

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Recurrent Neural Networks Applied for Human Movement in Subjects with Duchenne Muscular Dystrophy

Author:
Daniel Heaton

Supervisor:
Aldo Faisal

Submitted in partial fulfillment of the requirements for the MSc degree in MSc
Computing (Machine Learning) of Imperial College London

September 2019

Abstract

Your abstract.

Contents

| | | |
|-----------|---|-----------|
| I | Background and Basis of Research | 1 |
| 1 | Project Overview and Background | 2 |
| 1.1 | Motivation for Project and Purpose of Work | 2 |
| 1.2 | Aims and Objectives | 3 |
| 2 | Basis of Research Project | 6 |
| 2.1 | Duchenne Muscular Dystrophy | 6 |
| 2.2 | The North Star Ambulatory Assessment | 7 |
| 2.3 | The KineDMD Research Initiative | 8 |
| 3 | Overview of Recurrent Neural Networks | 10 |
| 3.1 | Outline | 10 |
| 3.2 | Motivation, Architecture, and Training | 10 |
| 3.3 | The Long Short-Term Memory RNN Architecture | 13 |
| 3.4 | Implementing an RNN using TensorFlow | 15 |
| II | System Preparation: Choices and Setup | 18 |
| 4 | System Choices: Language, IDE, and Libraries | 19 |
| 4.1 | Python | 19 |
| 4.2 | Integrated Development Environment | 20 |
| 4.3 | TensorFlow | 22 |
| 5 | System Setup: Software and Data Preparation | 24 |
| 5.1 | Overview | 24 |
| 5.2 | Necessary System Resources | 24 |
| 5.3 | Data Sets Setup | 26 |
| 5.4 | Setup of Python, Pip, Necessary Packages, and PyCharm | 27 |
| 5.5 | Installation of PyCharm (IDE) | 29 |
| 5.6 | Configuring of TensorFlow to Use the GPU | 30 |

| | |
|--|-----------|
| III System Overview and Explanation | 32 |
| 6 Project and Local Directories: Overview and Explanations | 33 |
| 6.1 Background | 33 |
| 6.2 The Project Directory | 33 |
| 6.3 The Local Directory | 36 |
| 6.3.1 The Local Directory: 'rnn.py' Outputs | 37 |
| 6.3.2 The Local Directory: Data Sets | 38 |
| 7 Reference Documents Explanation | 42 |
| 7.1 Background | 42 |
| 7.2 Google Annotations Sheet ('nsaa_17subtasks_matfiles.csv') | 42 |
| 7.3 NSAA Scores Reference Document ('nsaa_6mw_info.xlsx') | 44 |
| 7.4 Results for Experiment Sets ('RNN Results.xlsx') | 45 |
| 7.5 Results for Model Predictions Sets ('model_predictions.csv') | 47 |
| 8 Background | 49 |
| 9 Contribution | 50 |
| 10 Experimental Results | 51 |
| 11 Conclusion | 52 |

Part I

Background and Basis of Research

Chapter 1

Project Overview and Background

1.1 Motivation for Project and Purpose of Work

Modern machine learning algorithms and methodologies has seen great success in regards to being applied to biomedical data in order to provide insights that assist specialists and help diagnose potential conditions that may afflict subjects. One example of this is DeepMind's recent progress with AI-assisted eye scans to detect over 50 different types of diseases potentially in a subject's eye as accurately as world-leading expert doctors [1]. Deep learning techniques have also been utilized by HeartFlow to help build 3D models of subjects' hearts and assess the impacts of blockages on blood flow to the heart [2]. Based on these breakthroughs, among others, in recent years, it is very probable that AI-assistance will become the new norm in various clinics and hospitals around the world within the next decade [3]. Taking note of the prominent applicability of artificial intelligence to the analysis of human movement in particular, Imperial College London have undertaken a research initiative in collaboration with Great Ormond Street Hospital to investigate the applicability of various AI techniques to the analysis of subjects with Duchenne muscular dystrophy [4], a form of muscular dystrophy that predominantly affects young males under the age of 12 and severely impacts their movement ability to varying degrees. The hope is that great strides in treatments can be achieved by utilizing artificial intelligence to inform and make decisions on a subject-by-subject basis and potentially draw insights about the condition.

With regards to this project specifically that is undertaken as part of this research initiative, we wish to investigate the applicability of recurrent neural networks (RNNs) to the features of human movement data provided by body suit measurements captured from subjects with Duchenne muscular dystrophy (DMD). This will hopefully provide not only evidence on the applicability of these models to this sort of data, but also should provide important insights into the data itself and of the subjects providing it. These insights from the project will hopefully have a direct positive impact in the ability to assess subjects via the North Star Ambulatory Assessment (NSAA) and possibly provide insights into other features of the condition. Generally, in this project we are trying to gain insights about the body suit data that

is captured as ‘.mat’ files (MATLAB data files) through the different measurements that are captured by the 17 sensors of the suit (joint angles, position, accelerometer values, etc.) of NSAA or 6-minute walks assessments of the subjects by means of sequence modelling using RNNs. This use of sequence modelling is necessary to model the dependencies through time of measurements, and we are more likely to have a robust model if measurement values are treated as NON-independent with respect to time.

The overall goal of the project is therefore to provide a deliverable that includes a complete system that works with varying forms of suit data, learns from it, and provides insights about it, while hopefully being able to be adapted to new subjects, new NSAA assessments of existing subjects, and help inform specialists of the severity of their condition. A significant hope, therefore, is to provide a software solution that positively and directly impacts the lives of those with DMD through hopefully making their assessments easier and more accurate.

1.2 Aims and Objectives

With the overall aim of the project outlined and with the motivation for undertaking the work provided above, we now turn our attention to covering some of the main aims of the project. These include:

- Building a reasonably good model (with surrounding supporting scripts that are outlined later on) that, when presented with new, unseen ‘.mat’ files of body suit data, can give a reasonably good approximation of individual NSAA activity scores and an overall NSAA score (i.e. the accumulation of all individual activity scores). A prominent limitation currently, however, is the overall lack of data files: we have no more than 50 complete ‘.mat’ files in total for each of the 6-minute walk and NSAA assessments. This is primarily due to the fact that the data collection is currently an ongoing process and a large repository of previously-collected suit data from other subjects with DMD does not appear to exist that’s publicly available. Hence, an implicit requirement of the project is to be able to make the most out of the data we have available, such as using it to train a model to predict different things, use different measurements contained within the ‘.mat’ files, look at applying statistical analysis on the raw data, and so on.
- Being able to use trained model(s) to gain insights into the most influential activities and measurements from the ‘.mat’ files on overall NSAA score and to identify activities that correlate highly with overall assessment. In doing so, it could possibly enable the reduction of 17 activities needed for accurate overall NSAA assessment to far fewer if only a few are needed to correctly assess the subject. The conclusions that we could possibly draw from the project, therefore, hopefully have the potential to aid specialists in the practical undertaking

1.2. AIMS AND OBJECTIVES

of the assessments through minimizing the amount of testing the subjects have to do.

- Investigating the impact of training models on different types of source data directly. For example, we'd like to see whether or not it's possible to train models on natural movement behaviour data sets to the same standard as if we were using NSAA data sets when training towards overall and individual NSAA scores. If this were to be the case, then there exists a real possibility of not requiring the NSAA assessments to be completed by subjects at all, and instead simply requiring the subject to undertake natural movement instead, which may be significantly easier and/or more practical for subjects.
- Building models that are trained only on one 'version' of assessments of subjects and attempt to generalise to subsequent versions. Here, by 'version' we mean an assessment of a subject that takes place at a certain time, with subsequent versions being the same assessment but taken 6-months later on. For example, a subject's initial NSAA assessment would be stored as the subject name 'D4', and when that subject returns 6-months later to undertake their subsequent NSAA assessment the resultant data file would be stored as 'D4V2'. The hope is therefore to train models on non-'V2' files and generalise to newly presented 'V2' files. This should provide an advisory tool for any specialists wishing to assess how a subject's conditioned has progressed during the time between assessments.
- Looking into how possible it is to build models that generalise well to new subjects and the system settings needed to achieve this; by this, we mean models that are able to assess subjects that they have never come across before during training (which differs to the previous bullet point, which looks at new data from existing subjects). Therefore, if this were to be the case then we would be able to extend the applicability of this system to not only new assessments of the existing subjects but brand new subjects to the overall research initiative. There are numerous techniques that we would have to look into if the models have a problem generalizing, and so a large amount of the model predictions sets will focus on this aim.
- Package all the scripts and models necessary for a specialist or any other researcher wishing to use any of the built tools in a way that is easy to use and gives intuitive output. This requires us to construct the system in a way where it is possible to be used by others outside of the development environment in order to be practically applicable to achieve the aims outlined above.

With our overall project aims outlined above, it's also useful to cover some of the objectives that we intend to achieve in order to complete these aims, many of which will be investigated within their own experiment set or model predictions set. These include:

- Comparing models built from different measurements (e.g. joint angles, acceleration values, computed statistical values, etc.) on their performance of evaluating unseen sequences of data to an accurate D/HC classification and overall/single-act regression of NSAA scores.
- Evaluating the ideal values for different sequence setups for the data going into the model with respect to the performance for various output types. This includes finding the ideal sequence length, sequence overlap, and discard proportion of frames within the sequences.
- Investigating the ideal number of features needed for the raw measurements and computed statistical values to train a model. This will involve a trade-off, with more features providing more of the inherent variance within the data and fewer features making it easier for the model to learn from.
- Looking into how well models performed when evaluated on files from a different source directory than their own. For example, we'd like to investigate the potential to models built from NSAA files and assess subject files from the natural movement behaviour data set. If this is possible, then with the finished models we could use these to assess a subject based solely on their natural movement, which might be much more practical than requiring the subject to undertake the NSAA assessment.
- Investigating how well models perform when they are familiar with the subject as opposed to when the model has never seen the subject before in training (even if it was trained on different data from the same subject than was used for assessing the subject).
- Assessing the applicability of generalisation techniques that includes downsampling the data, adding Gaussian noise to the data set, concatenation of features for multiple measurement types, and the leaving-out of anomalies within the subjects.

Chapter 2

Basis of Research Project

2.1 Duchenne Muscular Dystrophy

The data that we shall be working with for this project is from subjects who have varying severities of Duchenne muscular dystrophy (DMD). DMD is a genetic disorder that is characterized by progressive muscle degeneration and weakness and is caused by the absence of dystrophin, a protein that helps keep muscle cells intact [5]. This leads to increasing levels of disability and is a progressive condition, meaning that it gets worse over time. It's classified as a rare disease, with around 2,500 patients in the UK and an estimated 300,000 sufferers worldwide [6]. There are currently no known cures for any form of muscular dystrophy (MD), though there are treatments available to help manage the conditions [7].

DMD is one of the more severe forms of MD and generally affects boys in their early childhoods, and those with the condition generally only live into their 20s or 30s. The muscle weakness starts in early childhood and symptoms are usually first noticed between the ages of 2 and 5. The weakness mainly affects the muscles near the hips and shoulders, so among the first signs of the disorder are when the child has difficulty getting up off the floor, walking, or running. The weakness progresses to eventually affect all muscles used for moving and also those involved in breathing and the heart muscle. Many are confined to a wheelchair by 12 years of age, with those in their late teens generally losing the ability to move their arms and experiencing progressive problems with breathing.

With the aim to increase the life span and movement options of sufferers of DMD, a range of treatments are available. These range from steroids to increase muscle strength, physiotherapy to assist with mobility, and surgery to correct postural deformities. And thanks to advances in cardiac and respiratory care, life expectancy for sufferers is increasing and many are able to survive into their 30s and 40s with careers and families, with there even being cases of men with the condition living into their 50s. Additionally, there is ongoing research looking into ways to repair the genetic mutations and damaged muscles associated with various forms of MD.

2.2 The North Star Ambulatory Assessment

The North Star Ambulatory Assessment (NSAA) is a 17-item rating scale that is used to measure functional motor abilities in subjects with DMD and is generally used to monitor the progression of the disease and the effects of treatments. The tests are to be completed without the use of any thoracic braces or any equipment assistance that may help the subject to complete the activities. To carry out the assessments, the assessor conducting the assessments needs a mat, a stopwatch, a box step, a size-appropriate chair, and at least 10-metres of pathway [8].

