunities which have brought me here today.

Abstract

Using Sensor Fusion and Deep Learning to Improve Activity Tracking

In 2018, 6.6 billion miles of activity data was uploaded to the Strava platform [1]. People use this platform to compete against family and friends, the ambitious can even pit their abilities against elite athletes. This provides a strong motivator for them to go out and improve their fitness, hence it is important for them to get accurate data. Users are often frustrated that after completing their activity, the recorded GPS route differs from the route they actually took. Recorded routes can often suggest that the user has run through buildings or perhaps cycled across a body of water. This inaccuracy can be accounted for by noisy, low frequency GPS data.

This paper aims to find a solution to this problem by utilizing the sensors found in an Inertial Measurement Unit (IMU), such as accelerometers, gyroscopes and magnetometers. It explores different methods that aim to strike a compromise between the low frequency GPS sensor and the high frequency IMU sensors that are subject to drift over a long time frame.

In particular, this paper discusses methods for two use cases that are relevant to users. The first assumes that the correction of the activity data is performed in ‘real time’ during the activity. The consequence of this means that correction algorithms must be computationally efficient and may only make use of past data. The second assumes that the activity data is corrected after the activity has been completed, giving algorithms more scope to leverage past and future measurements when computing estimates of position.

The Kalman Filter is relevant for both of these use cases. This classic 1960’s technique can be used to combine sensor readings, only requiring knowledge of the previous position and current sensor readings. A reverse pass of this algorithm can also be computed after a completed activity to further refine the estimate.

Recurrent Neural Networks (RNNs) are a modern approach to correcting activity data and do not require explicit knowledge of sensor variances. Interestingly, there seems to be little or no research into using Recurrent Neural Networks as a means to improve GPS tracking for either vehicles or sports activities, even though they have had great success in other time-series data applications. Relatively new RNN architectures such as Long Short Term Memory Networks and Gated Recurrent Units can use built in memory modules to uncover trends and patterns in sequences of sensor data.

Both the Kalman filter and RNN yield qualitatively plausible results, when overlaid on maps depicting GPS data. These showed the RNN outperforming the Kalman filter in all cases. Quantitative evaluation, however, was problematic, given the unavailability of reliable ground truth. A detailed discussion of the issues surrounding the ground truth limitations is provided, and solutions are also proposed for future work.

By developing this paper, I have achieved the following:

- I have spent 15 hours collecting my own sensor recordings and obtaining the appropriate ‘ground truth’. This data-set comprised of mixture of walking, cycling, and running data, and is the first

data set of its kind. The collection of this data-set was fraught with challenges, and many sensor recording apps were not fit for purpose. Often this was found after custom parsing functions were written for different file formats.

- I have taught myself the theory of the Kalman filter and then adapted it for improving on activity tracking. I have demonstrated this knowledge by coherently describing how this works in the context of the project.
- Having taken the 3 year BSc, I did not have the opportunity to study the Deep Learning course and so have spent 20 hours researching and teaching myself about Recurrent Neural Networks. This has enabled me to confidently detail how they work and can be applied.
- I have learned how to use popular machine learning libraries and frameworks, Tensorflow and Keras, in order to implement both LSTM and GRU Recurrent Neural Network architectures.
- I applied and successfully entered into the Oracle Innovation Accelerator program, this empowered my research with the use of an Oracle Cloud Compute Instance. I learned how to correctly set up this instance for remote training and evaluation of my Recurrent Neural Network.
- I am confident that I have produced an algorithmic pipeline that can produce real testable improvements, given a more accurate data-set with a reliable 'ground truth'.
- I have spent time discussing the benefits and caveats of different ways to obtain 'ground truth' data. I have gathered and shown insight into where different approaches will be successful, and proposed future work that would be possible with greater resources.

Contents

Chapter 1

Introduction

1.1 Introduction

As of 2019, there has been some research into improving GPS tracking using Inertial Measurement Unit (IMU) sensors. Sensor fusion techniques such as the Kalman Filter have mainly been used for correcting routes taken by cars and other motor vehicles [2, 3, 4]. However, these vehicles often have significantly more accurate sensors than those found in a smartphone. Another consideration is that activities such as running and walking are subject to a higher degree of noise than a car on a road, due to the up and down motion that occurs by taking a step on different terrains. Interestingly, there seems to be little or no research into using Recurrent Neural Networks as a means to improve GPS tracking for either vehicles or sports activities, even though they have had great success in other time-series data applications [5, 6, 7]

Clearly the small amount of research that has been done in this area coupled with the large potential user base (9,000,000 million running participants in the UK [8]), provides strong motivation to explore how GPS tracking can be improved with additional sensors.

This paper aims do this by using 3 distinct chapters, the first of which being the 'Background'. This chapter will give a comprehensive view of different sensors and the methods that combine these to improve GPS tracking. Furthermore, it introduces key terminology that relate to sensor fusion and recurrent neural networks, in order to provide an understanding of how these work and can be applied. There is also some discussion of current research in the area and state-of-the-art systems.

The following 'Implementation and Methodology' chapter describes the details of how the implementation was produced and the challenges in doing so. It walks through the steps taken to create the system, outlining the software and technologies used, and the reasons why each of these were chosen.

The final chapter delves into the accuracy and success of the solution that has been created. It evaluates both the Kalman Filter and the Recurrent Neural Networks performance, contrasting this to the GPS accuracy without the use of additional sensors. It also answers questions regarding the usability of the solution with respect to different use cases, and future directions for research.

1.1.1 Contributions

- A survey of various different methods that either correct or supplement GPS tracking. Each method is evaluated in terms of it's suitability for the real use-cases defined in 2.5.
- A data-set was collected specifically for this project. This data-set comprised of mixture of walking, cycling, and running data, and is the first data set of its kind.
- GPS and IMU sensors have widely differing frequencies, this paper discusses how these can be combined for use in a Kalman Filter and in a Recurrent Neural Network.

- Quantitative and Qualitative analysis is given about the success of both the Kalman filter and Recurrent Neural Network when improving activity tracking.
- An algorithmic pipeline that can produce real testable improvements, given a more accurate data-set with a reliable 'ground truth'.
- There is exhaustive discussion about the benefits and caveats of different ways to obtain 'ground truth' data.
- Proposals for future work have been given, with insight about where innovation is needed to obtain further success.

Chapter 2

Background

The first part of this chapter details how each of a smartphone's sensors work, and how this relates to their strengths or shortcomings of estimating position. It goes further by looking at state-of-the-art positioning systems that use these sensors.

This chapter also recognizes that utilizing more sensors isn't the only way to improve the accuracy of a recorded GPS route, exploring a popular alternative known as 'Map Matching'. This technique attempts to identify which path or road a device is travelling upon, using a database of interconnected road data.

The chapter proceeds by giving the theory behind a Kalman Filter and how this can be specifically applied to generate a better estimate of position. It discusses performing a reverse pass of the data once the activity has been completed, as a means to improve accuracy further.

Finally, this chapter culminates by giving an overview of the inner-workings of Neural Networks and how these have been adapted to give rise to Recurrent Neural Networks.

2.1 GPS

The Global Positioning System (GPS) is a common way to identify a smartphone's position on the Earth. GPS works by calculating position relative to the known positions of dedicated GPS satellites. Each GPS satellite will carry an atomic clock that is synchronized with each of the other GPS satellites, all continually transmitting their current position and the time of sending. The smartphone collates the signals of at least 4 different satellites in order to calculate 4 unknown values. These 4 values relate to 3 position coordinates (X, Y, Z) and the time deviation from satellite time. These values can be calculated because the time for the signals to propagate to the smartphone are proportional to the distance between the smartphone and the satellite [9].

2.1.1 Limitations of GPS

There are several limitations of GPS that lead to errors, including poor signal, Geometric Dilution of Precision and Multipath Errors. Signals sent from GPS satellites cannot propagate through water, walls or other obstacles easily. This is what causes poor GPS signal, which is dependent on having direct 'line of sight' with at least 4 of the 24 GPS satellites [10]. This presents particular ramifications for this paper, as runners and cyclists are known to frequent urban and woodland areas. These areas have high buildings and trees that can cause signal fragmentation which stops signals being received by a smartphone [11].

Geometric Dilution of Precision (GDOP) is a more complex issue with GPS which relates to how the

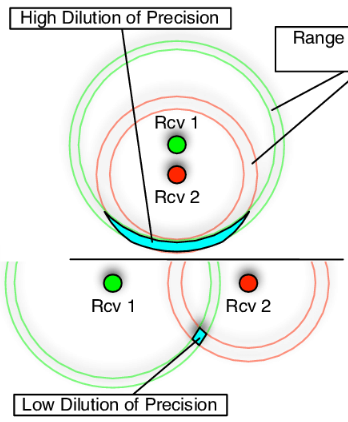

High Dilution of Precision
Range of Measurement Errors
Rcv 1
Rcv 2
Rcv 1
Rcv 2
Low Dilution of Precision

Figure 2.1: Examples of Poor and Good GDOP

GPS satellites are arranged in the sky. Satellite constellations that are spread out offer a good GDOP, whereas closely packed satellites have a poor GDOP. The effect of GDOP on GPS can be explained by considering 1 satellite sending a signal to a GPS enabled smartphone. The smartphone can calculate the distance to the satellite but doesn't know the direction in which the satellite is. Therefore, all the points at which the smartphone may reside, form a circle upon the Earth's surface within a degree of error. By considering the same scenario with 4 satellites, 4 circles are formed on the Earth. The intersection of these is then where the smartphone is located.

If the 4 satellites have a good GDOP, the circles formed will be relatively spread out and the intersection will be a clear point. This means the smartphone's position will be known to a high degree of accuracy and there will be a low dilution of precision. Conversely, if the satellites are positioned closely together, the circles formed will be very similar and overlap. This causes a high dilution of precision and has the potential to affect the GPS accuracy of the smartphone by several meters. An illustration of these two scenarios is shown in Figure 2.1. This error causes substantial problems for people recording activities in built-up areas with tall buildings. This is because each satellite must be in a small area in order to be in sight of the smartphone.

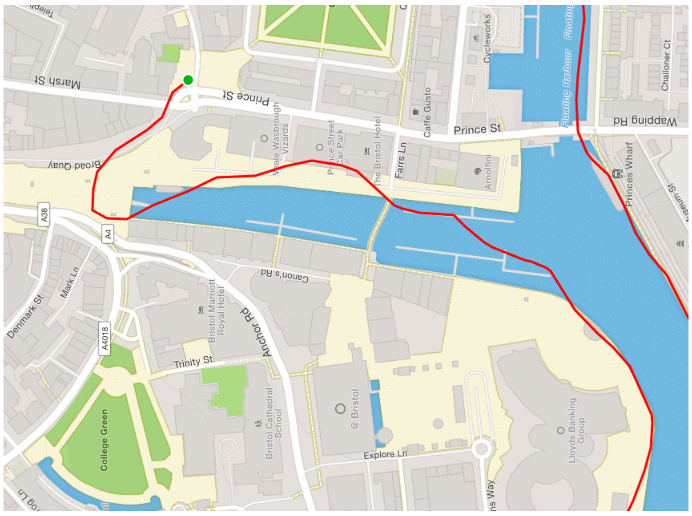


Figure 2.2: Images showing faulty GPS Recordings

Multipath errors are also an issue and are caused when the GPS satellite signal is reflected before reaching the GPS device. The signal taking the reflected route has a longer propagation time than expected, which affects distance calculations on the device [12]. This can happen when sports activities take place close to bodies of water or when there are buildings for the signal to reflect off. Figure 2.2 shows an example of a recorded GPS route that is subject to error.

2.2 Inertial Measurement Units (IMUs)

Inertial Measurement Units (IMUs) have their origins in rocketry where Robert Goddard experimented with basic gyroscopic systems as a means to create guided missiles. Nowadays, an IMU is a collection of sensors that measure the force, angular velocity and the magnetic field strength using an accelerometer, gyroscope and magnetometer. However, this is not strict and sometimes more sensors can be included to improve the calculations done by these sensors.

An accelerometer measures the acceleration of a device. This can be thought of as an object attached to a spring, where the distance the object moves is proportional to the force it experiences, and thus can be used to calculate acceleration. This is done along the 3 axes of the device to calculate each component of the acceleration.

The gyroscope that is found in smartphones is normally a Micro-Electro-Mechanical-System (MEMS) Gyroscope, which works by using a vibrating structure to determine the rate of rotation. Interestingly, a device only needs an accelerometer and gyroscope in order to determine its orientation and the magnitude at which it is accelerating. Despite this, another common sensor found in IMUs is a magnetometer, this measures the Earths magnetic field, and can be effective in finding the orientation of the device. It is important to note that magnetometers suffer from errors including magnetic declination, which occurs because magnetic north is no longer the same as true north.

Given the orientation, acceleration and starting location of the device, 'dead reckoning' can be used to calculate the current position of the device. 'Dead reckoning' is the process of continually calculating position based on observed speeds and the time since the last known position. Applications that utilize 'dead reckoning' often have limited access to GPS signal, or they require high security and need to be immune to electrical interference. Unfortunately, 'dead reckoning' involves integrating the acceleration twice, which magnifies small errors in the accelerometer readings. This becomes worse over time as errors accumulate, leading to rapidly occurring drift.

One security critical application of IMUs is aircraft, these need high accuracy navigation systems that can withstand the loss of GPS signal. Aircraft such as Boeing 787s use high precision technology such as quartz accelerometers and ring laser gyroscopes. Honeywell manufacture this sophisticated technology, they quote it is accurate to within 2 nautical miles ($\approx$ 2.3 miles) per hour of flight without a known location [13].

2.3 Smartphone Sensors

There are several ways that users record their GPS activities including GPS sport watches, but the most common is a smartphone that can directly upload to fitness platforms. Due to scope of the project, all data collection is performed using an Android smartphone. Android was chosen because in 2017 it had 2 billion monthly active users making it the biggest mobile OS in the world [14]. Of these devices Google says that 'most' devices have an accelerometer and many smart phones will have also have gyroscopes and magnetometers [15]. The choice of device that collects the sensor data is important to the paper, because it ensures that results from this paper can be applied to other smartphones. Therefore, it doesn't feature any specialist activity tracking hardware that GPS watches may be equipped with.

Although Android smartphones normally only have 3 physical sensors making up their IMUs, the Android OS does provide other composite 'software' sensors. These are computed using underlying combinations of the 3 physical sensors. 'Software' sensors that are relevant to this paper include the linear acceleration sensor, which is the acceleration sensor excluding the acceleration due to gravity ($\approx 9.8ms^{-2}$). Another relevant sensor is the rotation vector sensor that describes the orientation of the device in quaternion coordinates. This sensor is used in subsection 3.2.5 to convert acceleration so that it is relative to the Earth instead of the device.

Unsurprisingly, the accuracy of sensors found in an Android device are far lower than those used in

Boeing aircraft (section 2.2). However, for the purpose of this paper the IMU is not expected to be singly responsible for calculating the current position, and will be supported by GPS readings. Furthermore, there are still several advantages to combining the IMU sensors with those of the GPS, due to the frequency disparity between the two. Typically smartphone GPS sensors have a frequency of $\approx$ 1Hz compared with the $\approx$ 1600Hz that accelerometers and gyroscopes provide.

GPS accuracy is easier to quantify and is defined by the US Dept. of Defense as approximately 4.9m (16ft) for a typical smartphone [16]. Although, this does significantly worsen around buildings, bridges and trees. There is little in terms of specifications for the accuracy of IMU sensors in an Android device, which presents problems for the Kalman filter. This dependency on knowledge of IMU sensor variances is discussed later, alongside solutions to this problem.

2.4 Map Matching

An alternative to using extra sensor data is to leverage map data using ‘Map Matching’ algorithms. ‘Map matching’ is a way of linking geographic coordinates to meaningful pathways such as roads, cycle ways, etc [17].

An elegant ‘map matching’ algorithm was proposed in 2009 by Paul Newson and John Krumm, who wrote a paper [18] detailing the use of ’Hidden-Markov-Models’ in combination with the Viterbi algorithm. The Viterbi algorithm given a number of observed states, attempts to find the most likely sequence of hidden states. In this paper’s case, observed states would be GPS coordinates and hidden states would be street segments. An advantage of this method is that it accounts for noise in the GPS data and the layout of the road network.

Unfortunately, Geographic Information Systems (GIS) do not have knowledge of every road, let alone every path that a runner could be travelling on. Therefore, ‘map matching’ may not identify any paths that the runner has taken and thus there will be no improvement over GPS. An even worse scenario is where there is a nearby path that the runner did not take, but which the algorithm identifies as the route taken. This could lead to a final result that is worse than the original GPS route.

Another problem occurs when cyclists are travelling down a very wide road, the ‘map matching’ algorithm may then identify the cyclist as travelling down the other side of the road. This could be as much as 10-15 meters error, which will accumulate as more roads are included in the route.