To carry out the assessment, the assessor gets the subject to carry out 17 sequential tasks. Each task is graded as follows: '2' if there is no obvious modification of activity, '1' if the subject uses a modified method but achieves the goal independent of physical assistance from another, and '0' if the subject is unable to complete the activity independently. The 17 activities involved in the assessment with the requests to the subject are given below:

1. **Stand:** "Can you stand up tall for me for as long as you can and as still as you can?"
2. **Walk:** "Can you walk from A to B (state to and where from) for me?"
3. **Stand up from chair:** "Stand up from the chair, keeping your arms folded if you can"
4. **Stand on one leg – right:** "Can you stand on your right leg for as long as you can?"
5. **Stand on one leg – left:** "Can you stand on your left leg for as long as you can?"
6. **Climb box step – right:** "Can you step onto the top of the box using your right leg first?"
7. **Climb box step – left:** "Can you step onto the top of the box using your left leg first?"
8. **Descend box step – right:** "Can you step down from the box using your right leg first?"
9. **Descend box step – left:** "Can you step down from the box using your left leg first?"
10. **Gets to sitting:** "Can you get from lying to sitting?"
11. **Rise from floor:** "Get up from the floor using as little support as possible and as fast as you can (from supine)."
12. **Lifts head:** "Lift your head to look at your toes keeping your arms folded."

2.3. THE KINEDMD RESEARCH INITIATIVE

13. **Stand on heels:** “Can you stand on your heels?”
14. **Jump:** “How high can you jump?”
15. **Hop right leg:** “Can you hop on your right leg?”
16. **Hop left leg:** “Can you hop on your left leg?”
17. **Run (10m):** “Run as fast as you can to... (give point).”

The NSAA assessment has been shown to be a quick, reliable, and clinically relevant method to measure the functional motor ability of ambulant children with DMD, and is also considered to be suitable to be used in research. It has also been shown to have high intra-observer reliability and high inter-observer reliability [9]. This means that NSAA is generally fairly reliable so as to be used as part of research assuming adequate training is provided to assessors. Furthermore, the hierarchy of items within NSAA was shown to be supported by clinical expert opinion, with items in the NSAA assessment being listed based on their level of difficulty which is agreed upon by most experts, while a questionnaire-based study shows that clinicians generally consider NSAA as clinically relevant [10].

2.3 The KineDMD Research Initiative

The project undertaken as described in this report is part of a wider research initiative known as the ‘KineDMD’ study, conducted by Imperial College London in collaboration with Great Ormond Street Hospital (GOSH). The study involves around 20 DMD subjects (‘D’) subjects and 10 healthy control (‘HC’) subjects who participate in the study for 12 months, who are assessed wearing a sensor suit on selected days during clinical assessments at GOSH, along with using fitness tracker bracelets in the form of Apple watches throughout the trial, which collect data of everyday movements while the subjects are at home or school. A broad aim of the research initiative is to make use of AI to make sense of the data patterns collected from the suit and watches for each of the subjects which, from there, would aid doctors in being able to monitor disease progression with more precision [11]. The initiative has been funded with £320,000 through the Duchenne Research Fund to develop and test the bodysuit that captures the motions of subjects suffering with DMD [4].

The hope is that insights found would cut down the time taken to test new treatments and thus drive down the costs of future clinical trials. A further aim of the initiative is that the developed suit and associated AI techniques and research projects undertaken as part of the initiative will help determine whether any new treatment regimes are working, which would be able to help inform doctors on future treatments. This is particularly useful for specialists, as the condition can be difficult to treat due to relatively slow progression and each subject responds uniquely; furthermore, many of the assessments are done by ‘eye’ instead of using measurement and

objective methods. The initiative hopes that bringing AI techniques into the assessments will take a lot of the human fallibility elements out of the assessments and give a more informed perspective that is better able to understand the progression of the condition in the subjects.

Chapter 3

Overview of Recurrent Neural Networks

3.1 Outline

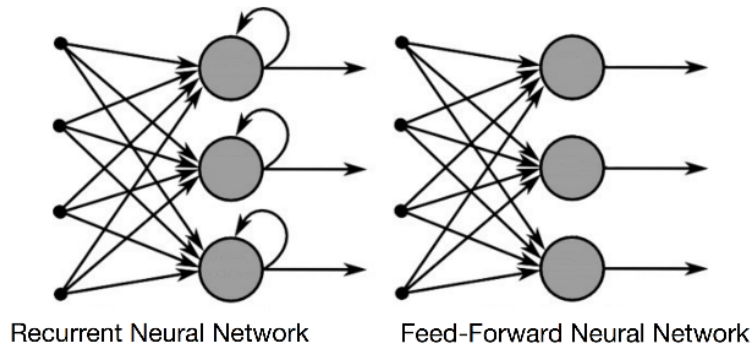
As recurrent neural networks (RNNs) will form the basis of the engineering aspect of the project through implementations using Python and supporting libraries, it's necessary to outline some of the theory and applications prior to implementing them; in doing so, we can explore why exactly they are useful when adapted to work with the body suit data we're concerned with and to solve the problems we're looking to solve. We begin by exploring the shortcomings of feedforward neural networks (FFNNs) and how using RNNs in their place works to overcome them (and, in particular, why they're applicable here). We then touch on the mathematical basis of how the hidden nodes' states change with respect to time and the input, along with how we use long short-term memory (LSTM) units to help deal with the vanishing/exploding gradient problems. Finally, we look at how we can implement this type of network within a Python script by means of the TensorFlow framework, including how we build the model, train it, and test it on unseen data.

3.2 Motivation, Architecture, and Training

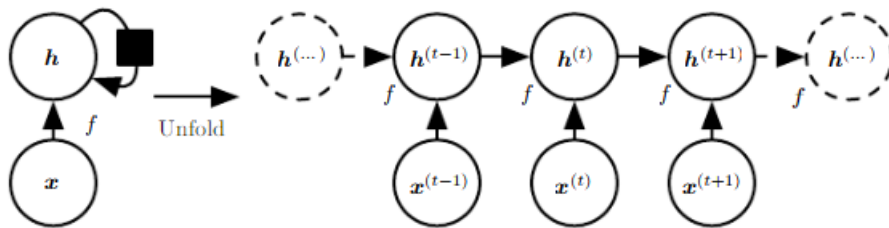
A problem with FFNNs is that they don't handle the order of data that is fed into the network: each sample fed through and classified or regressed is independent of all other samples. For example, if a convolutional neural network is trained to determine whether images are either of a cat or of a dog, then its assessment of each image's label is independent (rightly so) of the images it's seen before. This is appropriate in many situations; however, here we're dealing with time-series data from the suit, where each row ('frame') of joint angle values, position values, and so on that are contained in a '.mat' file produced by the body suit is a single sample in time. We also know instinctively that these values in real-life are dependent on previous values: for example, for a person in movement, the position of their various body parts are influenced by what they were a short time ago. FFNNs don't handle order

of the values of data that are fed in. For example, if there were inputted numerous frames of data from the body suit, then it would treat each as independent entities with regard to the network's predictions.

This lack of memory with FFNNs is something that RNNs attempt to fix in the following way: rather than the values of the hidden nodes of the FFNN being only affected by the values that feed into it (e.g. the network inputs or the values from the previous layer), hidden layers in RNNs are also affected by their own previous values. In contrast with FFNNs, RNNs share their weight parameters across different time steps: this allows a sequence to extend and apply the model to examples of different forms and generalize across them (which allows a sentence like “I went to Nepal in 2009” and “In 2009 I went to Nepal” to recognize the year, ‘2009’, in the same way)[12]. The resultant core architectural difference between the two can be seen in the image below:



In this sense, the RNN can be seen to contain a memory of sorts that enforces time-dependencies of data that is fed through it. If we considered a sequential model where the state of a hidden node h^t is modified by not only the input x but also the layer's previous state h^{t+1} , we can essentially ‘unfold’ this dependency with respect to time to get:



In other words, the output of a hidden node at time t is given as h^t and is a function f of the input at this time from the previous layer x^t and the output of the same layer at the previous time step h^{t-1} . This can therefore be seen as the ‘memory’

aspect of an RNN: values from previous parts of a sequence carry over to influence the subsequent parts. It should be noted, however, that this is only within sequences and subsequent sequences are considered to be independent of each other (in the same way that images fed through a convolutional neural network are independent of each other); hence, the choosing of the correct time-dependency in the form of the sequence length is a key aspect of experimentation which we further look into in our experimentation. This idea of unfolding in time can be extended to apply to multiple layers and multiple nodes per layer and can be seen in the general equation that the hidden nodes in an RNN use to calculate their output:

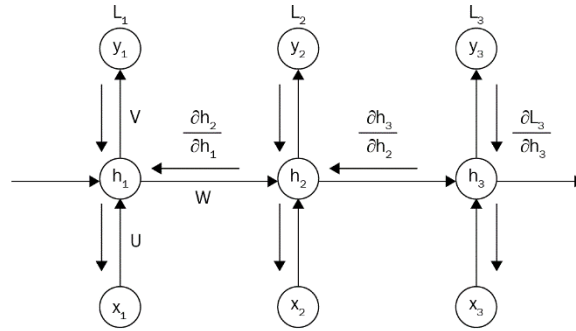
$$h^{(t)} = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

Note that the above function ϕ_h is equivalent to f in the above diagram, as in both cases they represent the activation function for layer h . We interpret this as the output of a hidden layer at time step t being a combination of the previous hidden layer output and the current input to the hidden layer, with both being modified by their respective weight matrices. It's also important to note that, not only is there a weight matrix W_{xh} that is learned through training to map from the previous layer's values to the current one, but there is an additional weight matrix W_{hh} that is learned to control how much of the same layer's previous values impact the current layer. Alternatively, a way to look at it is the W_{hh} matrix controls the influence each left-to-right arrow in the unfolded sequence image has on the state it points to, while the W_{xh} controls how much influence up down-to-up arrow in the unfolded sequence image has on the state it points to.

This requirement of using the additional weight matrix for the states of the hidden layers is also reflected in the backpropagation through time (BPTT) equation for the updating of the W_{hh} matrix via gradient descent:

$$\frac{\delta L^{(t)}}{\delta W_{hh}} = \frac{\delta L^{(t)}}{\delta y^{(t)}} * \frac{\delta y^{(t)}}{\delta h^{(t)}} * \left(\sum_{k=1}^t \frac{\delta h^{(t)}}{\delta h^{(k)}} * \frac{\delta h^{(k)}}{\delta W_{hh}} \right), \text{ where } \frac{\delta h^{(t)}}{\delta h^{(k)}} = \prod_{i=k+1}^t \frac{\delta h^{(i)}}{\delta h^{(i-1)}} = \frac{\delta h^{(t)}}{\delta h^{(t-1)}} * \frac{\delta h^{(t-1)}}{\delta h^{(t-2)}} * \dots * \frac{\delta h^{(k+1)}}{\delta h^{(k)}}$$

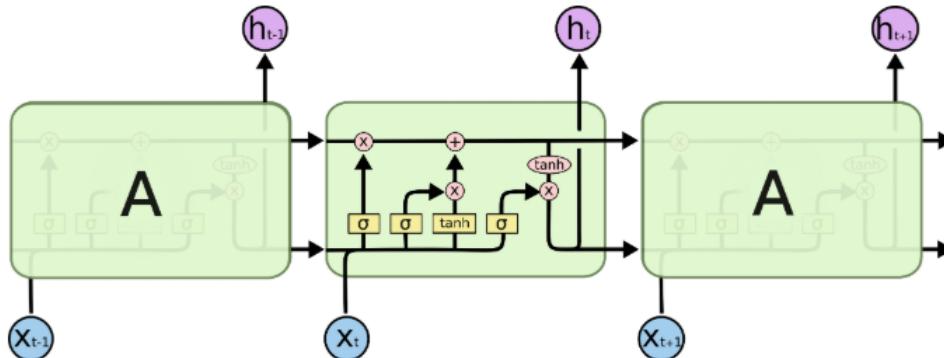
The idea is that the overall loss L is the sum of all the loss functions from times $t = 1$ to $t = T$, and since the loss at time t is dependent on the hidden units at all previous time steps, the gradient is as seen above. Note that its chain rule structure is still very similar to standard backpropagation used in FFNNs, with a primary difference coming from the impact of all previous values of the hidden layer prior to time t on the overall derivative of loss with respect to W_{hh} . Below, we can also see how these derivative values propagate backwards through time:



However, a large problem arises from the $\frac{\partial h^t}{\partial h^k}$ terms having $t - k$ multiplications in the above equation, which therefore multiplies the weight matrix W_{hh} as many times. If this weight matrix is less than 1, this factor becomes very small, which results in a vanishing gradient, severely impacting the ability of the network to train on data and thus learn anything useful (the opposite happens if the weight matrix is greater than 1, which results in the network having an exploding gradient and never converging). To counteract this, a common way to implement sequence models is by using gated RNNs and, for this project, we chose to use LSTM units.

3.3 The Long Short-Term Memory RNN Architecture

The idea of using gated RNNs, which includes the LSTM architecture, is that we are able to create paths through time that have derivatives that neither vanish nor explode and involve connection weights that may change at each time step. Gated RNNs are also automatically able to decide when to clear a hidden state (i.e. set it to 0), and a core idea of LSTMs is to introduce loops within themselves to produce paths where the gradient can flow for long durations of time, while the weight on this internal path loop is conditioned on context, rather than fixed as in the standard RNN. The weight of this path is controlled by another unit and thus the time scale can be changed based on the input sequence [12]. A diagram of a single LSTM network cell and how it interacts with the wider RNN can be seen in the image below, with the LSTM unit itself being the central part of the three green boxes below:



The central idea of this input is the horizontal line running through the top of the unit which allows the data to run along the unit relatively unchanged if required. The gates, represented in the above small yellow boxes within the central green box, allow the unit to let information in (we shall denote these as gates 1 to 4, where gate 1 is the leftmost yellow box and gate 4 is the rightmost box). These gates are there to protect and control the cell state [13].