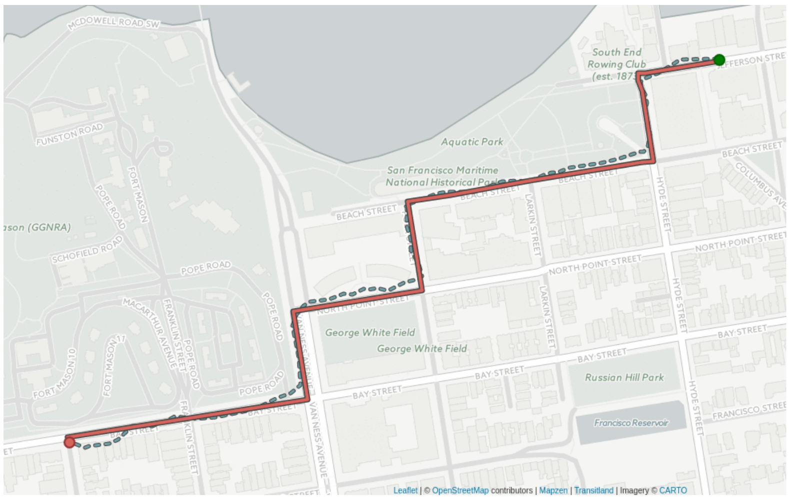


Figure 2.3: Map showing the GPS recorded route (dashed blue line) and the estimated route (red line) given by the map matching algorithm.

It is important that solutions provided by this paper work in areas where there is limited path data, therefore this paper will not explore ‘map matching’ any further. For completion, the rest of this section looks at how ‘map matching’ could work with limited path data in the future. If this is not of interest, please skip to section 2.5.

As mentioned, a fundamental problem with map matching is that it relies on accurate path data. This large amount of data is hard to obtain, and harder still to maintain. However, this problem could be significantly reduced in the near future, and is in the early stages of development by Strava. Code-named project ‘Slide’ [19], this algorithm makes use of a huge data set made up of all of their user’s recorded GPS activities. These then create a heat-map of different segments that are run regularly by users. This allows them to identify a route that is not found in a GIS but is popular with users. The massive advantage of this is that the upkeep of path data is taken care of as users switch to travelling on the new path. It is important to note that although this would work for popular routes, this would still not work if a route was being followed for the first time.

2.5 Use Cases

To develop techniques that improve on GPS inaccuracies when activity tracking, it is sensible to first define the use cases of the problem. This paper identifies two distinct use cases for how a user may go about correcting their recorded GPS activity.

- The first use case assumes that the user would prefer to correct their activity while it is being recorded, this paper defines this to be the ‘real time’ use case. The ‘real time’ use case enforces that algorithms must correct the current estimate position using only sensor readings recorded so far during the activity. The correction must also be efficient to compute on a smartphone while recording the activity. This use case is advantageous to the user as it allows them to see the corrections live on the phone, which could help them navigate when lost. A disadvantage of this use case is that further corrections may be possible if all activity data is known.

- This leads to the second use case, which is known as the ‘post processing’ use case. This assumes that ‘real time’ correction is not relevant to the user, and that all activity corrections take place post-activity. This grants algorithms extra scope by giving them access to all sensor readings. A key benefit of this approach means that an algorithm can leverage past and future sensor readings when estimating a corrected position in the activity.

2.6 Chosen Algorithms

This section discusses the chosen algorithms that attempt to improve the accuracy of activity tracking, and the reasons behind these choices.

A classic sensor fusion algorithm that has been around since 1960 is the Kalman Filter. Originally proposed by Rudolph E. Kalman [20], the Kalman Filter is well suited to improving GPS tracking for each of the use cases defined in 2.5. The estimated positions only require knowledge of the previous position and the current sensor measurements given by the smartphone. This makes it appropriate for ‘real time’ correction.

Additionally, the Kalman filter estimate can be further refined by reversing the activity data and recomputing the filter after the completion of the activity. This ‘post processing’ measure utilizes future data to calculate a past position. Forward and reversed routes can then be combined to obtain an improved estimate, details of this are found in section 2.7.2.

A more modern technique for predicting sequential data is to use a deep learning approach. Several methods were considered including the use of encoders and decoders, however these only complied with the ‘post processing’ use case and were not suitable for ‘real time’ correction on a smartphone. The most appropriate deep learning method was identified as Recurrent Neural Networks (RNNs).

RNNs are capable of learning features and long term dependencies from sequential and time-series data, thus appropriate for a sequence of sensor readings. RNNs work by being trained using large amounts of input and 'ground truth' training pairs, and can be run in 'real time' or as 'post-processing'. This paper will use architectures such as the Long Short Term Memory Network and Gated Recurrent Unit which have shown significantly high performance in a variety of applications [21].

2.7 The Kalman Filter

The Kalman Filter is an effective algorithm in sensor fusion for recursively filtering data in order to predict past, current and future states of a system. Since being proposed in 1960 it has been used for a large number of applications, the most prominent being the Apollo program and getting man to the moon [22]. This section looks at how the Kalman filter can be specifically applied to estimating GPS positions given observations from both the linear accelerometer and GPS sensors. It does this with respect to both the use cases defined in section 2.5.

2.7.1 Real-Time Corrections using a Kalman Filter

When constructing the Kalman filter, the state vector (x_k) must be defined. For this paper, the current state stores the current position (X, Y) and velocity (X', Y'). The Kalman filter also keeps track of the covariance of the state vector. This is known as the error covariance (P_k) and captures the uncertainty in the state vector estimate.

The system state vector, x_k.

$$x_k = \begin{bmatrix} X \\ Y \\ X' \\ Y' \end{bmatrix}$$

The error covariance, P_k.

$$P_k = \begin{bmatrix} \Sigma_{XX} & \Sigma_{XY} & \Sigma_{XX'} & \Sigma_{XY'} \\ \Sigma_{XY} & \Sigma_{YY} & \Sigma_{YX'} & \Sigma_{Y'Y'} \\ \Sigma_{XX'} & \Sigma_{YX'} & \Sigma_{X'X'} & \Sigma_{X'Y'} \\ \Sigma_{XY'} & \Sigma_{YY'} & \Sigma_{X'Y'} & \Sigma_{Y'Y'} \end{bmatrix}$$

The next step is to define the state transition model matrix, F. This represents the mapping between the previous state and the current one, and will use the velocities in the previous state to update the position in the current state. F can be derived using the following Newtonian equations of motion:

$$\begin{bmatrix} X_k \\ Y_k \end{bmatrix} = \begin{bmatrix} X_{k-1} \\ Y_{k-1} \end{bmatrix} + \Delta t \begin{bmatrix} X'_{k-1} \\ Y'_{k-1} \end{bmatrix}$$

$$\begin{bmatrix} X'_k \\ Y'_k \end{bmatrix} = \begin{bmatrix} X'_{k-1} \\ Y'_{k-1} \end{bmatrix}$$

Combining these equations gives,

$$\begin{bmatrix} X_k \\ Y_k \\ X'_k \\ Y'_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_{k-1} \\ Y_{k-1} \\ X'_{k-1} \\ Y'_{k-1} \end{bmatrix}$$

$$x_k = F x_{k-1} \tag{2.1}$$

F is also used to calculate the next state's error covariance P_k, which can be calculated using the following equation:

$$P_k = F P_{k-1} F^{-1} \tag{2.2}$$

The linear acceleration values are then incorporated into the Kalman filter by using a control input vector, u_k.

$$u_k = \begin{bmatrix} X''_k \\ Y''_k \end{bmatrix}$$

The Kalman filter then defines the relationship between equation 2.1 and the control input vector. This is done using a control input model matrix, B, which can also be derived using the Newtonian equations of motion.

$$\begin{bmatrix} X_k \\ Y_k \end{bmatrix} = \begin{bmatrix} X_{k-1} \\ Y_{k-1} \end{bmatrix} + \Delta t \begin{bmatrix} X'_{k-1} \\ Y'_{k-1} \end{bmatrix} + \frac{1}{2} \begin{bmatrix} X''_{k-1} \\ Y''_{k-1} \end{bmatrix} \Delta t^2$$

$$\begin{bmatrix} X'_k \\ Y'_k \end{bmatrix} = \begin{bmatrix} X'_{k-1} \\ Y'_{k-1} \end{bmatrix} + \Delta t \begin{bmatrix} X''_{k-1} \\ Y''_{k-1} \end{bmatrix}$$

Combine these two equations and use equation 2.1 to define B.

$$x_k = Fx_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} & 0 \\ 0 & \frac{\Delta t^2}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} X''_{k-1} \\ Y''_{k-1} \end{bmatrix}$$

$$x_k = Fx_{k-1} + Bu_{k-1} \tag{2.3}$$

Equation 2.3 can be extended in order to account for the assumed noise in the system. This is done by adding a parameter Q, or the covariance of the process noise. This noise has mean 0 and is assumed to be gaussian. The calculation of Q is discussed in section 3.3.2.

$$P_k = FP_{k-1}F^{-1} + Q \tag{2.4}$$

Equations 2.4 and 2.2 are the only equations that need be implemented in order to predict the next system state. Because of this, these are known as the predict step of the Kalman filter.

The second step of the Kalman filter is known as the update step. This is responsible for updating the predictions of the current system state based on the observed state, z_k. The observed states are GPS sensor readings.

$$z_k = \begin{bmatrix} X_{GPS} \\ Y_{GPS} \end{bmatrix}$$

The uncertainty in z_k is represented by the covariance matrix R, which is intuitively known as the observation noise. The relationship between the predicted system state, x_k, and the observed state, z_k, is the observation model, H. H is the mapping from the predicted state space to the observed state space. As this paper is only interested in correcting the position in a GPS activity, the observation model is the identity matrix with the bottom two rows removed. This selects the position values and disregards the current velocity values:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The Kalman filter then finds the updated position using both the predicted and observed positions. It does this by modelling the predicted and observed positions as multi-dimensional gaussian distributions, and calculating the joint distribution of the two. This can be thought of as finding a compromise between the accelerometers predicted position and the one observed by the GPS sensor, an illustration is also shown in Figure 2.4. Each distribution is defined below:

The predicted position distribution (mapped into the observation space):

$$N(\mu_0,\ \Sigma_0) = N(Hx_k,\ HP_kH^T) \tag{2.5}$$

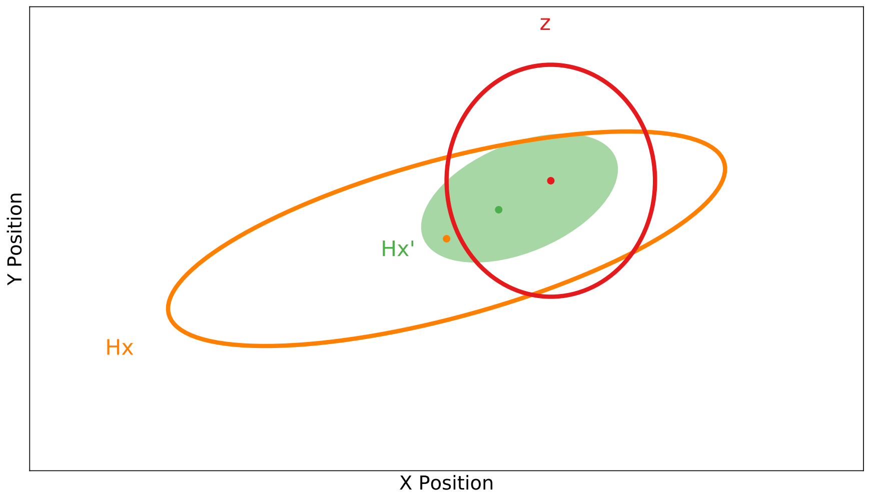


Figure 2.4: Illustration of predicted (Hx_k) and observed (z_k) position distributions. Their joint distribution, or the updated position distribution (Hx'_k) is also shown.

The observed position distribution:

$$N(\mu_1,\ \Sigma_1) = N(z_k,\ R) \tag{2.6}$$

The updated position distribution:

$$N(\mu',\ \Sigma') = N(Hx'_k,\ HP'_kH^T) \tag{2.7}$$

Now that these distributions have been defined, the updated position can be calculated. This is done using formulae for computing a joint distribution (Equations 2.8-2.10).

$$N(\mu',\ \Sigma') = N(\mu_1,\ \Sigma_1) \cdot N(\mu_0,\ \Sigma_0) \tag{2.8}$$

$$\mu' = \mu_0 + \frac{\Sigma_0(\mu_1 - \mu_0)}{(\Sigma_0 + \Sigma 1)} \tag{2.9}$$

$$\Sigma' = \Sigma_0 - \frac{\Sigma_0\Sigma_0}{(\Sigma_0 + \Sigma 1)} \tag{2.10}$$

The Kalman Gain, K, is defined to be:

$$K = \frac{\Sigma_0}{(\Sigma_0 + \Sigma 1)} \tag{2.11}$$

This reduces equations 2.9 and 2.10 to:

$$\mu' = \mu_0 + K(\mu_1 - \mu_0) \tag{2.12}$$

$$\Sigma' = \Sigma_0 - K\Sigma_0 \tag{2.13}$$

Hence, by substituting the means and variances from equations 2.5-2.7:

$$Hx'_k = Hx_k + K(z_k - Hx_k)$$

$$HP'_kH^T = HP_kH^T - KHP_kH^T$$

$$K = HP_kH^T(HP_kH^T + R)^{-1}$$

These three equations can be simplified further by pre-multiplying all of them by H^{-1} and post multiplying the second one by $(H^T)^{-1}$. This changes the Kalman gain which becomes denoted K'.

$$x'_k = x_k + K'(z_k - Hx_k) \tag{2.14}$$

$$P'_k = P_k - K'HP_k \tag{2.15}$$

$$K' = P_k H^T (HP_k H^T + R)^{-1} \tag{2.16}$$

This section has been given as a derivation of the specific Kalman filter used. However, it is important to realize that only these equations from the predict (2.2, 2.4) and update stages (2.14-2.16) are required for implementing a 'real time' Kalman filter.

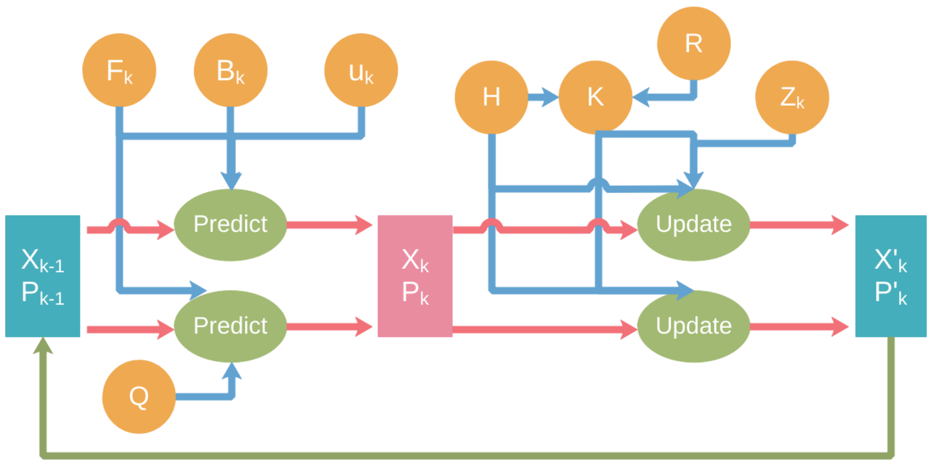


Figure 2.5: Illustration showing the Information Flow of the Kalman Filter.

2.7.2 Post-Processing Corrections using a Kalman Filter

As a means to further improve the estimate given by subsection 2.7.1, the Kalman Filter can also complete a reverse pass of the sensor data. Obviously, this must be done after the activity has been recorded which fits in with the 'post-processing use case of this project.

To perform a reverse pass of the Kalman filter, there are two possible options. The first is to build a new Kalman filter that changes the state transition and control input matrices to account for the reverse direction of the data. However, a simpler alternative that reuses the current Kalman filter, is to just reverse the data that is given to the filter.

The data is relatively easy to reverse given that the acceleration and GPS data are in the following matrix form:

$$Acceleration = \begin{bmatrix} Time & X' & Y \\ 0 & x_0 & y_0 \\ .. & .. & .. \\ t_{n-1} & x_{n-1} & y_{n-1} \end{bmatrix} GPS = \begin{bmatrix} X' & Y' \\ x'_0 & y'_0 \\ .. & .. \\ x'_{n-1} & y'_{n-1} \end{bmatrix}$$

The first step is to reverse the order of the rows in both the acceleration and GPS matrices.

$$Acceleration = \begin{bmatrix} Time & X' & Y \\ t_{n-1} & x_{n-1} & y_{n-1} \\ .. & .. & .. \\ 0 & x_0 & y_0 \end{bmatrix} GPS = \begin{bmatrix} X' & Y' \\ x'_{n-1} & y'_{n-1} \\ .. & .. \\ x'_0 & y'_0 \end{bmatrix}$$

The acceleration data must then be negated, this represents the change in the acceleration's direction, which is now the opposite at each moment in time. There is no change in the GPS data, as it already represents the reverse list of GPS coordinates.

$$Acceleration = \begin{bmatrix} Time & X' & Y \\ -t_{n-1} & -x_{n-1} & -y_{n-1} \\ .. & .. & .. \\ 0 & -x_0 & -y_0 \end{bmatrix}$$

Finally, the time column in the acceleration matrix is corrected by adding t_{n-1} to each value.

$$Acceleration = \begin{bmatrix} Time & X' & Y \\ 0 & -x_{n-1} & -y_{n-1} \\ .. & .. & .. \\ t_{n-1} & -x_0 & -y_0 \end{bmatrix}$$

Once the Kalman filter has completed a pass of the reversed sensor data, it must be combined with the Kalman Filter estimate using the forward data. To combine these two estimates, the mean position at each timestep is calculated.

2.8 Deep Learning and Neural Networks

Recurrent Neural Networks (RNNs) have had great success in many applications from speech and handwriting recognition, to learning how to tie knots during surgery [5, 6, 7]. These applications highlight the power of applying RNNs to a problem, and this section looks at how RNNs can be successfully applied to improving GPS tracking using IMU data.

The section begins by giving an overview of neural networks, which acts as a brief introduction to the topic for the uninitiated. Further detail is then provided by introducing some RNN terminology to explain a fundamental building block of neural networks, the Single Layer Peceptron. It develops on this concept to explain the workings of Artificial Neural Networks. In turn, these are then used to describe 'vanilla' RNNs and more advanced architectures like LSTMs and GRUs. At each stage of development, the network will be related to how it can improve GPS activity tracking.

The rationale behind the structuring of this section aims to give readers an understanding of how RNNs work in the context of improving GPS. Hopefully, it will also serve as an insight as to why RNNs were chosen for this purpose and the advantages of this choice.

2.8.1 Neural Networks - Overview

Neural Networks are an attempt to replicate the learning abilities shown by animals, which is done by modelling the brain as a network. These networks undergo training by being fed various different input and 'ground truth' pairs, this allows the network to identify patterns and learn to predict future inputs. The network consists of a series of connected nodes, these nodes represent the neurons of a brain and the connections model the synapses between them.

The network 'learns' when nodes interact with each other, or pass signals through the network. Each node controls whether a signal passes through it by subjecting it to a certain threshold. If a signal is above a node's threshold value, it triggers the activation function resulting in the node becoming active. This allows the network to be trained by using different weight and bias values that transform the signal at different connections. The network manipulates these weight and bias vectors in order to find an optimum for the system.

If applied to improving GPS tracking, the neural network's aim is to correct a recorded GPS location using sensor data. The network would complete this task after being trained on a large amount of recorded GPS locations and their corresponding 'ground truth' locations.

2.8.2 Single Layer Peceptrons

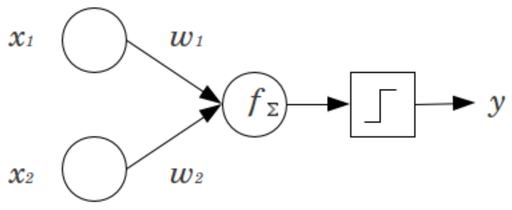


Figure 2.6: A Single Layer Peceptron (SLP) [23], It comprises of a single node with two inputs (x_1, x_2) which are multiplied with weights (w_1, w_2). These are summed in the node and then subject to an activation function which returns an output y.

Possibly the most simple neural network is the single layer perceptron (SLP), which is shown in Figure 2.6. Formally, inputs to this are known as feature vectors and the node calculates the dot product of the weight and feature vectors. The node then subjects the resulting value to an activation function.

The activation function returns 1 or 0, which corresponds to the node being active or inactive respectively (Equation 2.17). The output of the node is known as the output vector, and the aim of the network is to learn the function or mapping from the feature vector to the output vector.

In this paper, The feature vector corresponds to the sensor data, and will contain information such as the GPS, acceleration and angular velocities at each timestep. Whereas, the output vector is the ‘ground truth’ position at each timestep. Therefore, the SLP is attempting to learn the mapping from the sensor data to the actual position.

$$y = \begin{Bmatrix} 1, & if\ \Sigma_i x_i w_i >= threshold, \\ 0, & otherwise. \end{Bmatrix} \tag{2.17}$$

In order to learn this mapping, the final step of training an SLP is to compare the output vector to the ‘ground truth’. When comparing these, there are two possibilities:

- If the SLP has correctly computed an output that is equal to the ‘ground truth’, there’s no need to modify the weights. This is because the network is correct for this input vector.
- The other case deals with a positive or negative error between the output vector and the ‘ground truth’. In this case, the weights are updated by multiplying the learning rate, error and input together and adding this value to the previous weights. The learning rate can be thought of as a measure of how quickly the SLP abandons previous beliefs, or takes to learn new beliefs.

Unfortunately, the SLP suffers from a fatal flaw where regardless of how much the SLP is trained, the resulting model is always a linear classifier. This is shown when an SLP attempts to learn the XOR function. In order to correctly learn the XOR function, there must exist a decision boundary which has values of 1 on one side and 0 on the other. However, the SLP can only produce a linear decision boundary, and the XOR function is non-linear. This is shown in Figure 2.7.

At present, this limitation is preventative of learning how to correct GPS tracking. To combat this limitation, two possible changes can be made to the network.

- One change would be to incorporate a different activation function that maps the output to a space where the outputs are linearly separable. However, this does require some knowledge of the mapping you are trying to learn. In the case of correcting GPS, this knowledge may be hard to obtain and nullifies the key advantage of a neural network over a Kalman Filter. This advantage is its ability to learn without explicit knowledge of the function.
- The other change involves integrating more perceptrons into the network. This allows them to learn non-linear functions, without compromising their generality. These more complex networks are referred to as Multilayer Peceptrons or Artificial Neural Networks.

Table 2.1: XOR Function

X1	X2	Output
0	0	0
0	1	1
1	0	1
1	1	0

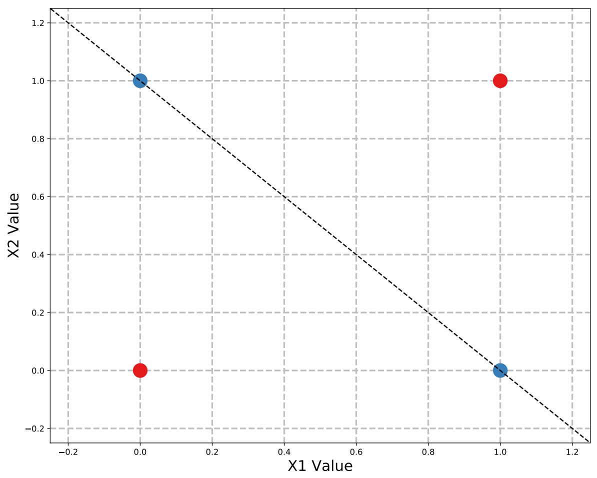


Figure 2.7: The table on the left shows the XOR function being applied to inputs X1 and X2. These values are then plotted in the graph on the right. Red and Blue points represent outputs of 0 and 1 respectively. The dashed line shows a decision boundary for the SLP.

2.8.3 Artificial Neural Networks

Artificial Neural Networks (ANNs) integrate more perceptrons into the network and create a deeper structure. This structure contains a number of hidden layers which reside between the input layer and the output layer. An illustration of a ANN with two hidden layers is shown in Figure 2.8. The number of hidden layers is known as the depth of the network.

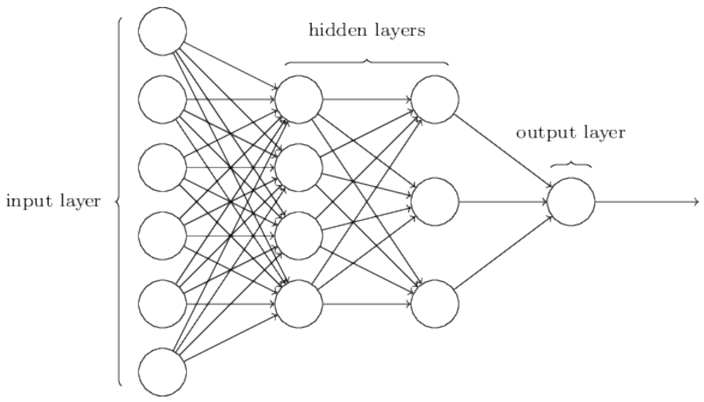


Figure 2.8: Artificial Neural Network with 2 Hidden Layers.

The training algorithm for Artificial Neural Networks is more complex than SLPs and can be split into two steps. The first step is known as ‘forward propagation’ and again involves computing the dot product between the weights and inputs at each node, similar to the SLP. These values are then passed through an activation function like the one detailed in Equation 2.17.

The second step is known as ‘backward propagation’. This propagates the error between the output and the ‘ground truth’, backward through each layer. This determines the amount of error each perceptron is responsible for, and allows the weight vectors to be adjusted accordingly. ANNs define the error using a loss function. The loss function for GPS tracking is defined to be the euclidean distance from the output position to the ‘ground truth’ position.

The network’s aim is to minimize the euclidean distance, and therefore the algorithm used is the ’Gradient Descent’ algorithm. This algorithm is implemented so that each weight is updated in the direction that causes the largest decrease in distance. The gradient is determined by calculating the partial derivative of each weight at a given node.

Although the ANN is a highly powerful network, it is still limited when attempting to learn how to correct GPS tracking. This limitation lies in the sequential nature of the sensor data, as the estimated position is dependent on recent sensor measurements. Recurrent Neural Networks are employed to exploit this sequential dependency.

2.8.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) were developed in response to dealing with sequential data, such as that obtained from IMU and GPS sensors. The 'vanilla' RNN is essentially an ANN with an additional feedback loop. The feedback loop allows the previous input and output information to be considered when calculating the successive output. An illustration of an unrolled RNN feedback loop is shown in 2.9.

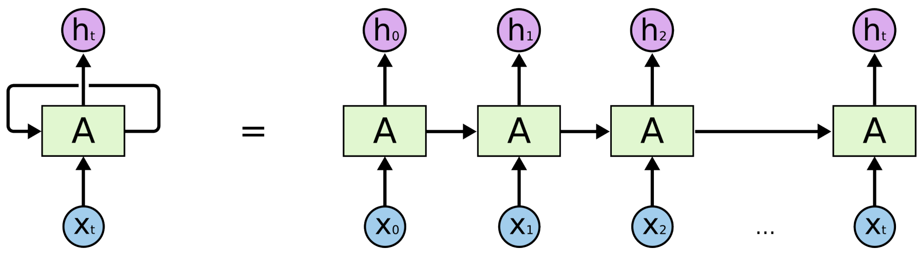


Figure 2.9: A unrolled RNN [24]. Each A is a seperate RNN cell. x_t and h_t are the inputs and outputs to the network respectively. It can be seen that information is passed onto the next RNN cell.

'Vanilla' RNNs are rarely used in real world applications due to a problem that they suffer with in the 'backward propagation' step, this problem is known as 'vanishing gradient problem' and severely affects the networks capability to learn long-term dependencies [25].

The vanishing gradient problem occurs because the gradient of the loss function, becomes exponentially smaller with time. A solution to this problem was proposed in 1997 by Sepp Hochreiter and Jrgen Schmidhuber who developed the Long Short-Term Memory RNN [26] architecture, this included a 'memory cell' and several gates to control the information that was remembered and forgotten.

2.8.5 RNN Architectures - LSTMs and GRUs

Long Short-Term Memory (LSTM) cells differ from 'vanilla' RNN cells as they comprise of 4 different neural network modules as opposed to 1. These interact in a set way, expressed by equations 2.18-2.22. For any given LSTM cell there are 3 different inputs: the current input vector, x_t, the previous output vector, h_{t-1}, and the previous cell state, C_{t-1}, the cell will then compute the current output vector and cell state.

$$f_t = \sigma(W_f \cdot concat(h_{t-1}, x_t) + b_f) \tag{2.18}$$

The first equation describes the LSTM network's capability to forget irrelevant information. It works by using a sigmoid activation layer (3.4.4) that decides which elements of the previous cell state should be remembered or forgotten based on the previous output vector and current input. Features in the sensor data that are irrelevant to calculating an improved GPS position will be forgotten here.

$$u_t = \sigma(W_u \cdot concat(h_{t-1}, x_t) + b_u) \tag{2.19}$$
$$C'_t = tanh(W_c \cdot concat(h_{t-1}, x_t) + b_c) \tag{2.20}$$

The next two equations are responsible for the network learning new information, which is done in two parts. Equation 2.19 details a sigmoid layer which decides whether information in the previous cell state should be updated. Whereas, Equation 2.20 uses a tanh activation layer to calculate new candidate values that will be added depending on the results of the sigmoid layer. These candidate values correspond to features that are found in the sensor data.

$$C_t = f_t * C_{t-1} + i_t * C'_t \tag{2.21}$$

This equation brings the first 3 equations together in order to calculate the new cell state, the $*$ denotes the Hadamard product or element-wise multiplication. The outputs of f_t and i_t are between 0 and 1, where 0 means forget and 1 means remember, these describe how values travel through the system in the cell state or are added to the cell state. The cell state is important as it allows features that improved a GPS estimate to help correct the following GPS estimate.

$$h_t = tanh(C_t) * \sigma(W_o * concat(h_{t-1}, x_t) + b_o) \tag{2.22}$$

The final equation relates to the output state of the LSTM cell, a sigmoid activation layer filters the current cell state which are scaled to between -1 and 1 by an element-wise tanh function. For the purpose of activity tracking, the cell state values won't be scaled between 1 and 0 using the tanh function, this allows the X and Y positions to be re-scaled to meaningful distances.

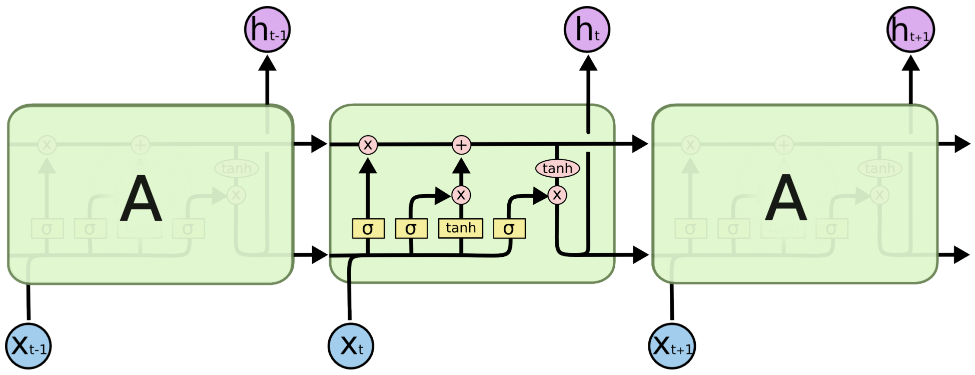


Figure 2.10: Information flow of LSTM Cells [24].

In 2014 the Gated Recurrent Unit (GRU) was proposed by Kyunghyun Cho et al. This was argued to be a much simpler architecture than the LSTM. It is simpler because it combines the forget and input gates into one update gate, and merges the hidden state and the cell state. This reduces the amount of information that must be passed to the next cell. In total the GRU has 3 gates, update, reset and output, these are shown in Equations 2.23-2.25 and Figure 2.11.

$$UpdateGate \quad z_t = \sigma(W_z \cdot concat(h_{t-1}, x_t) + b_z) \tag{2.23}$$
$$ResetGate \quad r_t = \sigma(W_r \cdot concat(h_{t-1}, x_t) + b_r) \tag{2.24}$$
$$OutputState \quad h_t = (1 - z_t) * h_{t-1} + z_t * \sigma(W_h \cdot concat(h_{t-1}, x_t) + b_h) \tag{2.25}$$

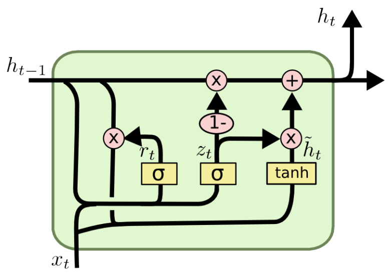


Figure 2.11: An illustration of a GRU Cell [24].

Evaluations of LSTMs and GRUs have shown that they have similar performance for many applications [27]. Therefore, both RNN architectures will be tested on how well they improve GPS tracking.