Gate 1 is the ‘forget gate’, which decides which information to ‘throw away’ from the cell state and is a combination of h_{t-1} and x_t and outputs a number via the sigmoid function σ between 0 (which signifies to ‘get rid of this completely’) and 1 (‘keep this completely’). This could see applicability with a word sequence (i.e. a sentence) where we wish forget older parts of a sequence in order to make a more accurate assessment of the next word of the sequence based on more recently occurred words. The output of this gate f^t is given as:

$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

Gate 2 is known as the ‘input gate’, which decides the values we’ll update within the cell in order to store new information in the cell state. This is used in conjunction with Gate 3, which is a ‘tanh’ gate that creates a vector of new candidate values that can be added to the state. The equations governing the outputs of each of these two parts are given as:

$$\begin{aligned} i_t &= \sigma (W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned}$$

These two are multiplied together to get the new information that we wish to store in the cell, which replaces the old information that we have lost via the ‘forget gate’.

With the information we wish to discard having been forgotten and the new information that we wish to replace it with having been calculated, we then turn to modifying the old cell state C_{t-1} into the new cell state C_t . This is done by multiplying the previous cell state by the forget gate output to forget the things we decided earlier to forget, followed by adding the new proposed candidate values scaled by how much we wish to update each value, and is given by the equation that governs the new cell states information:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Now that we have the cell state obtained, we finally decide how much of this information to output from the cell (i.e. a filtered version of the cell). We then use

Gate 4 (the ‘output gate’) to decide which parts of the cell we shall output, which we multiply by the ‘tanh’ of the new cell state C_t (which makes sure the cell state outputs between -1 and 1). This ensures that we only output the parts we decided to output (e.g. in the case of the language model it allows one to only output information pertinent to verbs if that is what comes next in the sequence). The output of the output gate and of the cell itself is thus given as:

$$\begin{aligned}o_t &= \sigma(W_o [h_{t-1}, x_t] + b_o) \\h_t &= o_t * \tanh(C_t)\end{aligned}$$

In using LSTMs as part of our architecture, we enable the models to condition themselves on sequences with some parts forgotten within the cell and also being able to chose which parts of hidden states it wishes to output to the next layer and the next state of the same hidden layer. As a result, this architecture of the hidden units is much more conducive to modelling long-term relationships within a sequence and also being able to train via backpropagation with much reduced effects from the vanishing and exploding gradient problems.

3.4 Implementing an RNN using TensorFlow

With all that said, there still must be a practical way of implementing RNNs using LSTM units as part of the project for the RNN architecture to actually be useful for us. Fortunately, there exists numerous open source APIs and Python libraries that handles much of the underlying details of a neural network that simply needs the user to design the architecture. For this project, TensorFlow was chosen to be the API of choice due to previous experience in using it for implementing RNNs in time-series data in other work, along with excellent supporting documentation being available and the ability to easily utilize a GPU to help with training the model. Further justification can be found in the ‘System Choices’ chapter of the report.

A detailed guide on building an RNN using TensorFlow is beyond the scope of this report; however, it was felt worthwhile to outline some of the central elements of the models that are built in ‘rnn.py’ and how they relate to the concepts outlined above. While we shall give a detailed breakdown of the script itself in the ‘Script Ecosystem Overview’ chapter of the report, we now turn our attention to specific sections of code within ‘rnn.py’ that are particularly significant to the architectural makeup of the models:

- The input shape of the model is setup with a placeholder variable that sets the input size equal to (x, y, z) , where x is the batch size (i.e. number of sequences per training batch), y is the sequence length (i.e. the number of frames of data that is ‘pushed through’ the model per sequence; generally either 60 for

3.4. IMPLEMENTING AN RNN USING TENSORFLOW

raw measurement data or 10 for computed statistical values), and z is the dimensionality of the frames itself (e.g. 66 for joint angle data).

```
tf_x = tf.placeholder(tf.float32, shape=(self.batch_size, self.seq_len, self.features_length), name='tf_x')
```

The equivalent is also done for the y data, depending on the output type the model is training towards.

- We define the LSTM cells, with their size and number of cells (i.e. equivalent to the number of nodes per hidden layer and number of hidden layers, respectively, if we were using a ‘traditional’ RNN) in a single line of code: we define multiple *BasicLSTMCell* objects with a size set as *self.lstm_size* (a hyperparameter that we can tune), a dropout percentage given as *tf_keepprob*, and create multiple of these in a loop, with the number of these given as *self.num_layers*. These multiple cells (i.e. hidden layers of the model) are then used to create a *MultiRNNCell* object, which acts as a wrapper for all the hidden layers of the model:

```
cells = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.DropoutWrapper(tf.contrib.rnn.BasicLSTMCell(
    self.lstm_size), output_keep_prob=tf_keepprob) for i in range(self.num_layers)])
```

- Finally, we set up the model architecture so that the input x held in the placeholder *tf_x* feeds into the ‘cells’ (i.e. the hidden layers), which modifies the initial state of the model throughout the application of the input sequence x to the hidden layers to give us an output *lstm_outputs* and a final state of the layers, *self.final_state*:

```
lstm_outputs, self.final_state = tf.nn.dynamic_rnn(cells, tf_x, initial_state = self.initial_state)
```

- These steps are defined within the *build()* method for the ‘RNN’ class which is called upon by the constructor of the object at object creation. Hence, when we create an RNN object as...

```
rnn = RNN(features_length=len(x_train[0][0]), seq_len=sequence_length, lstm_size=num_lstm_cells,
          num_layers=num_rnn_hidden_layers, batch_size=batch_size, learning_rate=learn_rate, num_acts=num_acts)
```

... it results in setting the attributes of the RNN object, including most of the hyperparameters that influence the architecture of the object, and calling the *build()* method of the object that uses many of these hyperparameters. Thus, the above statement sets up the computational graph that defines our RNN model which is now ready to have data inputted through it for the training process.

- To train the model, the method *train()* is called by the ‘rnn’ object, which results in splitting the training data components *x_train* and *y_train* into batches, whereupon each batch-pair (i.e. a batch of *x_train* with a batch of *y_train*) is placed into a dictionary that matches train components to batches (i.e. it would

match a batch of *x_train* to the *tf_x* placeholder variable described above, which ensures that the *x_train* batch is used as input to the model). Each of these dictionaries are then fed through via the *session.run()* method that specifies we are training the model (which hence calls upon the optimizer within *build()* to train the model) and takes in the dictionary to train the model on each batch:

```
for batch_x, batch_y in create_batch_generator(x_train, y_train, self.batch_size):
    feed = {'tf_x:0': batch_x, 'tf_y:0': batch_y, 'tf_keepprob:0': 0.5, self.initial_state: state}
    loss, _, state = sess.run(['cost:0', 'train_op', self.final_state], feed_dict=feed)
```

- A similar process is then used when we wish to test the model via the *predict()* method called by the 'rnn' object. Much like *train()*, it splits the *x_test* data into batches, which it adds to the 'feed' dictionary (setting the dropout probability to 0% this time via *tf_keepprob:0: 1.0* and the session to use the *test_state* of the model) and calls the *sess.run()* method to push this dictionary through the model to get the predictions made on the batch. We retrieve the predictions based on the output type we are working towards and adds the predictions obtained to the list of predicted values for the *x_test* input:

```
for ii, batch_x in enumerate(create_batch_generator(x_test, None, batch_size=self.batch_size), 1):
    feed = {'tf_x:0': batch_x, 'tf_keepprob:0': 1.0, self.initial_state: test_state}
    if return_proba:
        pred, test_state = sess.run(['probabilities:0', self.final_state], feed_dict=feed)
    elif choice == "overall" or choice == "indiv":
        pred, test_state = sess.run(['logits_squeezed:0', self.final_state], feed_dict=feed)
    else:
        pred, test_state = sess.run(['labels:0', self.final_state], feed_dict=feed)
    preds.append(pred)
```

While there are, however, many other steps in the process of using the 'rnn.py' to build the models, these are some of the crucial steps where we applied knowledge of RNNs and their usefulness to sequence modelling to create a deep learning solution in TensorFlow. And with easy access to other libraries that make reading in data from 'csv' and 'xlsx' files and manipulating it as matrix data easy (e.g. via 'pandas' and 'numpy') and general-purpose machine learning libraries such as 'sk-learn' to help with other tasks (such as the splitting and shuffling of sequences for training/testing, the evaluating of various metrics like mean squared error, and so on), we have all the resources needed to build RNN models using LSTM units using the applicable preprocessing steps to tailor it towards working with our data pipeline and produce results that can be observed and compared within experiment sets and model predictions sets.

Part II

System Preparation: Choices and Setup

Chapter 4

System Choices: Language, IDE, and Libraries

4.1 Python

As with most software-related projects, one of the primary choices that must be made is what programming language to implement the components of the system in, along with what development environment it is to be built using. Both of these have a large impact in the time and ease it will take to develop the system, as well as how optimal it will be running in its final variation. For the choice of programming language, we chose to use **Python (3.6)** to build all scripts from for the following reasons:

- It takes very few lines to implement many things when compared with other languages like Java; for example, the code below shows how a neural network can be implemented in Python in only 9 lines using the Keras library (a wrapper for TensorFlow):

```
num_extra_hidden_layers = 3
num_hidden_nodes = 30
ann = Sequential()

ann.add(Dense(units= num_hidden_nodes, activation= 'sigmoid', input_dim= num_dimensions))
for i in range(num_extra_hidden_layers):
    ann.add(Dense(units= num_hidden_nodes, activation= 'sigmoid'))
ann.add(Dense(units= 1, activation= 'sigmoid'))

ann.compile(optimizer= 'adam', loss = 'binary_crossentropy', metrics= ['accuracy'])
ann.fit(x= x_train, y= y_train, batch_size= 10, nb_epoch= 30)
```

This enables faster development and easier testing of new ideas and concepts than other languages, as we can afford to care less about problems of tricky syntax (e.g. with using C++) and can instead move towards development ‘at the speed of thought’.

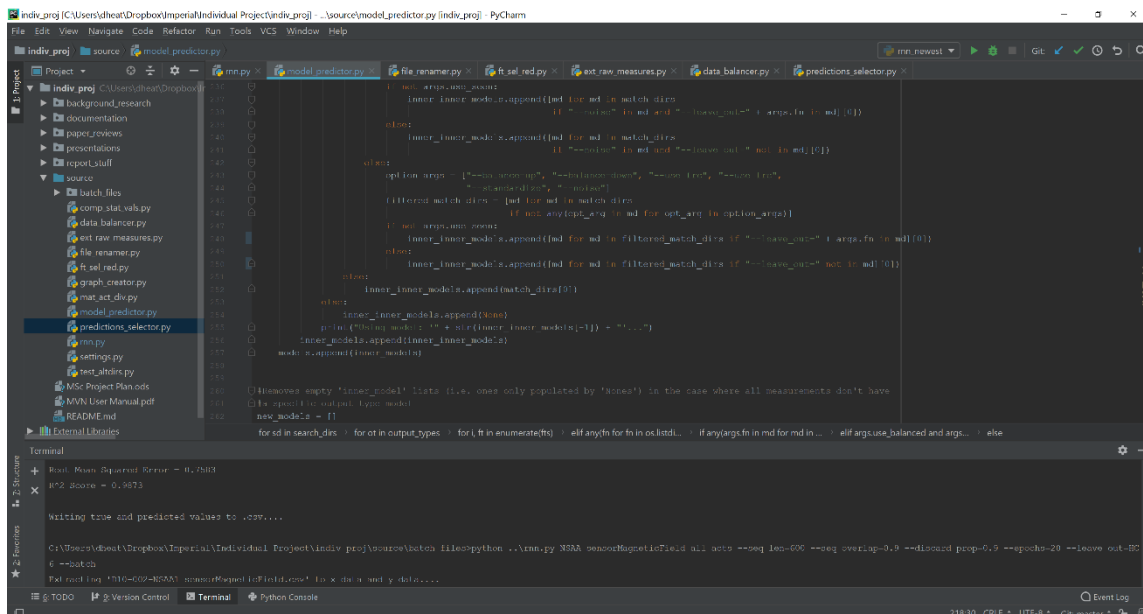
- The open-source nature of Python’s community means that there are very often libraries that others have built that fit the profile of what we need, so we don’t need to ‘reinvent the wheel’ by creating our own version of it; this can be seen

4.2. INTEGRATED DEVELOPMENT ENVIRONMENT

in the system's reliance on external functions from 'scikit-learn' to compute metrics, along with 'pandas' to handle the reading and writing to and from 'csv' files, as opposed to writing our own functions to carry out this functionality.