Chapter 3

Implementation and Methodology

In the previous chapter, the theory behind the Kalman Filter and Recurrent Neural Networks was presented. This chapter will detail their practical implementation along with the subtleties of their design.

The chapter starts by outlining the technologies that were used, including a description of why various software and hardware was appropriate for the implementation. After justifying each technology, the chapter proceeds by discussing the data set used. This discussion looks at how the sensor data was collected and how the 'ground truth' was fitted to this, any pre-processing methods that were necessary are also described here.

After preparing the sensor data, the chapter delves into the main practical implementations. Each implementation is given with respect to the use cases found in section 2.5, and underlines any challenges that were encountered.

3.1 Summary of Technologies

Python was used to implement both the Kalman Filter and the Recurrent Neural Network in this project. It was chosen as a programming language because of its array of packages that well suit it to linear algebra and creating deep learning models. In particular, three key packages were used throughout this project, `NumPy`, `SciPy` and `Pandas` [28, 29, 30]. These packages are commonly used for scientific computing and feature a vast catalogue of higher level mathematical functions. Of course these high level functions have potentially significant overheads in comparison to a custom C implementation, but this can be mitigated by code profiling, and custom implementations can be used where necessary. In addition, native support for objects such as matrices allowed the implementation to be concise and readable.

When improving activity tracking, often the best way to interpret the improvements is to visualize them. It is then trivial to spot scenarios where the algorithm performs well and predicts locations which are close to the 'ground truth' locations. This is clearly also useful for evaluating where algorithms do well and for further optimization. Visualization was also used incrementally throughout the project as a means to debug the code. This project used two python packages to help with visualization, `matplotlib` and `gmaps` [31, 32]. `Matplotlib` is a 2D plotting library that was used to plot graphs that enables quantitative analysis of the data. Whereas, `gmaps` was configured using a Google Maps API key so that routes could be plotted directly onto a map for qualitative analysis.

Data collection utilized two different pieces of software. Sensor data was collected using an Android smartphone, which recorded the data using a third party app, 'Sensorstream IMU+GPS' [33]. This 'off the shelf' application allowed data to be collected easily by storing the recorded data in a 'csv' file ready to be parsed. The other piece of software was the Ordnance Survey online route creation

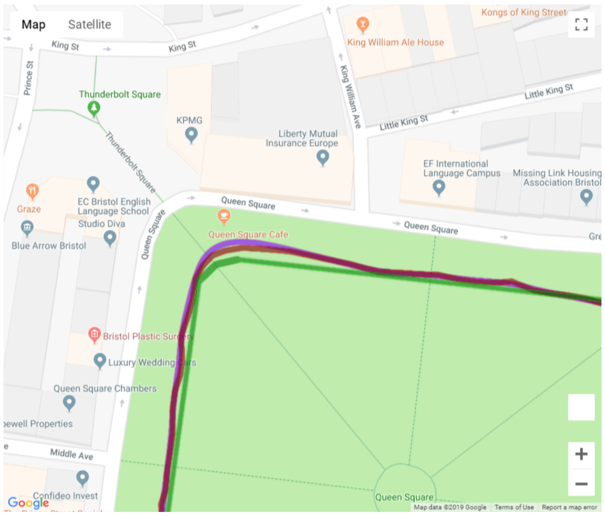


Figure 3.1: Using the **gmaps** package to plot the 'ground truth' (green), Kalman filtered (red) and the original GPS routes (purple).

website, this allowed the creation of a 'ground truth' gpx file from a series of coordinates. Details of why software was chosen to obtain 'ground truth' data can be found in subsection 3.2.2.

Due to the high processing requirements of training deep learning models, underlying hardware was an important concern. Training Recurrent Neural Networks (RNNs) involves lots computation that can greatly benefit from a GPU's many core architecture, along with it's high memory bandwidth [34]. Because of this, an Nvidia P100 GPU was used to train the RNN models. This GPU was provided by Oracle along with a 'BareMetal Compute Instance'. To take advantage of this extra computing power, the machine learning library 'tensorflow' was used. This library supports a range of compute devices, enabling offloading to a GPU. The Keras API was used on top of the tensorflow library as it was easier to get started, it's modular design also allowed testing different RNN layers like the GRU and LSTM.

In late 2018, Nature published an article detailing the reasoning why Jupyter notebook is so popular for data science [35, 36], these reasons aptly explain why Jupyter notebook was chosen as a developing environment for this project. The main advantages the article refers to, are 'remote access' and 'data exploration', both of which are relevant for this project. 'Remote access' relates to how Jupyter can be run remotely on a server and accessed through a web interface, which allowed this project to utilize the 'Oracle Compute Instance's' powerful hardware. Another key reason for choosing Jupyter, is because it encourages 'data exploration'. This is where analysts visually explore data in order to gain an understanding of it's characteristics. In the context of this project, one example of 'data exploration' would be regular graph plotting and experimentation to see which algorithms worked best at correcting certain parts of routes.

3.2 Acquiring an Appropriate Dataset

Even after extensive research, no recognized data set was found that could be used to evaluate either the Kalman Filter, or the Recurrent Neural Network's ability to improve activity tracking. Therefore, this section will first detail how the sensor data was recorded and then the strategy used for acquiring a corresponding 'ground truth'. The process of acquiring the 'ground truth' data was not straightforward, and therefore different ways to obtain this are discussed in subsection 3.2.2.

Methods that were implemented to pre-process the data for use in the Kalman filter and the RNN are covered in subsections 3.2.3-3.2.5. Subsection 3.2.3 is of particular significance as it deals with the difference in sensor frequencies which is critical to this paper.

Finally, this section concludes by recalling the various optimization techniques that were used to reduce the time taken to load and pre-process the data.

3.2.1 Obtaining Sensor Data

The sensor data was collected using a third party app, 'Sensorstream IMU+GPS'. The app had settings that enabled recording of all the smartphone's sensors, including the composite software sensors described in section 2.3. The recording process was simple and just consisted of starting the recording, and then stopping it when the activity was complete.

The app recorded the sensor measurements in a 'CSV' format, this could then be offloaded to a computer to be a parsed by an implemented python function.

3.2.2 Obtaining Ground Truth Data

Obtaining accurate 'ground truth' data for a given activity is critical to this paper's success. This is because the performance evaluation of the Kalman Filter and the Recurrent Neural Network (RNN) depends on the comparison between their estimate and the 'ground truth' positions. The RNN also depends on 'ground truth' data so that a model can be trained. For the 'ground truth' to be used for these purposes, it must be in a specific form. This form is a list of time-stamped locations that directly correspond to the time-stamped sensor measurements.

There was several different proposals on how to obtain this 'ground truth'. A convenient method, was to to use a high precision IMU along with a bike odometer. By using an odometer, the wheel size can be used to get a better estimate of the distance travelled, therefore improving the estimate. Although this would likely produce a better result than GPS alone, the specialist device would likely be subject to the signal errors as detailed in 2.1. This solution would also only applicable to cyclists, hence other solutions were considered.

Other ways to obtain 'ground truth' involved the use of cameras or NFC beacons. Cameras could record video while being strapped to the individual completing an activity. Coordinates could then be mapped to the video frames where the location is known . NFC beacons would work by being placed at various intervals along the activity route, which would then be tapped on route. If each NFC beacon was at a known location, the time at each beacon could be gathered. Unfortunately, both of these involve lots of time to either set up or post-process afterwards, which limits the variety of routes that can be tested. This limit would also dramatically reduce the amount of training data that could be fed to the RNN.

Because of the flaws in these methods, a good compromise between accuracy and time efficiency was found by plotting the 'ground truth' on a map. A key problem with this method is that the time at each position along the route is unknown. Because of this, each GPS reading had the wrong 'ground truth' point associated with it, this is shown in the left hand image of Figure 3.2. To determine the correct time at each 'ground truth' point, Algorithm 3.1 was implemented. After completing this algorithm, the GPS readings can be seen to correspond to the appropriate 'ground truth' points in the right image of 3.2.

Plotting the 'ground truth' points onto a map was done with the use of the online Ordnance Survey map service [37]. This service has a route creation application that allows the user to plot points directly onto satellite images, the resulting route can then be downloaded in a GPX file format. A parser was written for the GPX file using python, this loaded the coordinates into a `NumPy` matrix in similar fashion to the sensor data. The 'ground truth' data was then aligned to the sensor data as per Algorithm 3.1.

```
Data: A list of GPS points. A list of uncorrected ‘Ground Truth’ Points.
Result: A list of corrected ‘Ground Truth Points.
ground_truth= InterpolatePoints(ground_truth)
new_ground_truth = []
for i=0 to gps_data.length do
    new_ground_truth[i] = FindClosestGroundTruthPoint(gps_data[i], ground_truth)
end
return new_ground_truth
```

Algorithm 3.1: Aligning GPS and Ground Truth Data. This algorithm worked by interpolating between each ‘ground truth’ point to generate a large amount of potential points along the route. The GPS points were then iterated through, each GPS point’s corresponding ‘ground truth’ point was defined to be the closest ‘ground truth’ point using the Euclidean distance.

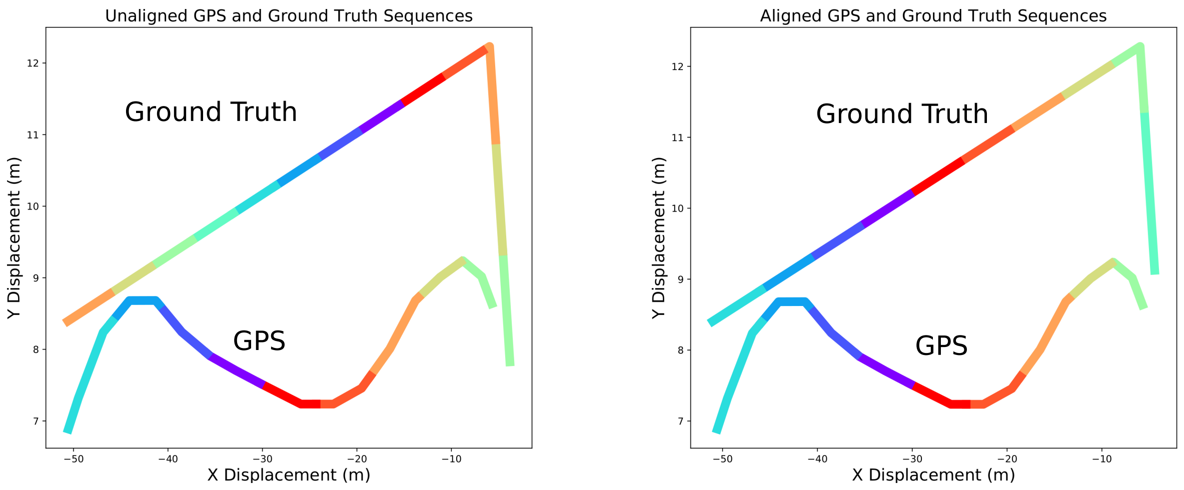


Figure 3.2: ‘Ground truth’ and GPS sequences. Corresponding sequences in the ‘ground truth’ and GPS have the same colour. The left figure shows unaligned sequences, the right shows aligned sequences. Unaligned sequences mean that each GPS sequence has a corresponding ‘ground truth’ sequence that is incorrect. Aligned sequences mean that corresponding GPS and ‘ground truth’ sequences are correctly matched.

3.2.3 Sensor Frequency Disparity

Both the Kalman Filter and Recurrent Neural Network (RNN) implementations require data that is of the same frequency. Unfortunately, there is a great disparity between the frequency of the GPS sensor and the sensors found in the IMU, which are $\approx$ 1Hz and $\approx$ 600Hz respectively. Clearly this disparity is what this project aims to capitalize on, by using high frequency IMU sensors as an asset for improving GPS data. However, the way that this difference is dealt with is a key consideration.

A naive solution to this problem removes IMU data readings that don’t have a corresponding GPS reading. However, this would dramatically decrease the effectiveness of both the Kalman Filter and the RNN, which aim to improve activity tracking by making use of the IMUs near continuous data stream.

Another approach would be to interpolate the GPS data so that it occurs at the same time intervals as the accelerometer data. Logically, this makes sense as its likely that the device will be somewhere between the two ’real’ GPS recorded positions. Unfortunately, this method is not perfect as inaccurate GPS points will skew more of the accelerometer readings. Furthermore, issues also arise if the device is moving fast around corners, this can cause them to be chopped off or in some cases missed altogether.

3.2.4 Converting Latitude and Longitude to Displacement

To ensure that the observation model matrix was simple to compute in the Kalman Filter, it made sense to convert from (latitude, longitude) coordinates to displacement in X and Y from the start position of the activity. It also allows position values in the RNN feature vector to be in the same range across different activities, more detail is given on this with scaling limitations in subsection 3.4.1.

To calculate the distance between two latitude and longitude coordinates, an equation normally used is the Haversine formula. However, as displacement values for X and Y are needed, it is easier to convert the latitude and longitude one at a time. When treated separately, the X and Y displacement from the origin can be calculated using the formulae for the length of an arc.

$$\Delta\theta = longitude_i - start\_longitude$$
$$\frac{\Delta\theta}{360} = \frac{X\_Displacement}{2R\pi}$$
$$X\_Displacement = \frac{2R\pi\Delta\theta}{360}$$

This can be repeated to calculate the displacement along the Y dimension by replacing longitude for latitude. The radius, R, is taken to be the mean radius of the Earth or 6,378,100m.

3.2.5 Earth Relative Acceleration

To ensure that the acceleration was in a suitable form for the Kalman Filter and Recurrent Neural Network, it must be relative to the Earth. Earth relative acceleration is defined below:

- Acceleration in the positive x direction is Eastward acceleration.
- Acceleration in the positive y direction is Northward acceleration.
- Acceleration in the positive z direction is acceleration that is directly upwards to the sky.

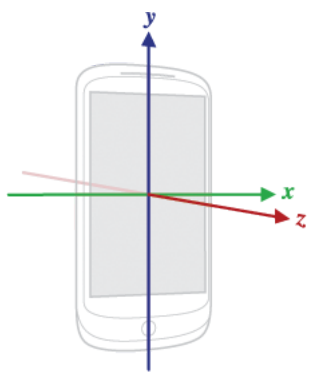


Figure 3.3: X, Y and Z axes relative to a device [38]

However, the Android OS returns acceleration that is relative to the device (Figure 3.3), and therefore it must be converted. This conversion is done by using the rotation vector sensor values collected from the device. This rotation vector is a vector of quaternion values that represent the orientation of the device. The following steps outline how a rotation matrix was obtained and used to calculate acceleration relative to the Earth:

1. Let rotation vector $r = [q_x, q_y, q_z]$ describe the mapping from Earth relative acceleration $\boldsymbol{e}$ to device relative acceleration $\boldsymbol{d}$.

2. Obtain quaternion $\boldsymbol{q} = [q_w, q_x, q_y, q_z]$ where $q_w = \sqrt{1 - q_x^2 - q_y^2 - q_z^2}$.

3. Calculate rotation matrix, $\boldsymbol{R}$, from $\boldsymbol{q}$:

$$\boldsymbol{R} = \begin{bmatrix} 1 - 2(q_y^2 + q_z^2) & 2q_xq_y - 2q_wq_z & 2q_xq_z + 2q_wq_y \\ 2q_xq_y + 2q_wq_z & 1 - 2(q_x^2 + q_z^2) & 2q_yq_z - 2q_wq_x \\ 2q_xq_z - 2q_wq_y & 2q_yq_z + 2q_wq_x & 1 - 2(q_x^2 + q_y^2) \end{bmatrix}$$

3. As per step 1, R maps Earth relative to device relative acceleration, $\boldsymbol{Re} = \boldsymbol{d}$. Therefore, the Earth relative acceleration is equal to $\boldsymbol{e} = \boldsymbol{R^{-1}d}$. This can be further simplified using the knowledge that rotation matrices are orthogonal, $\boldsymbol{e} = \boldsymbol{R^T v}$.

3.2.6 Optimizing Data Parsing and Pre-Processing

After implementing subsections 3.2.1-3.2.5, it was found that some pre-processing and file parsing functions was costly to performance.

To identify functions that were responsible for bottlenecks in the code, a python profiler was used called `line_profiler`. This revealed which functions were computationally 'expensive'. One such 'expensive' function was found to be the `NumPy` concatenate function, which can be used to add a row or column onto a matrix. This function is costly because it's implementation involves creating a whole new matrix, and copying data across. This function was used a lot in the class, and was used when parsing each line of the sensor CSV file. This performance hit was circumvented by adding data to a list, and converting the list to a `NumPy` matrix in one go after the CSV file had been fully parsed.

Even though profiling did produce substantial speedups in the loading of data, notoriously expensive operations like matrix multiplication in subsection 3.2.5 were still affecting performance. As a solution to this problem, a caching system was implemented. This prevented the program from pre-processing the sensor data more than once. It detected whether a cached file existed, and if it did, then data would be directly read from the file. Otherwise, data would be loaded as normal and a file would be written with the pre-processed sensor data. This provided a speedup of 4.5x when loading 10 minutes of sensor data.

3.3 Kalman Filter Implementation

The real challenge of the Kalman filter is understanding the underlying theory that was discussed in section 2.7.1, and applying this theory to improving activity tracking. The resulting hand-crafted sensor fusion algorithm allows the actual python implementation to become almost trivial. Despite this, an overview of the implementation is included in the start of the section, this is included for completeness and allows the reader some intuition about how the code is structured.

However, there is one part of the implementation that is complex and which proves to be a fundamental limitation of the Kalman filter. This part is the initialization of the noise covariances, which govern the uncertainty in the accelerometer and GPS sensors. Discussion of these covariance matrices is held in subsection 3.3.2, this subsection is important to the paper as this Kalman filter implementation provides motivation for using Recurrent Neural Networks.

Throughout this section, choices are made about the implementation with respect to achieving two key aims:

- The resulting Kalman filter must be time efficient, allowing it to be computationally feasible on a smartphone in 'real time' after the activity. A more efficient Kalman filter will also minimize the amount of time spent waiting after the activity, if the correction is done as a 'post processing' measure.

- The other aim is to configure the Kalman filter's parameters such that it maximizes the accuracy of the filter.

3.3.1 Implementation Overview

Once the `kalman_filter()` was passed the accelerometer and GPS sensor data, it allocated `NumPy` matrix objects for each of the matrices found in 2.7.1.

After these were allocated, the `kalman_filter()` proceeded by iterating through the sensor data, timestep by timestep. Within each iteration of this loop, 4 functions were implemented to calculate the corrected position for that timestep:

- The `kalman_predict_step()` function updated the state transition and control input matrices, using them to calculate the predicted position and variances for the timestep. This function used the equations 2.2 and 2.4.
- The `get_kalman_gain()` function was used to compute the 'Kalman Gain' for use in the following two functions. This used equation 2.16.
- The `kalman_update_state()` and `kalman_update_variance()` functions formed the core of the 'update' step of the Kalman filter. These were responsible for using the current GPS reading to update the predicted position. These used equations 2.14 and 2.15 respectively.

At the end of each loop iteration, the current position and it's associated variance was stored in lists. These lists were then returned at the end of the function. Pseudo code of how these methods were used in the `kalman_filter()` function are shown in Listing 3.1.

```
def kalman_filter(sensor_data, acc_std_dev):
    gps = sensor_data.gps
    acceleration = sensor_data.acc
    initialise_matrices()
    states = []
    state_variances = []
    for i=0 to acceleration.length:
        state, variance = kalman_predict_step()
        K               = get_kalman_gain()
        state           = kalman_update_state(K)
        variance        = kalman_update_variance(K)
        states.add(state)
        state_variances.add(variance)
    return states, state_variances
```

Listing 3.1: Kalman Filter Implementation

3.3.2 Initializing Noise Covariance Matrices

At a high level, the Kalman filter calculates an estimate of the current position using a probability distribution for both the accelerometer and GPS sensors. The identification of each distribution's covariance matrix is therefore critical to the Kalman filter's performance. Figure 3.4 shows an activity's sensor data and the noise that is present.

The observation noise matrix, R, describes the GPS sensor's covariance and is relatively straightforward to estimate. This is because the US Government states that the global GPS accuracy has an average of 7.8m for civilian devices in 95% of cases [16]. This translates to a standard deviation of 3.9m and thus a variance of 15.2m. Therefore:

$$R = 15.2 * I$$

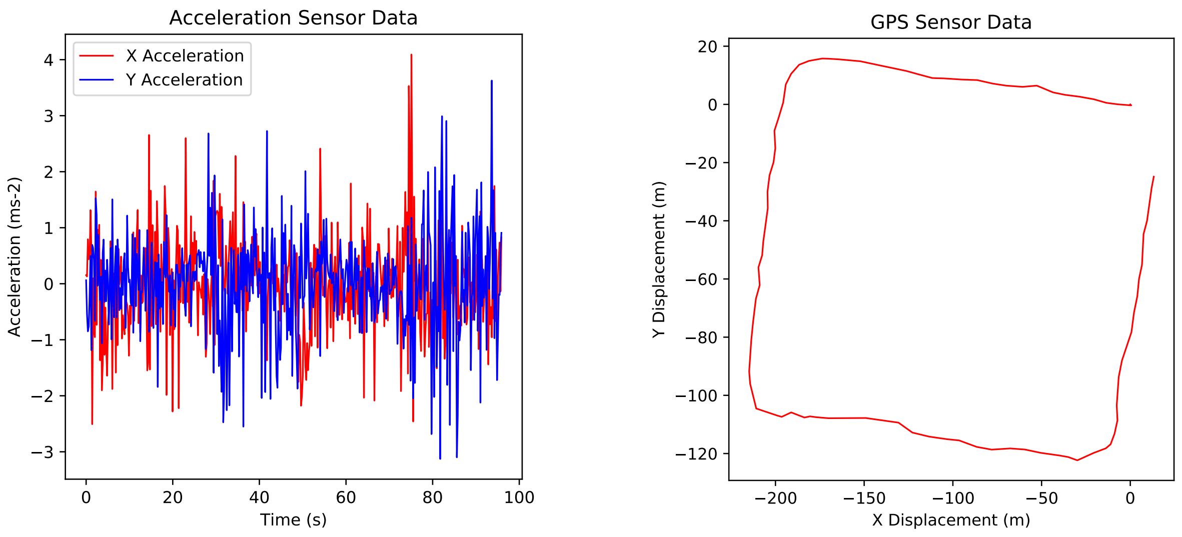


Figure 3.4: The left image shows a data sample from the accelerometer sensor, the right images shows the corresponding GPS route. Noise is clearly present in both sensor samples.

It is important to note that this value of R encodes the assumption that the error in GPS latitude is independent of the error in longitude.

Unfortunately, estimating the covariance of the position and velocity, Q, is a much harder task. Both the noise in the position and velocity estimates depend on the noise of the accelerometer sensor, and assuming that the acceleration in the X direction is independent of acceleration in the Y direction, the variance of the displacement and velocity noise can be calculated. This is done using the standard deviation of the accelerometer, $\sigma_{X''}$. Newton's laws of motion can then be used to derive both the standard deviation of the velocity, $\sigma_{X'}$, and the position, σ_X:

$$\sigma_{X'} = \sigma_{X''} * dt$$

$$\sigma_X = \frac{1}{2}\sigma_{X''} * dt^2$$

Hence, Q is:

$$Q = \begin{bmatrix} \sigma_X^2 & 0 & \sigma_X\sigma_{X'} & 0 \\ 0 & \sigma_X^2 & 0 & \sigma_X\sigma_{X'} \\ 0 & 0 & \sigma_{X'}^2 & 0 \\ 0 & 0 & 0 & \sigma_{X'}^2 \end{bmatrix}$$

As this definition of Q only depends on the unknown standard deviation of the accelerometer, multiple values were tested in order to find one that maximizes the performance of the Kalman Filter. Figure 3.5 shows this experiment given different standard deviations values for one recorded activity data. This shows that the standard deviation that gives the best Kalman filter performance is 0.012 meters. It can also be seen that as the standard deviation increases, the Kalman filter's trust in the accelerometer decreases, causing the Kalman filter's performance to tend towards the uncorrected GPS performance.

3.4 Recurrent Neural Network Implementation

This section details how the RNN implementation was created, starting from how the sensor data was fed to the network. It looks at how this data was processed to maximize the success of the network and the reasons for doing this.

Next, the section identifies several hyper-parameters that configure the RNN's ability to learn a pattern. The strategy used for choosing these is included, along with the different trade-offs that were

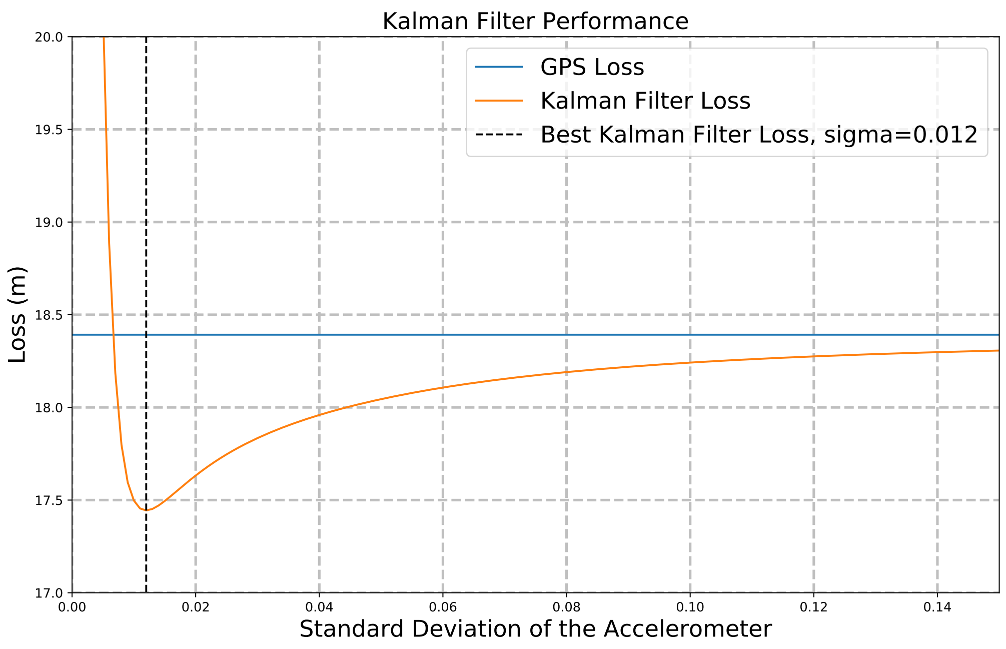


Figure 3.5: A graph showing the effect of different accelerometer standard deviations on the Kalman filter's performance, in comparison to the stand-alone GPS accuracy.

considered for each hyper-parameter. Tests are also performed which trial the impact of different hyper-parameters, and how these affect the training process.

This experimental approach is mirrored when choosing the network's architecture and deciding whether GRUs or LSTMs are more effective for improving GPS tracking. When testing these different RNN architectures, detail is also given on the number of hidden layers and the activation functions that each architecture used. The final recurrent neural network configuration is given in Table 3.1, and a diagram of it's architecture is included in Figure 3.6.

Table 3.1: Final Recurrent Neural Network Configuration

Parameter Name	**Value**
Sequence Length	300
Sequence Offset	30
Batch Size	128
Layers	Two Gated Recurrent Unit layers with 128 states each. Final layer was a fully connected 'dense' layer.
Activation Function	Linear

3.4.1 Feeding a Recurrent Neural Network

The Keras API was used to create the RNN implementation, this API allows developers to compile a custom model or network. Once the network has been compiled, the training process is initiated by calling the `fit()` function. Initially, this method was sufficient for training the network on the GPU, however, as more data was collected, the GPU ran into `object allocation` errors. This error is caused because the entirety of the GPU's memory (16GB) has been exhausted. The solution to this problem is to use the Keras `fit_generator()` function, which works by being passed a generator.

The generator acts as a handle for the network to request a tuple containing a batch of input and a batch of output data. Each batch must be a three dimensional array, which is a list of sequences. Each element in each sequence is a feature vector. The generator is defined using two values, the sequence length and its offset, the generator then iterates through the training data and computes sequences of

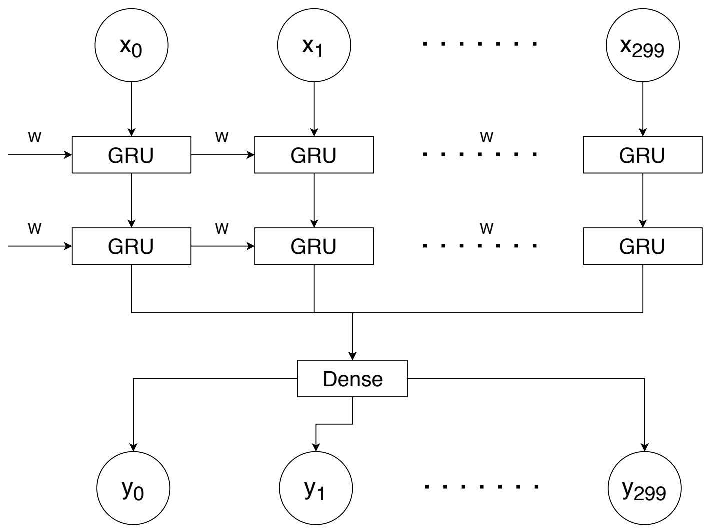
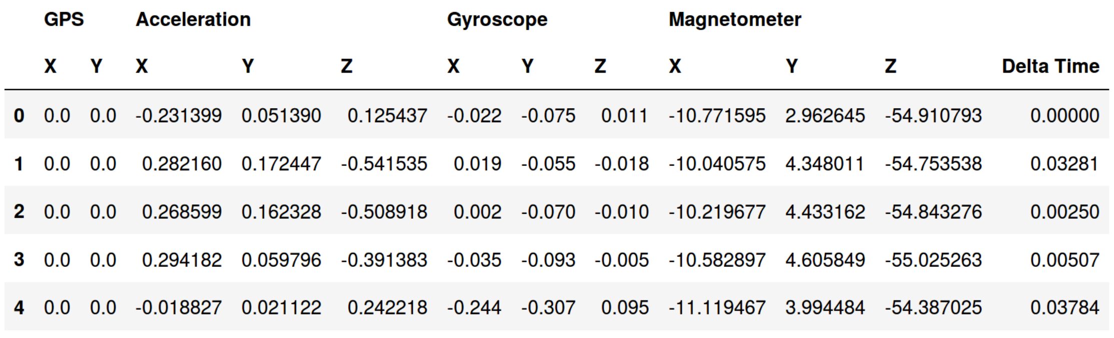


Figure 3.6: A diagram showing the chosen recurrent neural network architecture, with two GRU layers followed by a dense layer. x_t and y_t denote the feature vector and output vector at time t.

the given length, at intervals specified by the offset. After generating these sequences, they are shuffled and a batch of them are yielded for training on the GPU. The sequences are shuffled so that they appear in a different order for each training iteration or epoch, this increases the networks generality and helps prevent overfitting. It is important to realize that each feature vector in a sequence will be in chronological order, but that the sequences themselves may not necessarily be consecutive.

The feature vector comprises of the different sensor readings at a given timestep, specifically these are GPS, accelerometer, gyroscope and magnetometer readings. Another feature that was included was 'delta time', this is the amount time that has elapsed since these sensor readings. This feature was included to give the RNN some intuition behind the correlation between time and position. Figure 3.7 shows 4 example feature vectors of a recorded activity.



	GPS		Acceleration			Gyroscope			Magnetometer			
	X	Y	X	Y	Z	X	Y	Z	X	Y	Z	Delta Time
0	0.0	0.0	-0.231399	0.051390	0.125437	-0.022	-0.075	0.011	-10.771595	2.962645	-54.910793	0.00000
1	0.0	0.0	0.282160	0.172447	-0.541535	0.019	-0.055	-0.018	-10.040575	4.348011	-54.753538	0.03281
2	0.0	0.0	0.268599	0.162328	-0.508918	0.002	-0.070	-0.010	-10.219677	4.433162	-54.843276	0.00250
3	0.0	0.0	0.294182	0.059796	-0.391383	-0.035	-0.093	-0.005	-10.582897	4.605849	-55.025263	0.00507
4	0.0	0.0	-0.018827	0.021122	0.242218	-0.244	-0.307	0.095	-11.119467	3.994484	-54.387025	0.03784

Figure 3.7: `Pandas` DataFrame showing the first 4 feature vectors of a training activity.

Unfortunately, data must be between certain bounds to work with the activation functions in both LSTMs and GRUs. Because of this, the feature vector must be scaled to between 0 and 1. This presents a problem for distance features such as GPS which have no upper or lower bound. This is a problem because the network could be trained on data where the user only travels 1km away, but the model could be used to improve an activity where the user travels 2.5km away. This then prevents the rescaling of the output vector, as max values of 1 could represent distances of 1 to 2.5 kilometers. To combat this limitation, the project assumes that the maximum displacement in either the X or Y direction is less than 3km, and it scales accordingly across each activity.