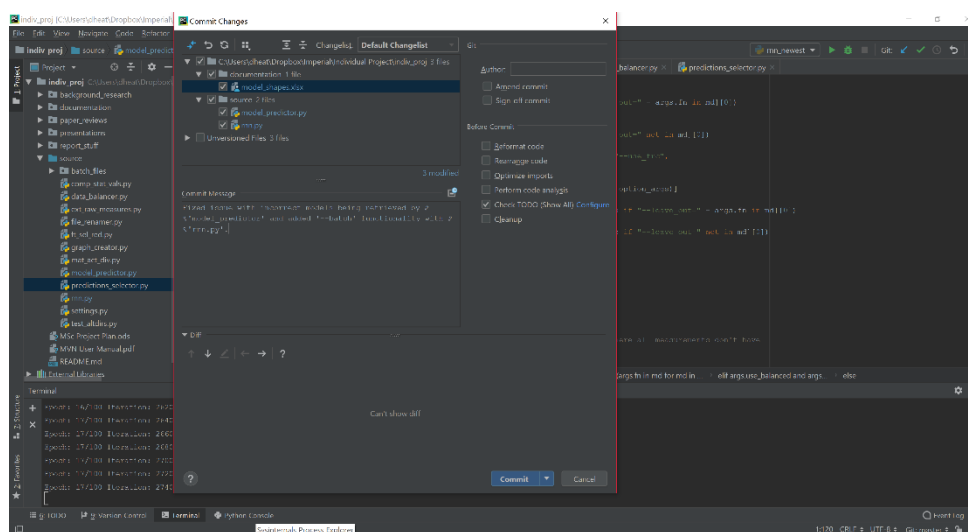
- Python has seen extensive use for building and testing machine learning and deep learning models by research and business communities; thus, it is easily the most well-developed with regards to open-source libraries such as TensorFlow, with TensorFlow's Python API being most complete of its various language implementations [14].
- Many of the frameworks and libraries that power the system we have built (e.g. NumPy and SciPy) build upon lower-layer Fortran and C implementations for fast and vectorized operations for multidimensional arrays, which helps overcome the inferior speed of a scripting language like Python when compared to these lower-level languages [15].
- Further development of the system is easier, as being written in a language like Python (which is close to English) lends itself to the easier understanding of what is going on within each script. Coupled with variables having intuitive names and commenting where necessary, this helps with anyone undertaking edits or rewrites to one or more of the scripts in the future.

4.2 Integrated Development Environment



With regards to where we shall develop the Python programs for this project, we have chosen to use the **PyCharm Community Edition 2016** integrated development environment. We can see above a screenshot of how the IDE is setup for this project. This has been chosen for the following reasons over a text editor or another IDE:

- Multiple program tabs: This enables the easy transitions between different scripts. This is particularly useful when we are making changes to multiple scripts simultaneously (e.g. adding the same optional argument to both ‘rnn.py’ and ‘model_predictor.py’ that ensures certain models built by ‘rnn.py’ are then retrieved correctly by ‘model_predictor.py’).
- In-built terminal: This allows us to run programs from command line within the IDE itself without having to open a separate terminal window and navigating to the script directory every time it’s reopened. This is a major quality of life improvement, as most of the scripts are run from the command line with required and optional arguments.
- Good compatibility with git: This has two main benefits. The first is that changes made to scripts and their specific lines of change from a previous commit are highlighted in blue, which helps with accounting for modifications made when writing commit messages and keeping track of all recent changes made. The other benefit is the GUI approach to committing to the GitHub repo (as can be seen below) which is a much easier way to make regular commits and also highlights easier more subtle changes made (such as writing lines to output files).



- Previous experience: We’ve used it before for many previous Python projects, including in two professional roles, for a final-year undergraduate individual project, and for many pieces of coursework involving the use of several machine learning and deep learning libraries such as ‘scikit-learn’ and ‘TensorFlow’. Hence, this previous experience and the resultant familiarity with the environment helps make development of this project a more expedient and easier experience.
- Debugging and error handling: The layout of PyCharm makes for writing and immediate testing and modifying of code very simple, with debug options

4.3. TENSORFLOW

showing locations of compilation errors very easily and with clarity. This minimizes the time lost in development due to basic syntax errors and other basic software-engineering-related issues.

- Package implementation: It's easy to add additional packages via 'Available Packages' in the 'Project Interpreter', which is useful as we need to add many additional libraries, from TensorFlow to simple libraries like 'pyexcel'.

4.3 TensorFlow

For programming in Python, there are numerous options for which library we can use to implement our central RNN models. One option is the 'Keras' wrapper that wraps the TensorFlow framework. Although this is a lot simpler to use and has fewer aspects to manually code, there are numerous advantages that TensorFlow has over this that includes the following:

- More extensive and highly detailed documentation and examples for TensorFlow.
- Higher amount of direct control over the RNN models with things such as weights and optimizers.
- Better performance with TensorFlow through threading and queues to speed up the training process.
- Availability of the TensorBoard visualization tool to help understand our models.

Thus, using TensorFlow will hopefully lead to more successful RNN models that can better learn from raw measurements and computed statistical values that can thus better operate on newly-presented subject data. Preparatory work for using this framework includes prior use as the engine for an undergraduate individual project, along with reading Chapter 14 and 16 of [15] where we learned:

- The benefits of using TensorFlow for neural network training performance in utilizing GPU cores, where using a high-end GPU resulting in 15 times more floating-point calculations per second than using an equivalently-priced CPU.
- Concepts of graphs, sessions, ranks, tensors and operations, which gave clarity to the concepts of the computational graph structure used by TensorFlow.
- The 'placeholder' concept of TensorFlow where a variable is an 'empty' variable that expects data input (in our case, these 'placeholders' will be implemented for x_{train}/x_{test}).
- How aspects specific to an RNN works in TensorFlow, such as implementing layers as LSTM cells and initial/final states for the variables.

With this obtained knowledge from the above textbook, along with examination of other examples found primarily on GitHub or the TensorFlow documentation website, we felt confident enough in our knowledge of the TensorFlow library that, coupled with prepared input data, our RNN models could now be created.

Chapter 5

System Setup: Software and Data Preparation

5.1 Overview

A necessity to either continue further work on this system, validate the results we obtained in experimentation and elaborate upon in this report, or produce one's own results through system use is to setup the required components to run the system. Before being able to modify or run parts of the system, there are several things that must be setup beforehand. This includes:

- Obtaining relevant system resources.
- Downloading the requisite data sets and setting them up in the correct directories.
- Setting up Python and 'pip', along with the IDE and necessary packages.
- Running all the setup scripts that are run via the 'setup.cmd' script.
- Setup of TensorFlow to use a GPU.

Once these steps are all completed, the editing of scripts, building of models, testing of files, and so on can be done by the user. In this part of the report, we shall be going through all the necessary steps to run the system on a different workstation; the hope is that, with the steps completed in this section, any user with the necessary computational resources can build models, reproduce results, and carry out additional experiments using the suit data captured that is used as part of the project.

5.2 Necessary System Resources

As this system works with large amounts of data and requires a heavy computational workload in order to build and test models, among other tasks, anyone running parts of this system requires a workstation setup with the necessary resources

to execute many of the scripts, store the data, and so on. The vast majority of the work done on this project has been undertaken on a 'Dell XPS 15 (9570)' laptop with the following specs:

- CPU: 'Intel Core i7-8750H'
- RAM: '16GB DDR4, 2666MHz'
- GPU: 'Nvidia GeForce GTX 1050Ti with 4GB GDDR5'
- Storage: '512GB SSD'

A system with similar specs should be adequate to run the system; however, the following is ideal:

- CPU: At least a 7th gen Intel i5 or i7 (or AMD equivalent). A lot of the data preparation, computing of statistical values, reading from and writing to '.csv's, and so on are done using the CPU (i.e. anything the system does that's not including the training, testing, and assessing of models); hence, a good CPU will enable the user to run these scripts in reasonable time.
- RAM: Minimum of 10GB needed; some of the larger datasets we look at (e.g. when multiple raw measurements' data are combined into one data set for a data shape of (16000, 60, 180)) require in excess of 8GB of memory just for the data, not including resources required for the IDE and other parts of the script being run. Any less than 10GB, therefore, may risk system instability or limit the user from carrying out certain parts of the experimentation outlined in this report. Additionally, DDR4 is recommended so as to increase the speed at which data is able to be written and read from memory.
- GPU: We make heavy use of TensorFlow running using the inbuilt GPU; the alternative would be to use the CPU, which by our estimation is approximately 10x slower to train models than using the GPU. Hence, to realistically build models in this system we need a good GPU. Ideally, the user would use an 'Nvidia' card as that is the easiest to setup with TensorFlow and, preferably, the card would have many CUDA cores (the 1050ti has 768) to enable faster parallel processing when training models.
- Storage: As of writing, the total storage required for all data sets contained within the 'local directory' (including '.mat' files for NSAA assessments, 6-minute walks, natural movement behaviour, and the intermediate data extracted from all files via the data pipeline) amounts to over 170GB, while the system itself contained within the 'project directory' requires approximately 725MB of space; hence at least this much storage is required, ideally on an SSD to enable fast read speed from storage (which will happen a lot during the setup scripts). See the chapter on the 'Project and Local Directories' for more information about what these specific directories contain.

5.3 Data Sets Setup

With a workstation obtained with the requisite resources, the next step is obtaining the data sets required by the system. There are two approaches that can be taken: the first involves being able to access the link shown below (which should be possible for anyone with Imperial College London credentials), which contains all the data used as part of this project (i.e. that also contains all the intermediate data constructed via the scripts that make up the data pipeline). Hence, a user that downloads all the data from this link would not need to run the Python scripts contained within the ‘setup.cmd’ script, only the pip installation commands. The second approach should be taken if the user either doesn’t have Imperial College London credentials or otherwise can’t access the files, or if one wishes to run the pipeline ‘from scratch’ (i.e. computing ones own intermediate data files via the data pipeline); this does require the user to fully run the ‘setup.cmd’ script. Note that the below explanations assume the user already has access to the complete ‘project directory’ if one is reading this report; however if not, consult the chapter on ‘Project and Local Directories’ for guidance on how to access this.

The first approach is as follows:

1. Setup a base directory in the user’s storage directory; the default name for this that has been used thus far has been ‘C:\msc_project_files\’; however, a more intuitive name can be chosen by the user if desired. This becomes the user’s ‘local directory’ for the project.
2. Download each of the directories contained within the OneDrive link: https://imperiallondon-my.sharepoint.com/:f:/g/personal/djh18_ic_ac_uk/Euymu00dXG1Cmm-e=L5C62Z and place each directory in the ‘msc_project_files’ directory (e.g. ‘left-out’, ‘allmatfiles’, etc.) within the user’s created ‘local directory’.
3. Modify the requisite line within ‘<project directory>\source\settings.py’ to point to this new location. For example, if the user has setup the ‘local directory’ as ‘example’ in ‘C:\’, then they should modify the ‘local_dir’ variable (line 7) to now contain: ‘local_dir=“C:\example\”’. In doing this, it ensures that all other scripts that need to access the data files in ‘example’ are correctly pointed to it.
4. Once the steps outlined in the section below on Python, pip, PyCharm, and the necessary packages have been undertaken, open ‘<project directory>\source\batch\setup.cmd’ for editing in a text editor, comment out line 19 and onward (as these create the intermediate data from the pipeline which we now already have), save the file, and run it to setup the Python packages.