3.4.2 Hyperparameters

There were several hyperparameters which needed to be set when creating the network, possibly one of the most significant was the sequence length. This length dictated the amount of feature vectors seen at one time when training the network. To decide which sequence length caused the best performance, the network was trained using the same configuration for a selection of different sequence lengths. Using the frequency of the linear acceleration sensor (600Hz) the sequence lengths were chosen to be 300, 600, 1200, and 3000, which represent 0.5, 1, 2, and 5 seconds worth of sensor readings respectively. For all these tests, the offset of different sequences was chosen to be a tenth of the sequence length. 'Loss vs Epoch' graphs show the training process for each sequence length in Figure 3.8.

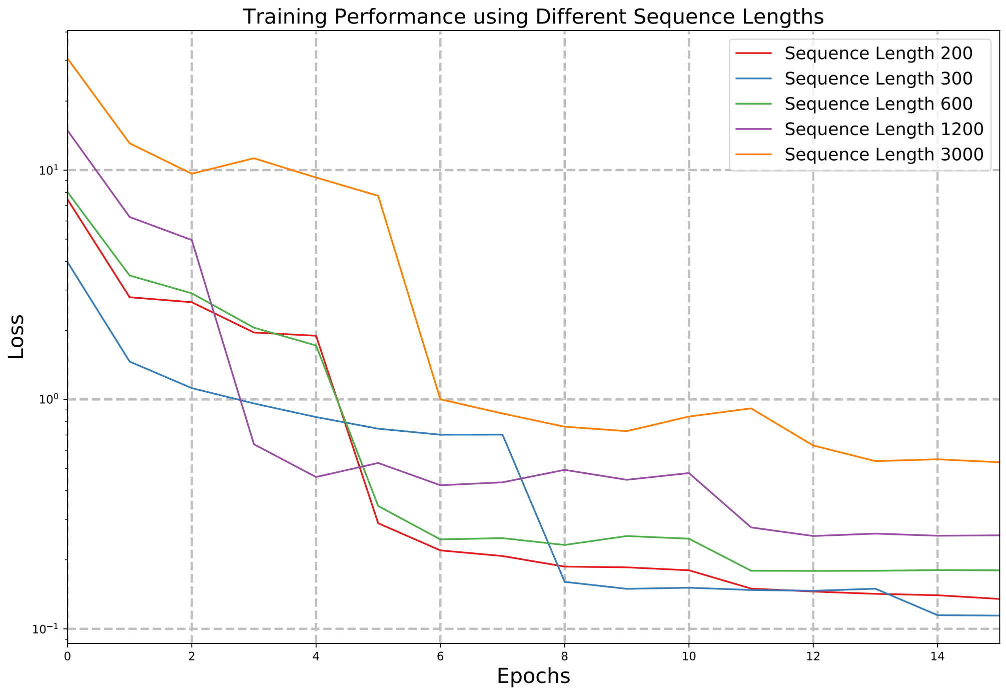


Figure 3.8: Graph showing the training performance using different sequence lengths.

Clearly, the sequence length affects the minimum loss that the network converges at, with there being a correlation between a higher sequence length and a higher loss value.To capitalize on this correlation, another sequence length of 200 was tested to check that performance couldn't be improved further. From this experiment, it can be seen that the sequence length that achieves the lowest loss value is the shortest 300 length, representing 0.5 seconds of sensor readings. Due to this, the sequence length of 300 was used henceforth.

After the sequence length had been determined, a related parameter is the number of sequences per batch. Various different sizes were tried throughout the tuning process of the network, and by varying this, there were two observations:

- Increasing the batch size allowed the network to train faster, due to more sequences being stored in the GPU's memory at one time.
- Decreasing the batch size increased the performance of the network, allowing it to converge at a lower loss value.

To attempt a compromise between these observations, different batch sizes were used for different stages of the project. A large batch size was used during the configuration stage where the network was tested with many different architectures (subsection 3.4.4). Then after refining the network, a smaller batch size was used to maximize the network's performance. For consistency, an epoch was set to be the number of unique sequences divided by the batch size, therefore an epoch in both stages was a full pass of the data and contained the same number of sequences.

3.4.3 Training a Recurrent Neural Network

A key concern when training a neural network is overfitting. This is where the network performs very well on the training data but doesn't perform well on other test data. To combat this, unseen or 'validation' data is tested between epochs to ensure that the network's performance is improving on both the training and validation data. Validation data was chosen to be a random 30% of the testing sequences.

To adjust the weights of the network, and to check the performance of the validation data after each epoch, a loss function is used. In this implementation, the loss function was defined to be the euclidean distance from the estimated position to the actual 'ground truth' position.

Keras provides several callback functions that can be used to automate the training process, and prevent issues like overfitting. The two callback functions used in this implementation were:

- `Early Stopping`, this was used to stop training when the network's performance on the validation data decreased after an epoch. The callback has a 'patience' parameter that was set to 1 which allowed the network to decrease in performance after 1 epoch, but if the performance didn't get better after the next epoch, the training phase would be forced to end.
- `Decrease Learning Rate on Plateau`, this was used to decrease the LR if the networks performance on the validation data set got worse. The learning rate controls how quickly the network converges to a solution.

3.4.4 Recurrent Neural Network Architecture

The RNN implementation consisted of several different layers which allowed it to learn the mapping from the sensor data to the 'ground truth' data. The general structure included several specific RNN layers, and then the final layer was a fully-connected 'dense layer'. The reason for including several layers is because it enables the network to learn a non-linear function. All neurons in the 'dense' layer are connected to all the neurons in the final RNN layer, the 'dense' layer can then calculate the dot product of the input and weight vectors. An activation function is applied to this, and an output is produced.

Activation functions define the output of a node, a simple example being a binary classifier which returns 0 or 1. Most neural networks use a more complicated activation function than this, and produce a range of output values. Sigmoid and tanh are both commonly used activation functions, and have lower and upper bounds that prevent values from 'exploding' in size. However, as the output in this case is a position vector, there is no clear upper bound and therefore a linear activation function is more suitable for this implementation. Although this is necessary, the linear function does have a constant derivative, and thus can't be optimized during the 'backward propagation' step. Figures 3.9 and 3.10 contain illustrations of the linear and sigmoid activation functions respectively, these visually depict the difference in each function's gradient.

Using the Keras API, it is relatively straightforward to try a range of different architectures. To maximize the learning capability of the network and reduce training times, an experiment was devised to test the effect of using more RNN layers and the number of nodes per layer. The network was trialled with 1, 2, 3, 5 or 10 RNN layers, and for each of these, a different numbers of states were tested. A graph of these results is shown in Figure 3.11.

This graph shows the impact of how different layer configurations can significantly affect the number of epochs taken to converge, and the loss that this occurs at. It can be seen that the network converges in around 5-8 epochs for the majority of tests, however the networks with 1 or 10 layers were at a

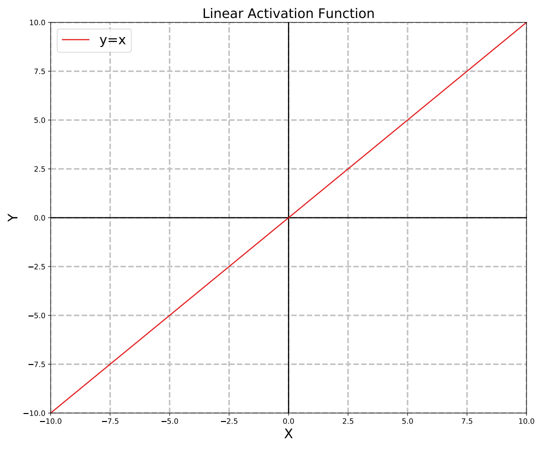


Figure 3.9: Linear Activation Function

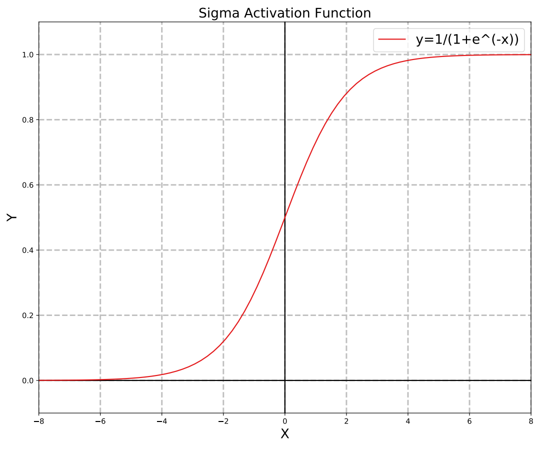


Figure 3.10: Sigmoid Activation Function

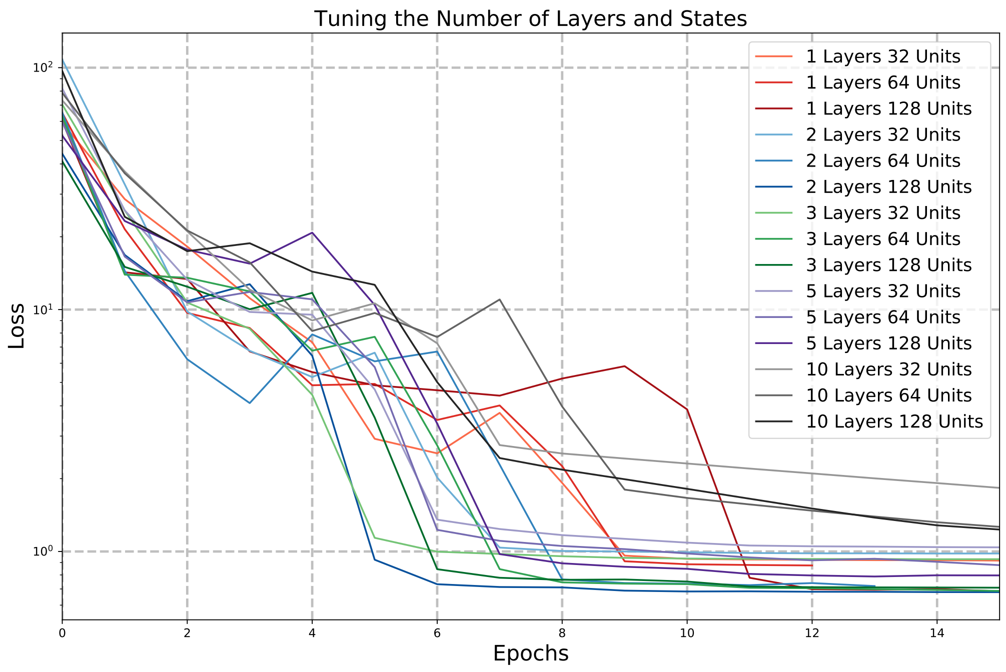


Figure 3.11: Graph showing experimentation with different numbers of RNN layers and states.

much higher loss values after 15 epochs. Given more time, the 10 layer networks would likely converge at a comparable loss value to the smaller networks, unfortunately the training time would have been substantial due to the large network depth. The 1 layer networks had lower values of loss than the 10 layer networks after 15 epochs, but they were still higher than the networks with 2-5 layers. Unlike the 10 layer networks, this is more likely caused by the networks lack of depth, which prevented it from understanding the sensor data's complexity.

The networks that converge at the lowest loss values tend to have either 2 or 3 RNN layers. Although there is very little difference in loss between these, it does affect the time to complete one epoch. Adding a third layer caused epochs to take approximately 1.5x longer than using 2 layers, and for this training data, the 3 layer networks converged at similar values to the network with 2 layers and 128 states. For this reason, the 2 layer network with 128 states was chosen for the evaluation stage.

3.4.5 LSTM vs GRU Performance

As mentioned in Chung's evaluation paper [27], Long Short Term Memory (LSTM) networks and Gated Recurrent Units networks achieve comparable performance, with GRUs typically converging in less time. Because of this and the modular nature of the Keras API, both architectures were tested to identify which performed best.

Figure 3.12 shows the training process of each of these different RNN architectures. It can be seen that both achieve very similar levels of loss, and start to converge after the first 5 epochs. Using loss alone, it is hard to distinguish which architecture was best for improving GPS. However, one distinguishing factor was the time taken taken to complete each epoch, where it was found that the GRU was 1.27x faster. This was a worthwhile speedup over the LSTM, and thus it was chosen as the best RNN architecture for improving GPS.

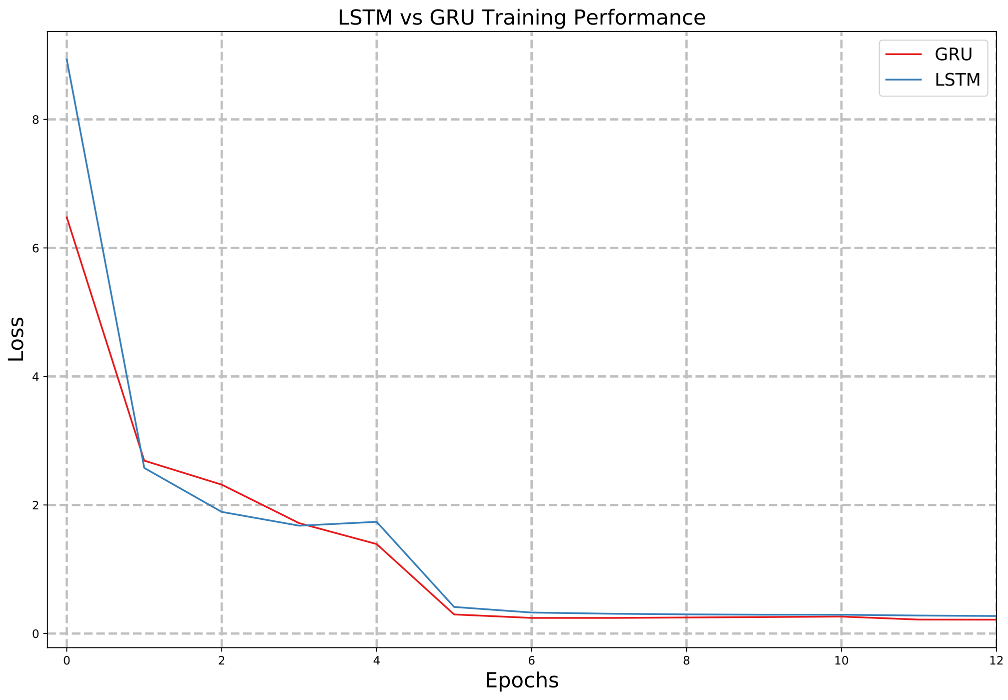


Figure 3.12: Graph show training performance of Long Short Term Memory and Gated Recurrent Unit networks.

Chapter 4

Critical Evaluation and Results

The final chapter of this paper aims to evaluate the performance of both the Kalman filter and the Recurrent Neural Network (RNN) implementations. These implementations are being used to improve the accuracy of GPS activity recordings, and thus each implementation will be examined with respect to the use cases found in section 2.5.

The chapter begins by detailing how the accuracy of each implementation can be quantitatively evaluated, so that it can compared to the accuracy of stand-alone GPS. This provides a measure of success for experiments contatined in this chapter.

This quantitative measure is employed to describe the results given in Table 4.1. Section 4.2 aims to give some context for these results and outlines the practices that were used to obtain them, followed by some conclusions that can be drawn from them. To further aid discussion, route estimates from the Kalman filter and the Recurrent Neural Network have been superimposed onto maps, this provides some qualitative evidence of their performance.

The next section recognizes that the activity type may impact the performance of each implementation. Therefore section 4.3 aims to compare walking, cycling and running activities as a means to identify the strengths and weaknesses of each implementation.

After analyzing the results, this paper aims to highlight the challenges that were faced so that further research can be focused into these areas. This will allow readers to understand how GPS can be improved in the future, and where innovation is needed. The issues faced by this project can be largely classified as 'noise' or 'ground truth' related, with these being discussed in Section 4.4 and 4.5 respectively.

The 'ground truth' section is of the utmost importance as it details why the 'ground truth' presents such an issue. It is not the identification of the route that presents this issue, but understanding where each sensor reading is along the route. Obtaining this corresponding 'ground truth' data-set for activity tracking is far from trivial, almost becoming a topic for research in it's own right. In the future, this may require a more laborious approach such as filming the route and tagging video frames to known positions.

The penultimate section is a summary of the entire paper, detailing the accomplishments of the project and all the contributions that have been made. These achievements will be measured against the initial aims of the project. Finally, this paper ends by identifying the current stage of the project, and uses the challenges that were outlined in this chapter to give future directions for research.

4.1 Evaluating Accuracy

GPS routes and those that are estimated by both the Kalman filter and the RNN are essentially a sequence of coordinates. These coordinates can be described as a displacement value in both X and Y dimensions. The 'ground truth' is also in this form once it has been interpolated to the linear accelerometer's frequency (600Hz).

Therefore, the distance between each route and the 'ground truth' represents the accuracy of the route estimate. Assuming that both the test route and the 'ground truth' are correctly aligned, this accuracy can be measured by calculating the euclidean distance. This paper defines the accuracy of a route to be the mean euclidean distance between each coordinate and its corresponding 'ground truth' coordinate.