The second approach is as follows:

1. Setup a base directory in the user’s storage directory; the default name for this that has been used thus far has been ‘C:\msc_project_files\’; however, a more

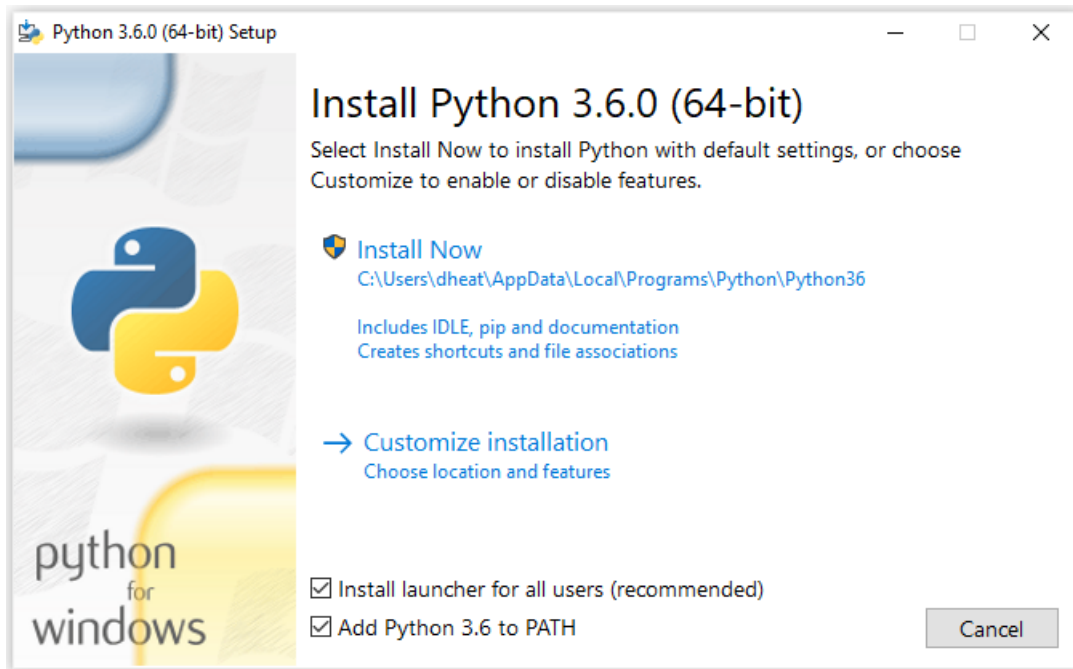
intuitive name can be chosen by the user if desired. This becomes the user's 'local directory' for the project.

2. Obtain permission from the owners of the data sets used as part of the KineDMD research initiative to access and download the directories.
3. Create another directory within the 'local directory' called 'output_files'. This shall contain a number of things produced by the scripts and by the models, including the '.csv' files of computed statistical values, the constructed models themselves, among other parts of the system.
4. The user should have links to the following data sets (though if they don't the requisite permission for each must be obtained by the relevant parties): 'NSAA', 'NMB', 'allmatfiles', '6MW-matFiles', and '6minutewalk-matfiles'. Each of these directories should then be downloaded and directly placed within 'local_dir'.
5. Once the steps outlined in the section below on Python, pip, PyCharm, and the necessary packages have been undertaken, run the 'setup.cmd' in full to obtain the necessary Python packages for the project and compute the intermediate data used as part of the project.

With these steps done, the data should now be in a form in which all the scripts that form the system should be able to access with the necessary directories constructed.

5.4 Setup of Python, Pip, Necessary Packages, and PyCharm

We now turn our attention to the setting up of the language, the 'pip' tool for package installation, and the libraries required to run the scripts. The first step is the installation of Python; the version this system runs on is '3.6.0' and, while installing any subsequent versions should be acceptable, we shall install this version to avoid any possible complications to do with the language further down the line. Version '3.6.0' can be downloaded from <https://www.python.org/downloads/release/python-360/> and by selecting the relevant installer from 'Files'. With the installation window open, ensure that the 'Add Python 3.6 to PATH' box is checked; this shall ensure that the user is able to run Python commands from the command line or terminal:



With this installed, we can ensure that Python is setup correctly with the required version by entering ‘python –version’ at the command line:

```

C:\> Command Prompt
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\dheat>python --version
Python 3.6.0

C:\Users\dheat>

```

It should be noted that the installation window for the user should also say that it is installing ‘pip’. This can be asserted to have been installed for the user by entering ‘pip’ at the command line, which should show output of something like this:

```

C:\Users\dheat>pip

Usage:
  pip <command> [options]

Commands:
  install           Install packages.
  download          Download packages.
  uninstall         Uninstall packages.
  freeze            Output installed packages in requirements format.
  list              List installed packages.
  show              Show information about installed packages.
  check             Verify installed packages have compatible dependencies.
  search            Search PyPI for packages.
  wheel             Build wheels from your requirements.
  hash              Compute hashes of package archives.
  completion        A helper command used for command completion.
  help              Show help for commands.

```

If, for whatever reason, ‘pip’ has not been installed, follow the steps outlined at <https://pip.pypa.io/en/stable/installing/>, which includes the modification of the PATH environment variable to ensure that we are able to use ‘pip’ correctly.

Once ‘pip’ has been installed and/or asserted to be setup ready to use, we are able to install the required packages. Navigate to the ‘<project directory>\source\batch_files’ directory and execute the ‘setup.cmd’ script. This will setup the required packages to run the project. Not only that, but it will ensure that the versions of each of the packages installed for this system will also be setup by the user; this should minimize the chances that the user will encounter dependency issues between packages (different versions of ‘tensorflow’ have been shown to interact badly with certain versions of ‘numpy’, for example). It should be noted that ‘setup.cmd’ will also run all the Python scripts within the ‘source’ directory that are used to setup the files for other scripts such as ‘rnn.py’ and ‘model_predictor.py’, so if the user doesn’t wish to run these at this time, the recommended way is to comment out each of these lines. This can be done by commenting out each of these setup lines (line 19 and onward) by inserting ‘REM ‘ at the beginning of each line. Note that this means that ‘setup.cmd’ must be run later when the user wishes to prepare the data for the system and if the user doesn’t already have the intermediate data.

5.5 Installation of PyCharm (IDE)

If the user is intending to do any long-term modifications to the system, or if they simply want a more convenient place to launch the scripts from, then it is recommended that they install an IDE for the project; specifically, PyCharm Community Edition. Using this provides several advantages to the user:

- An in-built terminal to run the scripts of the system with the necessary arguments.
- Easy interaction with Git to work from the project code in GitHub/GitLab.
- Syntax assistance when editing any files.
- Multiple tabs to help with editing multiple files simultaneously along with the script variable explorer.

To setup the IDE, the following steps should be undertaken:

1. Download the community edition of PyCharm, the link for which can be found at: <https://www.jetbrains.com/pycharm/download/#section=windows>.
2. In the ‘Installation Options’ window, it’s recommended that the user selects the ‘Add “Open Folder as Project”’ option (to allow opening the ‘project directory’ as a PyCharm project) and the ‘.py’ association (so Python files open in PyCharm as default).

5.6. CONFIGURING OF TENSORFLOW TO USE THE GPU

3. Launch PyCharm and select 'Do not import settings'.
4. From the 'Customize PyCharm' window, click 'Skip Remaining and Set Defaults'.
5. Prior to continuing with PyCharm setup, we first must setup Git if the user doesn't have it already. The following section applies to Windows, but the equivalent can easily be done for iOS or Linux.
 - (a) Download Git from <https://git-scm.com/download/win>.
 - (b) Run the installation, making note of where Git is installed, with the default settings.

With this now completed (or not, depending on whether Git is already installed), the user should launch PyCharm and in the 'Welcome to PyCharm' window, click 'Configure' and 'Settings', followed by navigating to 'Version Control' and Git. In the 'Path to Git executable', enter the location of the 'git.exe' program that was just installed (or previously installed). This can be found within the 'Git\bin\' directory within the location where Git was setup. Click 'Apply' and 'OK'.

6. In the 'Welcome to PyCharm' window, select 'Check out from Version Control' and Git. This is because we will be directly installing the 'project directory' directly from 'GitHub'. Note that it's recommended doing this even if the source directory has already been downloaded and setup elsewhere, as doing it this way ensures that it is easily to modify and commit to git if any changes to the scripts are to be made. Within this new window, enter the URL: https://github.com/dan-heaton/MSc_indiv_project within the URL window and click 'Clone' to clone the repository. Alternatively, if one wishes to clone from GitLab instead, swap the URL above with: https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project.
7. The final step is to associate PyCharm with the Python interpreter we have previously installed. To do so, with the project opened in PyCharm, navigate to 'File' and 'Settings'. Under 'Project: MSc_indiv_project' and 'Project Interpreter', click the settings icon and 'Add...'. Under 'System Interpreter' the Python executable should already be detected within 'Python36\' as 'python.exe'. However, modify this path to the 'python.exe' file if has not done so already. Click 'OK', 'Apply', and 'OK'. In the main PyCharm window, it may take a few seconds to configure this interpreter but, once done, the user will be able to edit and run the scripts of the system within PyCharm.

5.6 Configuring of TensorFlow to Use the GPU

The following section works on the assumption that the user is working on a workstation containing a GPU. This is more or less a necessity to build models using

'rnn.py': while a reasonable GPU with >700 CUDA cores builds a typical model in 5-15 minutes, building these using an equivalently-priced CPU would take 1-3 hours. While the necessary steps to undertake the complete setup of a GPU are somewhat arduous, there exists a useful guide to doing so at <https://www.tensorflow.org/install/gpu>, along with a CUDA installation guide at <https://docs.nvidia.com/cuda/> for multiple OS's. This includes details on setting up the CUDA toolkit and the CuDNN (CUDA Deep neural Network library), which are required to run TensorFlow on the GPU. With this setup, TensorFlow will subsequently run as default in all scripts using the GPU to create models as opposed to using the CPU.

Part III

System Overview and Explanation

Chapter 6

Project and Local Directories: Overview and Explanations

6.1 Background

Prior to undertaking an in-depth discussion of the individual scripts, their uses, and the various experiment and model prediction sets we undertook as part of this project, it's worth giving a brief explanation of the two types of directories that we are concerned with for this project. There are two directories that we are concerned with: the 'project directory' (containing the source files, the documentation used and written to, among other things) and the 'local directory' (containing the source '.mat' files that we use as inputs to scripts and some of the outputs of the models, including the models themselves). Below, we cover each of the two in turn, how to access and/or set them up, and so on. The aim is thus to educate any users on how the project is laid out and how each part of the system connects to each other.

6.2 The Project Directory

The project directory is the directory containing all of the source code, the information required about the NSAA subjects ('nsaa_6mw_info.xlsx'), the documents containing the results of the experiment sets and model prediction sets, and other reports and presentations constructed throughout the duration of the project. Hence, this is where the majority of the deliverables of the project lie, with the exception of the majority of the models that were created throughout the project and the data that is required to train the models. The reason we keep these apart (i.e. why project directory and local directory are not synonymous) is as follows:

- The local directory requires >170GB of storage for all the data sets that is used in the project. However, we've been making extensive use of storing the project directory within DropBox and as a Git repository and, as it would be not possible and very impractical, respectively, to store the local directory on both DropBox and within GitHub or GitLab, we chose to keep these separate.

6.2. THE PROJECT DIRECTORY

- When we run several of the data pipeline scripts (e.g. ‘comp_stat_vals.py’ or ‘ext_raw_measures.py’) we end up producing a lot of new files within the local directory). Therefore, to avoid constant DropBox synchronizations or many new or deleted files to appear in each Git commit, it was felt that it would be easier to simply keep the data aspects apart from the source code and other documentation.
- Permission may be required to deal with certain data directories contained within the local directory, as this contains data about ongoing subjects of a research initiative that is not freely available. Hence, it was the desire to keep the project directory available to whoever wished to access it, without predicated access on also being able to access the data required to populate the local directory (e.g. if one wished simply to browse or borrow ideas from the scripts within the project directory); this also enables and encourages an easier transition to applying the project to other data sets possibly for other domains.

To access the complete project, the advisable way to obtain it would be to clone it via GitHub. The repo can be accessed at https://github.com/dan-heaton/MSc_indiv_project and cloned via https://github.com/dan-heaton/MSc_indiv_project.git. Alternatively, one can access it via GitLab at https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project and clone with https://gitlab.doc.ic.ac.uk/djh18/MSc_indiv_project.git. The current name for the project directory as used in its local form for development has been ‘indiv_proj; however, one could rename this freely without requiring any other changes to the scripts. However, it’s recommended that the directories without the project directory should not be changed (e.g. ‘source’ or ‘documentation’), as doing so would require rewriting of several of the scripts that hardcoded paths to access things outside of its own directory.

Broadly speaking, the directories can be broken down as the following:

- **background_research:** this folder contains some preparatory programs that were written (with help from sources cited in the scripts themselves) to familiarize oneself with the building of RNNs with the chosen libraries in order to make using them for the actual project later that much easier; hence, these aren’t used by the rest of the project at all and are included for historical documentation reasons only.
- **documentation:** contains a number of files pertaining to the outputs of the data pipeline for the project. This includes the following:
 - ‘RNN Results.xlsx’, which covers the performances of various model setups on test data (i.e. a large proportion of the experimentation that covers different types of raw measurements, sequence lengths, overlaps, etc., source their results from here). These are what form the basis of our later discussion on the experiment sets.
 - ‘model_predictions.csv’, which (unlike ‘RNN Results.xlsx’) shows the performance of using ‘model_predictor.py’ to assess the performance of pre-trained models on whole data files. These provide the information needed for the model predictions sets, as discussed later.

- *'model_shapes.xlsx'*, which is just to be used by *'model_predictor.py'* to set the sequence length to the correct value (and is not particularly relevant to the user).
- *'nsaa_6mw_info.xlsx'*, which contains a table of the subject names and their corresponding single-act and overall NSAA scores (this provides the necessary *y_labels* for many RNN models).
- *'nsaa_17subtasks_matfiles.csv'*, which is the Google annotations sheet that was collaboratively created by others within the wider research initiative that contains the file names and times within said files where the 17 NSAA activities are performed by each of subjects. This is determined by watching the source videos of the *'mov'* files of the subjects performing the activities and recording at what times in the video these activities occur, along with making use of the *'dis_3d_pos.py'* script; this sheet is then used by *'mat_act_div.py'* to create the single-act files used in later model predictions sets.

We also have several other subdirectories in *'documentation'*:

- **Graphs:** all graphs created by *'graph_creator.py'* are placed in here. These source from *'RNN Results.xlsx'* and *'model_predictions.csv'* to create graphs that are easier to display the results of groups of experiments done than it would be to display the same information using a table. We see many of these graphs later on within the discussion of the experiment sets.
- **Script Explanations:** collection of *'READMEs'* for each of the scripts within *'project_files\source'*. The idea is that, if one wishes to find out what each script does, why it was written, etc., then reading its relevant *'README'* should provide sufficient detail. Much of these *READMEs* form the basis of our script overview later on.
- **paper_reviews:** contains paper reviews done of research papers that are believed to be relevant to the project. These predominantly focus on the use of RNNs when applied to real-world human movement data, and each paper consists of a slightly-shortened bullet-pointed version of the paper and then a section of the most significant points from these bullet points. Hence, these papers are useful in justifying decisions taken with respect to model choices, experimentation directions, etc.
- **presentations:** contains a collection of presentations that have been created to display to group members about the project's progress thus far (which were kept in order to aid in final report writings).
- **report_stuff:** contains several initial reports and other documentations of project progress thus far, and also *'MSc Project Plan.ods'*, which is where the already-completed and upcoming task lists are stored; this is particularly useful if one wishes to see what is currently being worked on or has recently been completed. The vast majority of the contents of this directory, however, are contained within this report.

- **source:** contains all of the scripts that are needed by the project pipeline to run. This includes the core Python scripts (such as `'comp_stat_vals.py'` and `'rnn.py'`), along with some 'supporting' scripts, such as `'settings.py'` (to contain global variables that are used across several scripts). For information about how to run each of these scripts, run the script of interest through the command line/terminal with the `'-help'` optional argument set (e.g. `'python comp_stat_vals.py -help'`). This will display each of the arguments that are available to be set, the significance of each, how they interact with other arguments (if relevant), etc.
- **Batch files:** within this, we contain the batch scripts that are used to automate some of the running of the scripts. For further info about the significance of any or all of the scripts, consult the `'README(s)'` for the relevant scripts in `'<project directory>\documentation\Script Explanations\'`, the script ecosystem overview in `'plans_and_presentations'`, or the diagram of the scripts and their connections to each other found at the beginning of the `'Script Ecosystem Overview'` chapter. Along with some of the simpler automation of the tasks, we also run each of the model predictions sets from their respective batch files, as many of them require building many models and testing many different combinations of files, which require many runs of `'rnn.py'` and `'model_predictor.py'`; hence, the automation of this makes the process of replication hopefully much easier for the user.

6.3 The Local Directory

The local directory is considered to have two purposes: to store the data sets that we make use of in this project, and to store the direct outputs of the `'rnn.py'` scripts that include the models themselves and the `'.csv'` output predictions that are written directly on a sequence by sequence basis (e.g. for a model created with `'rnn.py'` using a test set of 1000 sequences, there will exist within this `'.csv'` each 1000 predicted value and true value, depending on the output type set by the user). There are three reasons why we include this `'rnn.py'` output within the local directory:

- As we create many models as part of this project, the size of this directory has become an issue and thus we would prefer to keep it separate from the project directory for space reasons. We therefore want to keep a lot of this data 'clutter' apart from the what is considered the 'core' of the project with the project directory.
- We also want to keep a consistent philosophy with which we consider to be 'intermediate data', which is data that exists as a product of one script and that is used by other scripts: in this case, the models produced are intermediate data in that they are created by `'rnn.py'` and used by `'model_predictor.py'`. This holds for other forms of intermediate data such as data produced by

‘comp_stat_vals.py’ and ‘ext_raw_measures.py’, and so we wish to do the same for the models and ‘rnn.py’ predictions output.





- The majority of these models are only used once as part of one particular experiment set or model predictions set, and therefore they don’t form a part of the ‘complete’ system (with the exception of the final selected models that are contained within the ‘source’ directory, though this is only a small number of the overall number of created models). Thus, we keep these models separate from the ones constituting the completed system at the end of all experimentation; in other words, the models that are intended to be used by a user to do assessments with specific subjects are contained within the project directory, while the models created during all experimentation are contained within the local directory.

To access the complete local directory of files that we have been used for this project, use the OneDrive link given as https://imperiallondon-my.sharepoint.com/:f:/g/personal/djh18_ic_ac_uk/Euymu00dXG1Cmmeoz3xxq24BekH57ZuDmU9uZtoSr60xfg?e=L5C62Z where one can find a directory given as ‘msc_project_files’. This is the local directory as used during the development of the project. Download and store it while modifying the local variable in ‘settings.py’ (as directed in the ‘System Setup’ chapter). Note that the total directory is in excess of 170GB, so sufficient space may need to be made for it beforehand.

6.3.1 The Local Directory: ‘rnn.py’ Outputs

We first look at the two sub-directories within the local directory containing the outputs of ‘rnn.py’:

- **output_files\rnn_models:** this contains all models that have been produced by ‘rnn.py’ throughout the course of the project, including the final models used that are contained within the project directory. Each model’s contents are the product of the TensorFlow library working through ‘rnn.py’ and each model consists of a directory that looks something like this:

| | | | |
|--|------------------|--------------------|----------|
|  checkpoint | 19/08/2019 14:19 | File | 1 KB |
|  model.ckpt.data-00000-of-00001 | 19/08/2019 14:19 | DATA-00000-OF-0... | 2,738 KB |
|  model.ckpt.index | 19/08/2019 14:19 | INDEX File | 1 KB |
|  model.ckpt.meta | 19/08/2019 14:19 | META File | 261 KB |

These files constitute a single model in the eyes of ‘model_predictor.py’, which is the only script that makes use of these models. For instance, within these files contains the model input and output shapes, the numbers of hidden layers, other hyperparameter settings, and the weights of each of the neuron connections. In other words, it’s a fully trained model that is ready to be used by ‘model_predictor.py’ to be used on a whole subject’s data file.

The names of the models may seem unnecessarily long and complex, but they

6.3. THE LOCAL DIRECTORY

are in fact simply the exact arguments used to invoke the instance of ‘rnn.py’ that creates this specific model, excluding the always-necessary ‘python rnn.py’ parts of the argument sequence. There are two reasons why we do this:

- This creates the names of the directories automatically, which takes much of the human-element of labelling each directory out of the process based on what experiment set or model prediction set it is used for.
- In having them written by name based on the ‘rnn.py’ arguments, we can ensure that ‘model_predictor.py’ will always use the correct models during experimentation by writing a simple set of rules within ‘model_predictor.py’. For instance, if we want ‘model_predictor.py’ to use models that have been created with added Gaussian noise, we can ensure that it uses only model directories that contain ‘-noise’ within their names. Hence, it’s an easier way to determine the correct models to use rather than analysing the contents of the model directory (e.g. which would mean looking into the ‘model.ckpt.meta’ file) to determine if it’s a model we need to use.
- **output_files\RNN_outputs:** each model that is built by ‘rnn.py’ that is also tested on a test partition of data (i.e. if the ‘-no_testset’ argument is not set) writes a separate ‘.csv’ file to this directory. Each file contains, for a given created model, the arguments used to invoke it (in the form of the ‘.csv’ file name), the ‘overall’ results of the test set on the model (e.g. MAE of data for the ‘overall’ output type, accuracy of data for the ‘dhc’ output type, etc.), the individual predictions made for each test sequence versus their true values, and the model hyperparameter settings. An example of this can be seen below:

| Sequence Number | Predictions | Trues | Results | Settings |
|-----------------|-------------|-------|--|--------------------------------|
| 1 | 33.96 | 34 | NSAA,allmatfiles jointAngle all overall -seq_len=600 -seq_overlap=0.5X shape = (25533, 60, 66) | |
| 2 | 34.03 | 34 | Mean Squared Error = 0.4382, Mean Absolute Error = 0.3014, Root M/Y shape = (25533,) | |
| 3 | 33.97 | 34 | | Test ratio = 0.2 |
| 4 | 2.61 | 3 | | Sequence length = 60 |
| 5 | 34.11 | 34 | | Features length = 66 |
| 6 | 2.93 | 3 | | Num epochs = 20 |
| 7 | 25.4 | 26 | | Num LSTM units per layer = 128 |
| 8 | 34.09 | 34 | | Num hidden layers = 2 |
| 9 | 33.77 | 33 | | Learning rate = 0.001 |
| 10 | 34.1 | 34 | | |
| 11 | 34 | 34 | | |
| 12 | 27.9 | 28 | | |
| 13 | 27.27 | 26 | | |
| 14 | 34.05 | 34 | | |

The contents of this file are entirely optional to use, however, as all the requisite information about the test set performance (that is to be inputted into ‘RNN Results.xlsx’ and then used as part of experiment sets) are also produced as console output, which is easier to copy over for the user. However, this files serve as a log of model performance through time as we continue to create more models if we wish to use them as a reference at any point.

6.3.2 The Local Directory: Data Sets

With the directories containing model outputs having been covered, we shall now look at the directories containing the raw data sets, what is contained within each directory, and what ‘type’ of data these directories contain. It should be noted that

the ‘source’ scripts assume that each of these directories are a constant (i.e. that any other users don’t modify the names of the directories); any changes made to the names here require modifications to the necessary variables within ‘settings.py’ (e.g. the ‘sub_dirs’ variable).

Prior to looking at each data set in turn, it’s preferable to briefly discuss the general locations of raw source data (e.g. as source ‘.mat’ files) and intermediate data. This becomes particularly relevant when we shall shortly be discussing the contents of each data directory, and knowing what produced intermediate data and where is conducive towards understanding the data pipeline. Below, we can see how each script within the data pipeline produces data and where the data is stored:

1. **‘comp_stat_vals.py’**: This script takes the data stored as source ‘.mat’ files from within the data directories within the local directory and outputs data to the corresponding path within ‘output’. For example, if ‘comp_stat_vals.py’ intends to operate on ‘NSAA’ (based on the ‘dir’ argument passed to it), it sources its data from ‘<local directory>\NSAA\matfiles’ as ‘.mat’ files and produce the computed statistical values in ‘<local directory>\output_files\NSAA\AD’ as ‘.csv’ files. This functions in the same way for other source data sets (e.g. ‘6minwalk-matfiles’) where the path to the computed statistical value files is more-or-less the same as the source directory path with ‘output_files’ appended after the path to the local directory.
2. **‘ext_raw_measures.py’**: Unlike computed statistical values, the produced raw measurement files are instead stored within subdirectories of the source data set directory that they are sourced from, as opposed to a subdirectory of ‘output_files’. For example, if ‘ext_raw_measures.py’ intends to extract the raw measurements from the ‘6minwalk-matfiles’ directory, it retrieves files from ‘<local directory>\6minwalk-matfiles\all_data_mat_files’ and writes each measurement extracted per file to a sub-directory with a name matching the measurement name (e.g. ‘D4’s joint angle data is written to ‘<local directory>\6minwalk-matfiles\all_data_mat_files\jointAngle’ as ‘D4-6MinWalk-jointAngle.csv’ while its position data is written to ‘<local directory>\6minwalk-matfiles\all_data_mat_files\position’ as ‘D4-6MinWalk-position.csv’, and so on).
3. **‘mat_act_div.py’**: It should be noted first that, as this script extracts single-act files from complete-act files, it therefore only expects to be used on the ‘NSAA’ directory, as only files from ‘NSAA’ contain NSAA activities. When activities are divided, they are placed either within ‘act_files’ or ‘act_files_concat’ (depending on whether ‘-single_act_concat’ was set for the script) within the ‘NSAA’ directory itself, much like ‘ext_raw_measures.py’. For example, ‘mat_act_div.py’ would pull source ‘.mat’ files from ‘<local directory>\NSAA\matfiles’ and write single-act files (non-concatenated) to ‘<local directory>\NSAA\matfiles\act_files’ as source ‘.mat’ files but only containing single-activities within each. Note that if ‘ext_raw_measures.py’ then operates on these single-act files, they get written to a subdirectory within here, much like how it would operate on complete-act files; hence, drawing the joint angle files from the above case would write

the files to '`<local directory>\NSAA\matfiles\act_files\jointAngle`'. Similarly, when we compute the statistical values of single-act files, they get written to the 'output_files' directory; in the above case, the computed statistical values of the single-act NSAA files would be written to '`<local dir>\output_files\NSAA\AD\act_files`'.