This assumption that the estimate route and the 'ground truth' are aligned is discussed in more detail in Section 4.5.

4.2 Results

To test the effectiveness of the Kalman filter and the Recurrent Neural Network (RNN) implementations, a test data-set was collected. This data-set consisted of approximately 30 minutes of 3 different activity types. The activities in the data-set were walking, running and cycling, and each contained over 1 million sensor readings all with a corresponding ground truth. The RNN implementation also made use of an additional 5 hours of training data. Both the Kalman filter and the RNN implementations were tested in 'real time' and 'post-processing' configurations that were defined in Section 2.5. Results of each implementation's accuracy are shown in Table 4.1 and show the percentage improvement compared to stand-alone GPS.

Table 4.1: Accuracy of Implementations for Walking Activities

Implementation	**Test Accuracy**	**Improvement over GPS**
GPS	15.35	N/A
Kalman Filter 'Real Time'	16.14	-5.12%
Kalman Filter 'Post Processing'	16.01	-4.30%
Recurrent Neural Network 'Real Time'	15.99	-4.16%
Recurrent Neural Network 'Post Processing'	15.91	-3.64%

Disappointingly, these results clearly show a lack of improvement over stand-alone GPS. However, this does not necessarily mean that the Kalman Filter or RNN configurations described by this project are unfit for improving GPS. The reason for this, is each implementation's complete dependence on reliable 'ground truth' data. Acquiring 600 'ground truth' position coordinates per second of sensor readings is no mean feat, and is vital to check the accuracy of each implementation compared to GPS. Therefore, the GPS may well be closer to the 'ground truth', but the 'ground truth' may not be representative of the actual route taken.

Despite this issue, these results still offer insight into the strengths and limitations of each implementation, so that they can be improved in the future. This discussion is provided for each implementation in the following two subsections:

4.2.1 Results - Kalman Filter

With respect to the use-cases that were outlined in Section 2.5, it can be seen that the 'post-processing' Kalman filter was more accurate than the 'real-time' Kalman filter. The difference between these is that the 'post-processing' Kalman filter also completes a reverse pass of the activity data, which is then averaged with the forward pass.

Over the test data-set, the Kalman filter implementations were less accurate than the RNN implementations and the stand-alone GPS. This suggests that there are two possible problems with this evaluation of the Kalman filter, assuming that the motion model derived from Newton's laws of motion is correct:

- The first possible problem is that the 'ground truth' is inaccurate, this could feasibly result in the GPS being deemed more accurate than the Kalman filter estimate, despite the Kalman filter potentially being closer to the 'actual route' taken. The limitations of the 'ground truth' are explored comprehensively in Section 4.5.
- The second problem has already been mentioned by this paper and relates to the Kalman filter's dependence on the sensor covariance estimates. These describe the Kalman filter's trust in the each of the sensor's readings. Section 4.4 provides evidence of the noise negatively affecting the performance of the Kalman filter, and offers an alternative method for describing the sensor variance for future implementations.

4.2.2 Results - Recurrent Neural Network

The 'post-processing' RNN featured 2 bidirectional GRU layers, this was more accurate than the 'real time' RNN with just 2 forward GRU layers. This RNN implementation was also more accurate than both of the Kalman filter implementations.

In spite of this being the most accurate implementation, it was still 3.64% less accurate than just using the GPS sensor. Unlike the Kalman filter, it is much harder to pinpoint the source of the RNN's inaccuracy, but it is likely that it fails for similar reasons. A wide range of RNN hyper-parameters were experimented with throughout this paper's 'Implementation and Methodology' chapter, and thus the RNN configuration appears to be correct. Equally, the RNN re-scales the output to a 'sensible' estimate of the route, and therefore this is unlikely to be the problem. The output of the RNN can be seen visually in Figure 4.1.

As mentioned in the Kalman filter results section, an inaccurate 'ground truth' could explain why the GPS is deemed more accurate than either of the implementations. Additionally, an inaccurate 'ground truth' route would have more serious consequences for the RNN, which is dependent on accurate 'ground truth' during training. If each sensor reading does not have an appropriate 'ground truth', the RNN may fail to spot the correlation between the IMU readings and the next position. This maybe what leads the RNN to attempt to minimize the loss function by simply translating the GPS coordinates northward. Evidence of this occurring can also be seen in Figure 4.1.

Figure 4.2 is a set of graphs showing the training and validation loss against the number of epochs, each graph has a different validation data-set. Interestingly, each of these graphs have very similar performance on the validation and training data-set. In practice, this means that the validation data is too similar to the training data. This paper speculates that this may be the case because the network learns to forget the noisy IMU sensors in favor of the GPS sensor. This strategy for minimizing the loss function would therefore work well for any validation data-set, as the GPS sensor will always be relatively close to the 'ground truth'. Further evidence to support this theory could be given by performing cross-validation, to assess how the network model will generalize to an independent data-set.

4.3 Performance Comparison across Different Activities

Although both the Kalman filter and the Recurrent Neural Network provided no improvement in accuracy on the test data, this test data was compiled of a mixture of walking, running and cycling activity data. Table 4.2 identifies the accuracy of the 'post processing' Kalman filter and the Recurrent Neural Network on different activity types.

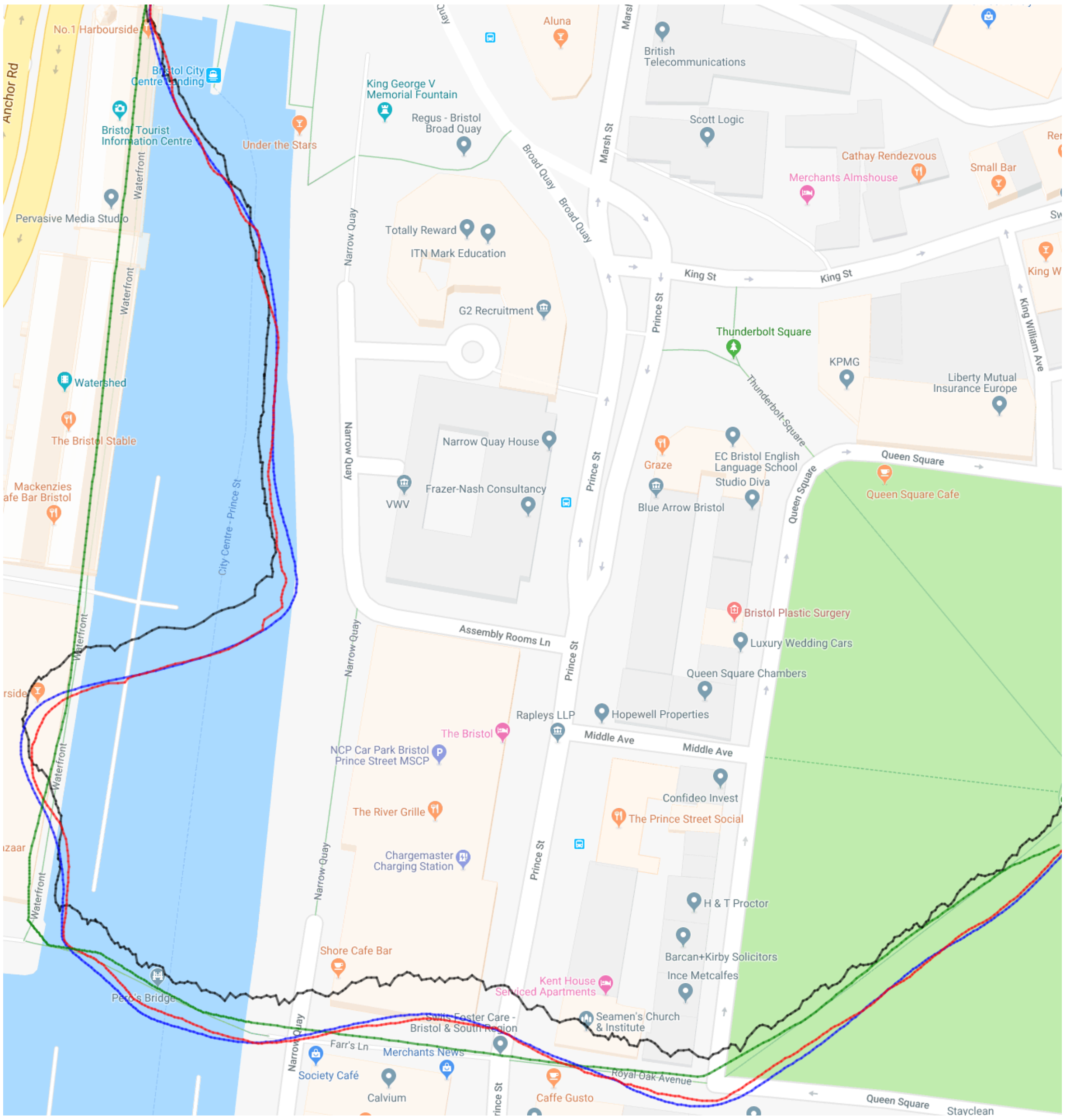


Figure 4.1: Map showing the GPS route (red), Kalman Filtered route (blue), Recurrent Neural Network route (black) and the actual route taken (green). This map shows a portion of a walking activity that was recorded where the GPS is clearly subject to errors cited in subsection 2.1.1. On the left hand side, the GPS route can be seen to divert off the actual route taken and travel across the body of water. Neither the Kalman filter or the RNN corrects this error. The Kalman filter produces a very smooth estimate, whereas the RNN appears to have retained some of the 'kinks' found in the GPS. Furthermore, the RNN appears to have translated the route northward' as a means to minimize the loss function.

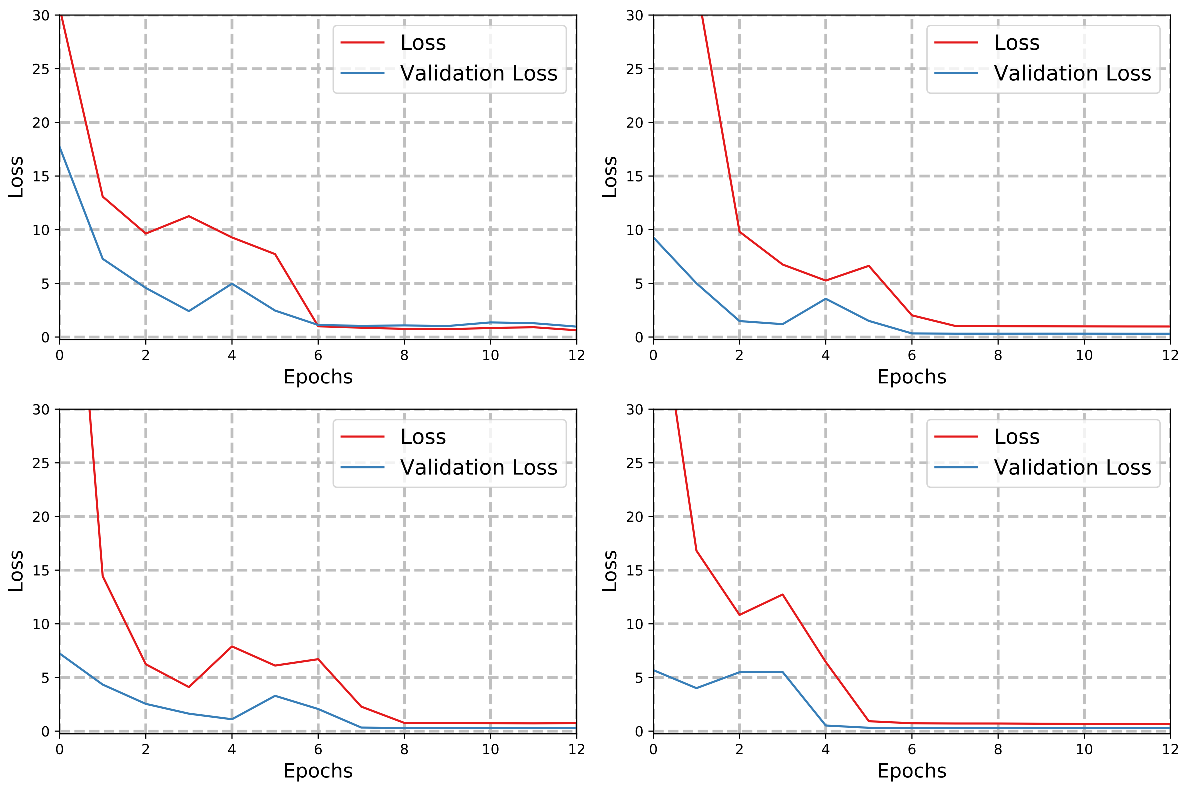


Figure 4.2: Four ‘Loss vs Epoch’ graphs, each with a different combination of training and validation data. For all of these plots, the validation and training data converge at very similar values of loss. Furthermore, only one shows the validation data performing worse than the training data.

Table 4.2: Improvement over GPS activity tracking for each ‘post processing’ implementation. The improvement is identified separately for each activity type that was tested.

Post-Processing - Implementations	**Walking - Improvement**	**Cycling - Improvement**	**Running - Improvement**
Kalman Filter	-0.57%	-2.12%	-10.20%
Recurrent Neural Network	-1.07%	-1.38%	-8.47%

Clearly, the type of activity data that is being tested has a direct impact on the accuracy of the estimated route. Initially, conclusions may be drawn that indicate that implementations work best on ‘slow’ activities like walking, and the faster activities like running and cycling result in less accurate routes. This maybe partly true as running and cycling activities will cut parts of corners off, especially when moving at high velocities.

Although this does not explain why cycling achieves far greater accuracy than running. This paper speculates that the noise in the recording causes this result. For the walking activity, the smartphone recording the activity was in a trouser pocket, which is a relatively restricted position. The smartphone was also in a firm position for cycling activities, where it was strapped into a cycle mount on the handlebars of the bike. Interestingly, running activities were the least accurate, and allowed the phone to be loose in a rucksack where it had a greater freedom of movement. In addition, cycling along roads is a far smoother motion, whereas, running causes an ‘up and down’ bouncing motion. This conclusion that noise has a dramatic effect on the success of each implementation is further explored in Section 4.4.

4.4 Mitigating the Impact of Noisy Data

Sections 4.2 and 4.3 have shown the consequences of how noise in the IMU sensors has prevented any implementation from improving the accuracy of GPS. This section looks at why noise has created problems for both the Kalman filter and the RNN, and discusses measures that could be implemented to mitigate the risk of this occurring again in the future.

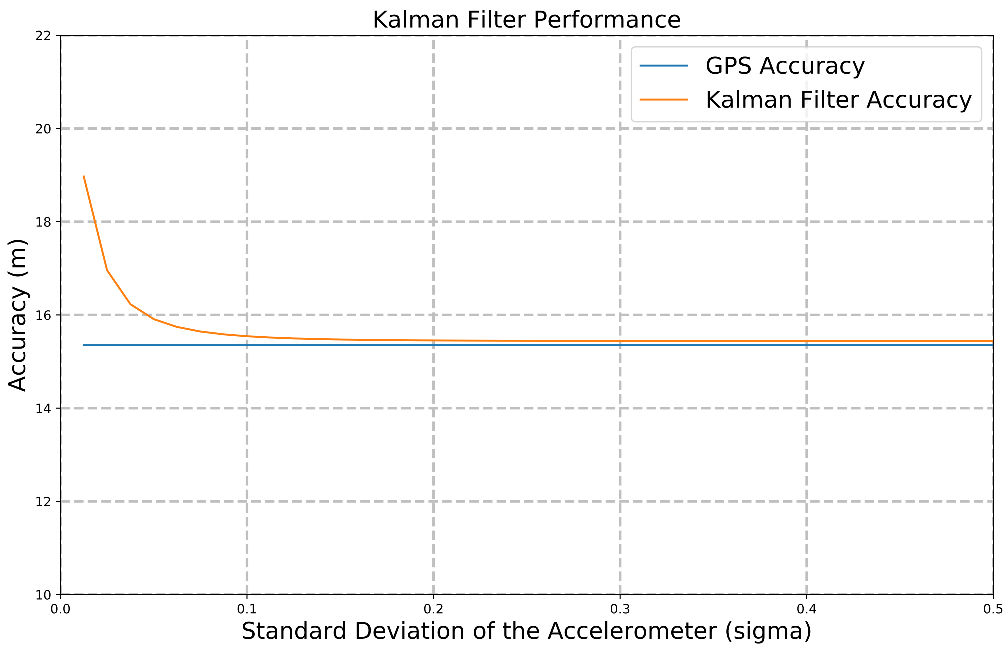


Figure 4.3: A graph showing the effect of different accelerometer standard deviations on the Kalman filter's performance, in comparison to the stand-alone GPS accuracy. As the standard deviation value for the accelerometer increases, the Kalman filter accuracy tends towards the accuracy of stand-alone GPS. This suggests that the accelerometer is too unreliable to provide an improvement

As mentioned in subsection 3.3.2, the Kalman filter is reliant on accurate covariance estimates. Moreover, the subsection details how the covariance matrices were set and the assumptions that were made. A key assumption that was made for both the GPS and the accelerometer covariance matrices, was that X and Y dimensions were independent, is not necessarily true. This assumption allowed the GPS covariance matrix to be calculated from the US governments expected accuracy, and the accelerometer covariance to be based on its standard deviation. Using this information, the optimal standard deviation could be calculated based on what improved accuracy the most, as seen in Figure 3.5. This experiment was repeated after the collection of all the test data, to identify the standard deviation that maximized the accuracy of the Kalman filter. These results in Figure 4.3 show that the greater the standard deviation of the accelerometer is, the more accurate the Kalman filter becomes across all of the test data. This implies that the inclusion of accelerometer readings hinders the performance of the Kalman filter, and produces results worse than GPS.