4. **'ft_sel_red.py'**: To simplify the process of storing the feature-reduced variants of the '.csv' outputs of 'comp_stat_vals.py', we decided to store the files in the exact same directory as the files they operate on. For example, if we wish to reduce the dimensions of the computed statistical value files for the 'NMB' source data directory, we would write the feature-reduced files to '`<local directory>\output_files\NMB\AD`', where the computed statistical values are already stored. The difference here, however, is that the feature-reduced equivalents will have 'FR_' appended to the front of each of the files. For example, the computed statistical values for the 'D4' subject whose data is from 'NMB' will be stored within the above path as 'AD_D4_stats_features.csv', and when 'ft_sel_red.py' operates on this subject, it will take the data from this file and write to a file stored in the above path as 'FR_AD_D4_stats_features.csv' (alternatively, if the feature-reduced-concatenation option is set within 'ft_sel_red.py' it will instead be stored as 'FRC_AD_D4_stats_features.csv'). This naming convention ensures that 'rnn.py' fetches the feature reduced variants of files to ensure that models of input nodes size 4000 isn't required.
5. **'rnn.py'**: Although this is considered part of the data pipeline, this simply fetches the data from all of the above locations dependent on the arguments given, while we have already discussed the output locations of 'rnn.py' in the previous section.

With the relationship between the data pipeline scripts and the data set directories having been established, we now move on to examining the data that's contained within each of these directories. The data directories are as follows:

- **6minwalk-matfiles**: This contains the 6-minute walk data of many (but not all) of the subjects within two sub-directories within this directory: 'all_data_mat_files' and 'joint_angles_only_matfiles'. The former contains the source '.mat' files for all available measurements for the subjects' walking assessments, while the latter contains source '.mat' files with only joint angles. Therefore, what we consider 'JA' and 'DC' files ('Joint Angle' and 'Data Cube', respectively) that are referenced in experiment set 1 comes from 'joint_angles_only_matfiles', while in most other cases when we use the data of 6-minute walk assessments, we use the 'all_data_mat_files' directory.
- **6MW-matFiles**: This directory also contains some of the 6-minute walk assessment files for some of the subjects and, while some of the files overlap with '6minwalk-matfiles', it also contains assessment data that is not found in the other directory. Additionally, we don't see any 'joint angle only' data within this data set, and so all source '.mat' files are stored directly within this directory.

- **allmatfiles:** This contains the natural movement behaviour as source '.mat' files that contains only the joint angle data (much like '6minwalk-matfiles\joint_angles_only.mat' files). As we only had the natural movement in 'joint angle only' form for quite a while (until we were given the 'NMB' data set), we had to make use of this for several of the later model predictions sets, though we later received the natural movement behaviour in true 'AD' form (i.e. containing all the measurements data for each subject) as 'NMB'.
- **left-out:** This is a small sample of data files that have been excluded from the main data sets that can act as data that is left-out of any of the data sets used to train models. It should be noted the difference between assessing using 'model_predictor.py' on files from left-out as opposed to 'left-out' subjects (as used in many model predictions sets): assessing a file from the 'left-out' directory will assess a model that is familiar with the subject (through having been trained on other files of the same subject), while assessing a complete left-out subject will assess a model that is not familiar with any files for the specific subject.
- **NMB:** This is the complete 'AD' data for the natural movement behaviour, as opposed to 'allmatfiles' which only contains the joint angle data in source '.mat' file form. As a result of receiving this data late in the project lifecycle, we are only able to use this data set in later model predictions sets. It should be noted the sheer number of files per subject: many of the subjects have up to 30 files captured from each of them that will have captured a variety of 'natural' activities, such as sitting, playing, eating, and so on. The data contained within these files, therefore, is much more 'unstructured' than either the 6-minute walk or NSAA assessment data.
- **NSAA:** This contains the data for each of the subjects' full NSAA assessments, and the source '.mat' files specifically are contained within the 'NSAA\matfiles' subdirectory. It should be noted that, for some subjects, their assessments are split over several files. We account for this when extracting raw measurements and computing statistical values by simply concatenating these source files with respect to time, while we select the correct file to use to get the single-act files via 'mat_act_div.py' by referencing the file name columns within the Google annotations sheet.

Chapter 7

Reference Documents Explanation

7.1 Background

As we can see from the system overview diagram shown at the beginning of the ‘Script Ecosystem Overview’ chapter, as well as having many Python and batch scripts that play integral roles within the system we also make heavy use of several documents that provide much needed information for training models and serve as places to store the results of experiments. These include the Google sheet of single-act annotations, the reference document containing the overall and individual scores of each subject used in the project, the ‘model_predictions.csv’ file, and the ‘RNN Results.xlsx’ file. Each of these files in turn can be found within the project source directory within the ‘<project directory>\documentation’ directory. In the section below, we will discuss file each in turn, from the information they contain to how and where they are used within the system (i.e. what other scripts depend on them and what scripts feed into the documents). The aim here, therefore, is to give a more concrete understanding of what these documents contain prior to seeing them referenced in the results discussion and general system overview sections.

7.2 Google Annotations Sheet (‘nsaa_17subtasks_matfiles.csv’)

| Step up R | | | | | Step down R | | | | |
|---------------------------------------|-------|--------|-----------|--|---------------------------------------|-------|--------|-----------|--|
| filename | start | finish | completed | notes | filename | start | finish | completed | |
| D2-010-NSAA | 12981 | 13281 | yes | | D2-010-NSAA | 13881 | 14181 | yes | |
| D3-006-NSAA | 8404 | 8846 | yes | | D3-006-NSAA | 9096 | 9256 | special | |
| d4-004-nsaa1a | 17236 | 17304 | yes | | d4-004-nsaa1a | 17048 | 17124 | yes | |
| d5-007-nsaa1 | 7217 | 7414 | yes | | d5-007-nsaa1 | 8000 | 8244 | yes | |
| D6-003-NSAA | -1 | -1 | no | | D6-003-NSAA | -1 | -1 | no | |
| D7-002-NSAA | 8997 | 9007 | special | was holding rails; actual steps | D7-002-NSAA | 9655 | 9767 | special | |
| D9-002-NSAA | 7098 | 7184 | yes | | D9-002-NSAA | 7299 | 7386 | yes | |
| D10-002-NSAA1 | 1776 | 1776 | no | can't do it | D10-002-NSAA1 | 1776 | 1776 | no | |
| D11-012-NSAA | 14100 | 14220 | yes | | D11-012-NSAA | 14520 | 14640 | yes | |
| D12-003-NSAA | 7800 | 8520 | yes | | D12-003-NSAA | 8520 | 8760 | yes | |
| D14-002-NSAA | 8520 | 8640 | yes | | D14-002-NSAA | 8820 | 8940 | yes | |
| D15-007-NSAAx | 19570 | 19656 | yes | | D15-007-NSAAx | 19777 | 19930 | yes | |
| D17-002-NSAA1 | 4680 | 4800 | yes | | D17-002-NSAA1 | 4860 | 5040 | yes | |
| D18-001-NSAA1 | 10908 | 11196 | yes | | D18-001-NSAA1 | 11770 | 11848 | yes | |
| D19-002-NSAA1 | 4740 | 4920 | yes | | D19-002-NSAA1 | 5160 | 5280 | yes | |
| D20-001-NSAA1 | 11283 | 11403 | yes | | D20-001-NSAA1 | 11523 | 11643 | yes | |
| D4V2-004-NSAA | 12427 | 13182 | special | help given: 12625 end frame without help | D4V2-004-NSAA | 13369 | 13444 | yes | |
| D5V2-010-NSAA1 | 7900 | 8000 | yes | | D5V2-010-NSAA1 | 8000 | 8100 | yes | |
| HC1-004-NSAAAttempt2-NaturalBehaviour | 5476 | 5560 | yes | | HC1-004-NSAAAttempt2-NaturalBehaviour | 5570 | 5671 | yes | |
| HC2-007-PUL-NSAA | 77765 | 77848 | yes | | HC2-007-PUL-NSAA | 77845 | 77921 | yes | |
| HC03-004-NSAA | 7798 | 7891 | yes | | HC03-004-NSAA | 7892 | 7957 | yes | |
| HC4-013-NSAA | 8222 | 8342 | yes | | HC4-013-NSAA | 8462 | 8582 | yes | |
| HC5-009-nsaa | 0 | 0 | yes | | HC5-009-nsaa | 0 | 0 | yes | |
| HC6-004-NSAA | 0 | 0 | yes | | HC6-004-NSAA | 0 | 0 | yes | |
| HC7-008-NSAA | 7030 | 7115 | yes | | HC7-008-NSAA | 7237 | 7303 | yes | |
| HC9-003-NSAA1 | 5961 | 6351 | special | tries to balance on step with one leg rather than step on to it with both legs | HC9-003-NSAA1 | 6489 | 6685 | special | |
| HC10-15-NSAA | 2936 | 3038 | yes | | HC10-15-NSAA | 3290 | 3506 | special | |
| HC11-004-NSAA1 | 5258 | 5348 | yes | | HC11-004-NSAA1 | 5358 | 5452 | yes | |
| HC12-002-NSA1 | 18917 | 19132 | yes | | HC12-002-NSA1 | 19561 | 19684 | yes | |
| HC13-001-NSA1 | 19263 | 19350 | yes | not done as per the protocol; left foot doesn't rest on the step | HC13-001-NSA1 | 19445 | 19512 | yes | |

As part of the wider research initiative, we collaboratively undertook to analyse

and record the times of each activity undertaken by the subjects as part of the initiative. The information that was collected into this document, therefore, was intended to be used as part of several projects that relied on the start and end times of each activity undertaken by the subjects. As a result, the subjects' corresponding activity videos were divided into three parts and each of us determined the activity times for our given subjects.

The process to undertake these annotations was as follows:

1. Load either the source video that corresponds to the '.mat' file (which is provided in the data sets as '.mov' files) OR use the corresponding '.mat' file with the 'dis_3d_pos.py' script to load a basic 3D dynamically updating image through the 'matplotlib' library (more on this in the 'Script Ecosystem Overview' chapter).
2. For each activity of the NSAA assessment set (e.g. 'raise from floor', 'run', 'step up using right foot', etc.), observe the time in seconds (or frames) the activity starts and finishes in the file. Note that we only count the **first completed** activity within the source file and we try to be slightly accommodating of the start time (for example, if the activity started between 3s and 4s in the file, we record it as having started at 3s to ensure we capture the complete activity with a bit of 'slack').
3. For the subject in question and for the activity in question, record the start and end times in **frames** in the 'start' and 'end' columns for the activity (note that if the time was observed in seconds, simply multiply this by 60 as the suit samples at 60Hz), along with recording the name of the file that the activity occurred in (as this will be the same name as the corresponding '.mat' file the suit data for this video will be in, just with a '.mat' file extension instead of '.mov').

In the image above, we can see a snapshot of several activities that have been recorded for some of the subjects, including the file names containing the activity in question for the subjects, along with the start and end times within the respective files. It should be noted that not every activity could be drawn from each of the subjects' files. This could be due to the subject simply not performing the activity they were told to perform or were otherwise unable to perform the activity. In these cases, the start and end times will be marked with either a '-1' or '0', with a '0' sometimes signifying an incomplete activity annotation done by the annotator.

For our project, the main use of this document is as a tool for the 'mat_act_div.py' script. For each of the source files that the script wishes to divide up, it will look in the table for the name of the subject in question for that source file, find the row corresponding to that file and, for each activity, get the start and end times for that activity and extract the corresponding frames from the source file (making sure its name matches that of the activity's 'filename' column entry). So while these single-act '.mat' files will be further processed by other scripts (such as 'comp_stat_vals.py'

7.3. NSAA SCORES REFERENCE DOCUMENT ('NSAA_6MW_INFO.XLSX')

and 'ext_raw_measures.py'), 'mat_act_div.py' is the only file that directly relies on the Google annotations sheet, and no file within the system modifies it in any way.