In future work, a calibration step could be included for the sensors before performing the Kalman filter. This would allow a more accurate covariance estimate for both the observation and process noise. However, this would not be a general solution for all phones and thus all users would have to complete this calibration. Although this may prove effective, it is possible that the IMU found in a standard smartphone is not currently accurate enough for improving GPS tracking.

Currently, the Kalman filter uses one covariance matrix to describe the accuracy of all IMU sensor readings, even though some readings will be more accurate than others. The current Kalman filter could be upgraded to make use of a new Android OS feature that gives accuracy information about individual readings, ranking them from unreliable to high accuracy. This information could be used to update the covariance matrix of the Kalman filter for individual readings, helping it understand where it can trust accelerometer readings. In addition, unreliable readings could be removed altogether.

4.5 Ground Truth

'Ground truth' has been cited throughout this project as challenging to obtain and critical to the project's success. It is not the identification of the route that presents this challenge, but understanding where each sensor reading is along the route. This project relies on accurate 'ground truth for training the RNN and for the evaluation of each implementation.

Subsection 3.2.2 details a relatively simple method of obtaining 'ground truth', by plotting coordinates onto a map. The closest 'ground truth' point is then determined for each sensor reading in order to use the 'ground truth' for evaluation. Initially, it was assumed that this method would be sufficient for both training and evaluation, with it being correct in the majority of places. However, the results in this chapter suggest that the 'ground truth' might be to blame for the lack of improvement over GPS.

The current method for calculating a sensor reading's corresponding ground truth point has a number of failure cases:

- If the 'ground truth' features a U-shaped bend, sometimes this corner can be cut off by the low sampling rate of the GPS. This becomes more likely when travelling at high speed, perhaps on a bike. This error leads to the GPS recording a shallower corner, which then leads to sensor coordinates being mapped to the sides of the U bend. An illustration of this is provided in Figure 4.4.

- This closest 'ground truth' policy can lead to more serious problems if the route features cycles or goes back on itself. This can cause the sensor readings to be matched up with 'ground truth' points that are not actually encountered until much later on in the activity.

In an attempt provide an improved algorithm for matching 'ground truth' coordinates to individual sensor readings, a number of alternate options were discussed. None of these were thought to improve the current system, and all proposals feature limitations of their own. However, a selection are included below as a means to provide evidence for the depth of discussion around the matter, and prevent future innovators from repeating research.

- One approach was to identify the closest coordinate pairs between the 'ground truth' data and the sensor readings. The sensor readings in between those identified as being close to the 'ground truth, would then be set to interpolated values along the 'ground truth' route. An advantage of this method is that it prevents the current failure case shown in Figure 4.4. However, it is difficult to define what is meant by "closest coordinate pairs", especially for a large number of routes. This approach also assumes that the the user is travelling at a linear speed along the route, which of course, is rarely the case.

- Another approach attempted to exploit the assumed normal distribution of GPS errors, based on differing satellite constellations. This approach assumes that a route could be completed several times, and defines the 'ground truth' to be the mean GPS route of these. Unfortunately, this proposal does suffer from a multitude of of issues, namely the route would have to be performed at exactly the same velocity each time to ensure the 'ground truth' time-steps corresponded to exactly the same position each activity. Furthermore, subsection 2.1.1 details how errors are more likely around obstacles, and these obstacles will likely present the same errors for each activity recording.

These proposals show that obtaining a corresponding 'ground truth' data-set for activity tracking is far from trivial, almost becoming a topic for research in it's own right. In the future, this may require a more laborious approach such as filming the route and tagging video frames to known positions. To avoid this painstaking approach, further innovation is required for matching up the corresponding sequences of 'ground truth' points and sensor readings.

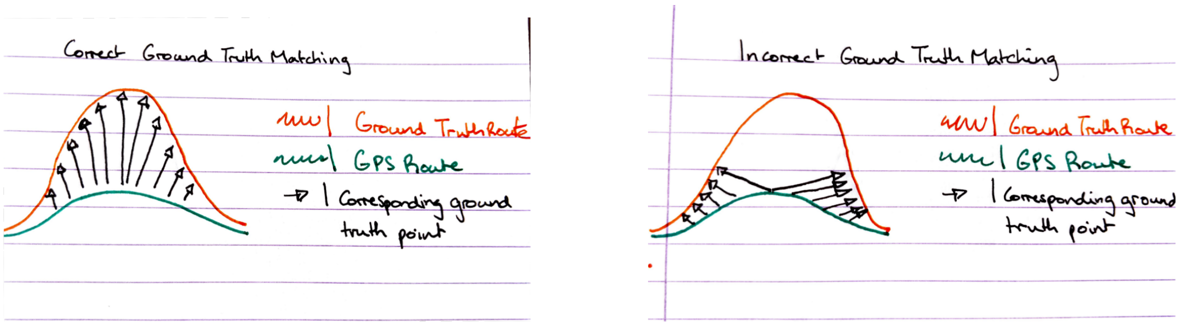

Correct Ground Truth Matching
Ground Truth Route
GPS Route
Corresponding ground truth point
Incorrect Ground Truth Matching
Ground Truth Route
GPS Route
Corresponding ground truth point

Figure 4.4: Ground truth failure case. In the right hand diagram, the ‘sensor recording’ coordinates are mapped to incorrect ‘ground truth’ points which are closer than those at the top of the U-shaped bend. The left hand diagram shows the correct mapping.

4.6 Conclusion

This paper has thoroughly explored how inaccurate GPS routes can be improved using additional sensors found in a smartphone. Furthermore, both classic and modern techniques have been comprehensively researched and applied in an attempt to solve this problem. At the outset, it was forecast that the smartphone’s current capabilities in combination with the appropriate algorithm would provide a substantial improvement over stand-alone GPS. However, this project has since grown in scope dramatically and encountered new interesting problems, opening a variety of avenues for research. Examples of these problems include obtaining ‘ground truth’ and adapting algorithms to deal with the noise found in a smartphone IMU. Preliminary results have indicated that these problems are preventing any substantial improvement over stand-alone GPS, and therefore using IMU sensors to supplement GPS remains inconclusive.

The first chapter starts by explaining why GPS can produce inaccurate routes, and how it’s low frequency can be supported by the high frequency sensors found in an IMU. Different algorithms that could improve GPS are surveyed in this chapter, and the rationale for choosing a Kalman filter and Recurrent Neural Network is given here. Specific terminology is then used to describe how these work in the applied context of improving GPS activity tracking.

The practical nature of this project is detailed in Chapter 2. This explains how the data was collected and all the pre-processing methods that were implemented to use the data in the Kalman filter and the Recurrent Neural Network. The Kalman filter implementation is then detailed, with particular focus on the initialization of the covariance matrices. The RNN implementation is described in a similar way, with various experiments included that provide reasoning for choosing different hyper-parameters.

The final chapter provides the results of this paper, and analyzes them in an effort to improve future results. It cites two key areas for improvement relating to the noise in the data, and the ‘ground truth’ that is fitted to this. All future directions for research are included in Section 4.7.

4.6.1 Contributions

- A survey of various different methods that either correct or supplement GPS tracking.
- A data-set was collected specifically for this project. This data-set was comprised of a mixture of walking, cycling, and running data, and is the first data set of its kind.
- GPS and IMU sensors have widely differing frequencies, this paper discusses how these can be combined for use in a Kalman Filter and in a Recurrent Neural Network.
- Quantitative and Qualitative analysis is given about the success of both the Kalman filter and Recurrent Neural Network when improving activity tracking.
- An algorithmic pipeline that can produce real testable improvements, given a more accurate data-set with a reliable ‘ground truth’.

- There is exhaustive discussion about the benefits and caveats of different ways to obtain 'ground truth' data.
- Proposals for future work have been given, with insight about where innovation is needed to obtain further success.

4.7 Future Work

This paper's target of improving GPS tracking using a smartphone's modest sensor capabilities was indeed ambitious. The results in Section 4.2 highlight this, and perhaps future research should first attempt to achieve a working proof of concept. Using the results in Section 4.3, it can be seen that cycling activities performed well even though they included the user travelling at higher velocities than the other activities. Therefore, this paper speculates that the implementations provided in this project should first be applied to a route recorded by a car. This would allow larger and more accurate IMU sensors to be used, and the system would hopefully encounter reduced noise due to 'smooth' roads and the car's suspension. In addition to these advantages, the ground truth would be able to use roads where exact coordinates can be obtained from Georgraphic Information Systems (GIS). If this proved to be successful, the project could then be adapted afterwards for fitness activities where more noise is encountered.

Regardless of whether sensor data is recorded using a car or during a sports activity, the method of obtaining 'ground truth' needs to be addressed. As there is currently is no available algorithm for solving the issues presented in 4.5, a more rigorous collection method is required. This paper suggests the use NFC beacons or cameras, which will help identify where along the route different sensor readings correspond to. This can be done by tagging individual video frames or NFC beacons to known position coordinates, the time of arrival at each coordinate can then be calculated.

Another key avenue of further research includes modifying the Kalman filter to deal with sensor readings of various degrees of accuracy. Section 4.4 details how new Android OS features can give information on the reliability of readings. These readings could then be used to help the Kalman filter identify how reliable each sensor reading is, and potentially remove erroneous samples. This removal of samples would not only improve computation time but would also reduce the number of 'ground truth' points required.

Currently, the Kalman filter and the Recurrent Neural Network rely on the interpolation of GPS data to force it to be of the same frequency as the IMU data. Thus each algorithm has no knowledge of when the input contains an 'actual' GPS coordinate, or one that has been estimated between two 'actual' coordinates. To combat this, the RNN could be modified with extra input features that indicate whether the GPS coordinate is an 'actual' coordinate, indicated using a mask value of 0 or 1. Alternatively, there has been research into 'phased' LSTMs that use an additional time gate, allowing them to deal with sensors of different frequencies [39].

Bibliography

[1] Strava. *Strava: 2018 in Stats*. 2017. URL: https://2018.strava.com/en-us.

[2] TOBIAS ODENMAN NIKLAS MAGNUSSON. “Improving absolute position estimates of an automotive vehicle using GPS in sensor fusion”. In: (2012). URL: http://publications.lib.chalmers.se/records/fulltext/159412.pdf.

[3] Shahram Rezaei. “Kalman Filter-Based Integration of DGPS and Vehicle Sensors for Localization”. In: (2007). URL: https://www.researchgate.net/publication/3332952_Kalman_Filter-Based_Integration_of_DGPS_and_Vehicle_Sensors_for_Localization.

[4] Alireza Shaghaghian. “Improving GPS/INS Integration Using FIKF Filtered Innovation Kalman Filter”. In: (2018). URL: https://www.researchgate.net/publication/328634878_Improving_GPSINS_Integration_Using_FIKF-Filtered_Innovation_Kalman_Filter.

[5] Geoffrey Hinton Alex Graves Abdel-rahman Mohamed. “Speech recognition with deep recurrent neural networks”. In: (). URL: https://ieeexplore.ieee.org/abstract/document/6638947/.

[6] Jfffdfffdurgen Schmidhuber Alex Graves. “Offline Handwriting Recognition withMultidimensional Recurrent Neural Networks”. In: (). URL: http://people.idsia.ch/~juergen/nips2009.pdf.

[7] Faustino Gomez et al. Hermann Mayer. “A System for Robotic Heart Surgery that Learns to Tie Knots Using Recurrent Neural Networks”. In: (). URL: https://ieeexplore.ieee.org/document/4059310.

[8] Sport England. *Active Lives Survey*. May 2017-2018. URL: https://activelives.sportengland.org/Result?queryId=4900.

[9] Wikipedia. “Global Positioning System”. In: (). URL: https://en.wikipedia.org/wiki/Global_Positioning_System.

[10] Alfred Kleusberg and Richard B Langley. “Limitations of GPS”. In: (Mar. 1990). URL: http://gauss.gge.unb.ca/gpsworld/EarlyInnovationColumns/Innov.1990.03-04.pdf.

[11] Alex Souza Bastos Hisashi Hasegaw. “Behavior of GPS Signal Interruption Probabilityunder Tree Canopies in Different Forest Conditions”. In: (2017). URL: https://www.tandfonline.com/doi/pdf/10.5721/EuJRS20134636.

[12] GIS Georgraphy. *GPS Accuracy*. 2018. URL: https://gisgeography.com/gps-accuracy-hdop-pdop-gdop-multipath/.

[13] HoneyWell. *HoneyWell - Specifications*. 2016. URL: https://aerospace.honeywell.com/en/~/media/aerospace/files/datasheet/laservi-productdescription.pdf.

[14] Ben Popper. *Google announces over 2 billion monthly active users*. 2017. URL: https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users.

[15] Google. *Android Sensors*. URL: https://developer.android.com/guide/topics/sensors/sensors_motion.

[16] US Dept. of Defense. *GPS Accuracy*. URL: https://www.gps.gov/systems/gps/performance/accuracy/.

[17] Wikipedia. *Map Matching*. URL: https://en.wikipedia.org/wiki/Map_matching.

[18] John Krumm Paul Newson. “Hidden Markov Map Matching Through Noise and Sparseness”. In: (). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/map-matching-ACM-GIS-camera-ready.pdf.

[19] Strava. *Strava Labs - Slide*. URL: https://labs.strava.com/slide/.

[20] Rudolf E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: (). URL: https://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf.

[21] Hojjat Salehinejad et al. "Recent Advances in Recurrent Neural Networks". In: (). URL: https://arxiv.org/pdf/1801.01078.pdf.

[22] MOHINDER S. GREWAL and ANGUS P. ANDREWS. "Applications of Kalman Filtering in Aerospace 1960 to the Present". In: (). URL: https://ieeexplore.ieee.org/document/5466132.

[23] Frank Rosenblatt. "The perceptron; a theory of statistical separability in cognitive systems". In: (1957).

[24] Christopher Olah. "Understanding LSTM Networks". In: (2015). URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[25] Paolo Frasconi Yoshua Bengio Patrice Y. Simard. "Learning long-term dependencies with gradient descent is difficult". In: (). URL: https://www.semanticscholar.org/paper/Learning-long-term-dependencies-with-gradient-is-Bengio-Simard/ba18247cd3ce9f711eecc7296f1c3561dbfb6cc

[26] Sepp Hochreiter and Jurgen Schmidhuber. "Long Short-Term Memory". In: (). URL: https://www.bioinf.jku.at/publications/older/2604.pdf.

[27] et al. Junyoung Chung. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: (). URL: https://arxiv.org/abs/1412.3555.

[28] *NumPy Library*. URL: http://www.numpy.org/.

[29] *SciPy Library*. URL: http://www.scipy.org/.

[30] *Pandas Library*. URL: https://pandas.pydata.org/.

[31] *Matplotlib Library*. URL: https://matplotlib.org/.

[32] *Gmaps Plugin*. URL: https://jupyter-gmaps.readthedocs.io/en/latest/index.html.

[33] *Sensorstream IMU+GPS App*. URL: https://play.google.com/store/apps/details?id=de.lorenz_fenster.sensorstreamgps&hl=en_GB.

[34] Erjin Zhou et al. Boxun Li. "Large scale recurrent neural network on GPU". In: (). URL: https://ieeexplore.ieee.org/abstract/document/6889433.

[35] Jeffrey M. Perkel. "Why Jupyter is data scientists computational notebook of choice". In: (). URL: https://www.nature.com/articles/d41586-018-07196-1.

[36] *Jupyter Notebooks*. URL: https://jupyter.org/.

[37] *Ordnance Survey Maps*. URL: https://osmaps.ordnancesurvey.co.uk/.

[38] *Sensor Types*. URL: https://source.android.com/devices/sensors/sensor-types.

[39] Michael Pfeiffer Daniel Neil and Shih-Chii Liu. "Phased LSTM". In: (). URL: https://arxiv.org/pdf/1610.09513.pdf.