7.3 NSAA Scores Reference Document ('nsaa_6mw_info.xlsx')

| Class | ID | Visit | 6MWDist | NSAA | Act 1 | Act 2 | Act 3 | Act 4 | Act 5 | Act 6 | Act 7 | Act 8 | Act 9 | Act 10 | Act 11 | Act 12 | Act 13 | Act 14 | Act 15 | Act 16 | Act 17 |
|-------|------|-------|---------|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|--------|--------|--------|--------|--------|--------|--------|--------|
| DMD | D2 | 1 | 250 | 15 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| DMD | D3 | 1 | 275 | 19 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 0 | 1 | 2 | 1 | 2 | 0 | 1 | 1 | 1 |
| DMD | D4 | 1 | 310 | 20 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 0 | 2 | 0 | 1 | 1 | 1 |
| DMD | D5 | 1 | 442 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| DMD | D6 | 1 | 395 | 30 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 1 | 2 | 2 | 1 |
| DMD | D7 | 1 | 375 | 26 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 1 |
| DMD | D9 | 1 | 335 | 22 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 0 | 2 | 0 | 2 | 1 | 1 |
| DMD | D10 | 1 | 125 | 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| DMD | D11 | 1 | 175 | 27 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 2 | 1 | 1 |
| DMD | D12 | 1 | 240 | 18 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| DMD | D14 | 1 | 340 | 28 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 0 | 2 | 2 | 1 | 1 |
| DMD | D15 | 1 | 300 | 28 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| DMD | D16 | 1 | 100 | 23 | 2 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 2 | 0 | 2 | 1 | 2 | 1 | 1 |
| DMD | D17 | 1 | 475 | 33 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 | 2 | 2 |
| DMD | D18 | 1 | 267 | 24 | 2 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 1 | 2 |
| DMD | D19 | 1 | 182 | 24 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 0 | 1 |
| DMD | D20 | 1 | 401 | 29 | | | | | | | | | | | | | | | | | |
| DMD | D4v2 | 2 | 301 | 14 | | | | | | | | | | | | | | | | | |
| DMD | D5v2 | 2 | 465 | 33 | | | | | | | | | | | | | | | | | |
| HC | HC1 | 1 | 485 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC2 | 1 | 525 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC3 | 1 | 490 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC03 | 1 | 490 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC4 | 1 | 469 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC5 | 1 | 450 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC6 | 1 | 450 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC7 | 1 | 442 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC8 | 1 | 450 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC9 | 1 | 425 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC10 | 1 | 453 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC11 | 1 | 463 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC12 | 1 | 463 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| HC | HC13 | 1 | 463 | 34 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

The next script in the system is the reference file which we use to create the y labels used by every model that is built in the system. As a result of it being the only place that contains the y label information for each of the subjects, it's referenced by several scripts including 'ft_sel_red.py', 'rnn.py', and 'model_predictor.py'. The script itself was a collaboration of several other scripts found within the source data directories for 'NSAA' and '6minwalk-matfiles', including 'KineDMD data updates Feb 2019.xlsx' and 'nsaa_matfiles.xlsx'. Some of these scripts contain information about certain subjects that others don't; hence, rather than having the scripts checking each of the files in turn, it was felt that it would be easier to combine the information from all of them into one file.

Each of the models that are built by the 'rnn.py' script are built to target one output type, which will be either 'dhc' (the D/HC label), 'overall' (the overall NSAA score), 'acts' (the 17 individual activity scores), or 'indiv' (the score of the individual activity assuming the input are single-act files) for each of the sequences the model is training and subsequently assessing on (see the 'Data Forms and Types' chapter for more information on each of these output types). For the 'dhc' output type, we don't need to reference this file, as this can be determined simply by the file name. For example, in preprocessing the data into x and y components within 'rnn.py', if the given data that is currently being preprocessed for a given file is from a file called 'D4_position.csv', then we know that the 'dhc' label would be 'D' (or 1 when being fed into the network), while if the data came from the 'HC6_position.csv' file the label would be 'HC' (or 0).

However, for the other types of y labels that we use to train our 'rnn.py' to target the other output types, it's not as simple as observing the name of the file. This is where the NSAA score reference file comes in. As we can see in the above image, it contains the information for every subject we have (that was collected by the initial assessors of the subjects) that includes their individual activity scores and their overall NSAA score. Hence, by finding the relevant row within the document, we get all the information we need for the other output types. For example, let's say the preprocessing function of 'rnn.py' is extracting data from the file corresponding to subject 'D16'.

Once the data file has been separated into their sequences with necessary sequence overlap and an optional proportion of the sequence dropped (more on these later in the discussion of experiment sets), we then need to get the corresponding label for this sequence. If the model is being trained for the 'overall' output type, then we check the table for the value in the 'D16' subject's 'NSAA' column, which corresponds to '23'. This is the y label that each sequence extracted from the 'D16' source file would get if we are training for the 'overall' output type. Alternatively, if the model output type was instead 'acts', the y label would be a list of values '[2, 2, 1, 2, 2, 1, 1, 1, 1, 2, 0, 2, 1, 2, 1, 1, 1]' or, if the source file was a specific single act file for the subject, e.g. 'D16_position_act4.csv', then it would get the label '2' (both of which are determined by the above table. This process of label extraction is identical across multiple scripts, be it for training models in 'rnn.py' or getting the true values of assessing subjects in 'model.predictor.py', and so on.

7.4 Results for Experiment Sets ('RNN Results.xlsx')

| Experiment Numb | Description | 'comp_stat_vals'/ext_raw_measures' input | 'ft_sel_red' arguments |
|-----------------|--|--|--|
| 1 | Stat values from NSAA/AD to perform D/H/C classification | python comp_stat_vals.py NSAA AD all --split_size=1 | python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize |
| 2 | Stat values from NSAA/AD to perform overall NSAA score regression | python comp_stat_vals.py NSAA AD all --split_size=1 | python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize |
| 3 | Raw joint angles from DC to perform D/H/C classification | python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv | (Not used) |
| 4 | Raw joint angles from DC to perform overall NSAA score regression | python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv | (Not used) |
| 5 | Raw joint angles from JA to perform D/H/C classification | python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv | (Not used) |
| 6 | Raw joint angles from JA to perform overall NSAA score regression | python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv | (Not used) |
| 7 | Stat values from 6minwalk-matfiles/AD to perform D/H/C classification | python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1 | python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize |
| 8 | Stat values from 6minwalk-matfiles/AD to perform overall NSAA score regression | python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1 | python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize |
| 9 | Stat values from NSAA/AD to perform single-acts classification | python comp_stat_vals.py NSAA AD all --split_size=1 | python ft_sel_red.py NSAA AD all pca --num_features=30 --no_normalize |
| 10 | Raw joint angles from DC to perform single-acts classification | python comp_stat_vals.py 6minwalk-matfiles DC all --extract_csv | (Not used) |
| 11 | Raw joint angles from JA to perform single-acts classification | python comp_stat_vals.py 6minwalk-matfiles JA all --extract_csv | (Not used) |
| 12 | Stat values from 6minwalk-matfiles/AD to perform single-acts classification | python comp_stat_vals.py 6minwalk-matfiles AD all --split_size=1 | python ft_sel_red.py 6minwalk-matfiles AD all pca --num_features=30 --no_normalize |

(continued)

| 'rnn' arguments | Results | Settings |
|---|---|---|
| python rnn.py NSAA AD all dhc --seq_len=10 | Test Accuracy = 92.97% | X shape = (742, 10, 30) Y shape = (742,) |
| python rnn.py NSAA AD all overall --seq_len=10 | Mean Squared Error = 28.7121, Mean Absolute Error = 2.9016 | X shape = (742, 10, 30) Y shape = (742,) |
| python rnn.py direct_csv DC all dhc --seq_len=60 | Test Accuracy = 99.88% | X shape = (8470, 60, 66) Y shape = (8470,) |
| python rnn.py direct_csv DC all overall --seq_len=60 | Mean Squared Error = 0.4762, Mean Absolute Error = 0.4037 | X shape = (8470, 60, 66) Y shape = (8470,) |
| python rnn.py direct_csv JA all dhc --seq_len=60 | Test Accuracy = 100.0% | X shape = (2143, 60, 66) Y shape = (2143,) |
| python rnn.py direct_csv JA all overall --seq_len=60 | Mean Squared Error = 0.1675, Mean Absolute Error = 0.2919 | X shape = (2143, 60, 66) Y shape = (2143,) |
| python rnn.py 6minwalk-matfiles AD all dhc --seq_len=10 | Test Accuracy = 92.81% | X shape = (552, 10, 30) Y shape = (552,) |
| python rnn.py 6minwalk-matfiles AD all overall --seq_len=10 | Mean Squared Error = 29.4065, Mean Absolute Error = 3.56 | X shape = (552, 10, 30) Y shape = (552,) |
| python rnn.py NSAA AD all acts --seq_len=10 | Individual Activity Accuracy = 92.92%, All Activities Accuracy = 79.69% | X shape = (742, 10, 30) Y shape = (742, 17) |
| python rnn.py direct_csv DC all acts --seq_len=60 | Individual Activity Accuracy = 99.77%, All Activities Accuracy = 98.5% | X shape = (8470, 60, 66) Y shape = (8470, 17) |
| python rnn.py direct_csv JA all acts --seq_len=60 | Individual Activity Accuracy = 99.97%, All Activities Accuracy = 99.74% | X shape = (2143, 60, 66) Y shape = (2143, 17) |
| python rnn.py 6minwalk-matfiles AD all acts --seq_len=10 | Individual Activity Accuracy = 85.48%, All Activities Accuracy = 73.44% | X shape = (552, 10, 30) Y shape = (552, 17) |

With the two main reference files used by the project covered, we now move onto the first of the two documents containing the results of the various experiment sets and model predictions sets that we run. The first of those, 'RNN Results.xlsx', contains the information of building various models within the 'Experiment Set's section of the results discussion which we cover later on. These include testing

different measurements with which to build models, examining different sequence lengths and sequence overlap proportions, and so on. What separates this document from the 'model_predictions.csv' document that we shall discuss shortly is that 'RNN Results.xlsx' contains the results obtained from just analysing the testing data sets supplied to the 'rnn.py': this is generally 20% of the source data set that is fed into the 'rnn.py' script. Hence, the results contained in each cell of the 'Results' column for this document covers the performance of various model setups on this testing data. In contrast, for most of the 'model_predictions.csv' sets these are the results of feeding in complete files of subjects to be assessed on prebuilt models via the 'model_predictor.py' script. Therefore, 'RNN Results.xlsx' gets its results from the direct console output of building models in 'rnn.py', while 'model_predictions.csv' gets its results from running 'model_predictor.py' on models already built in 'rnn.py' and assessing on complete files.

Another difference from 'model_predictions.csv' is that 'RNN Results.xlsx' has its information manually inserted into the table, as opposed to having it automatically done in 'model_predictions.csv' via the 'DataFrame.writecsv()' method called in 'model_predictor.py'. The reason this is manually done is that, for each model that is created that we wish to write to a row of in 'RNN Results.xlsx', there is additional information that 'rnn.py' has no knowledge about and therefore cannot write to the table. This includes a short description of the model that has been created and what prior scripts were needed to have been run in order to preprocess the data that is required for this model (such as 'comp_stat_vals.py' or 'ext_raw_measures.py'). Therefore, when we wish to write a row of data to 'RNN Results.xlsx', we manually fill in parts of the table ourselves and copy/paste in other parts of the console output (here, into the 'Results' column). Additionally, along with the description, necessary prior scripts with arguments to have been run, and the results of the model, we also output the general model configuration that includes the shape of the data and several of the more significant hyperparameters. The idea is that anyone examining our results in the results discussion section can refer back to this table fairly easily and see exactly how the models were built.

The way these results from 'RNN Results.xlsx' are generally assessed is via the 'graph_creator.py' script. This script reads directly from the file, grabs the requested rows, and plots one or more of the columns against each other depending on the arguments (for example, one configuration of 'graph_creator.py' could plot the results of the MAE of the overall NSAA score against the sequence lengths of the corresponding rows, which we do so in experiment set 8). This is therefore the primary way with which we use the 'RNN Results.xlsx' file in this report, though it also serves the purpose of providing results which can be compared to via other users using the system.

7.5 Results for Model Predictions Sets ('model_predictions.csv')

| | | | | | | | |
|-----|----------------------------|------|-----|---------------------------|--------|---|--|
| N/A | D11 (downsampled) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1] | [2, 2, 1, 2, 2, 2, 2, 1, 1, 1, 2, 0, 2, 1, 1] |
| N/A | D17 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D17 (downsampled) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | HC6 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | HC6 (downsampled) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] | [2, 2, 1, 2, 2, 2, 2, 1, 1, 2, 1, 2, 2, 1, 1] |
| N/A | D3 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 2, 1, 1] |
| N/A | D3 (feature concat = 60) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D9 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [1, 1, 1, 2, 2, 1, 1, 0, 1, 2, 1, 2, 0, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D9 (feature concat = 60) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 1, 1, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D9 (feature concat = all) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 1, 1, 2, 2, 2, 1, 1, 1, 0, 2, 0, 2, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D11 (feature concat = 60) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D11 (feature concat = all) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 1, 1] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D17 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D17 (feature concat = 60) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | D17 (feature concat = all) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | HC6 | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |
| N/A | HC6 (feature concat = 60) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] | [2, 2, 1, 2, 2, 2, 2, 1, 1, 2, 1, 2, 0, 2, 1, 1] |
| N/A | HC6 (feature concat = all) | NSAA | N/A | ['position', 'sensorMat'] | NSAA P | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] | [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] |

(continued)

| | | | | | | |
|--------|----|----|--------|--------|----|----|
| 58.82% | D | D | 100.0% | 0.0% | 27 | 24 |
| 94.12% | D | HC | 50.0% | 50.0% | 33 | 28 |
| 70.59% | D | HC | 50.0% | 50.0% | 33 | 25 |
| 100.0% | HC | HC | 25.0% | 75.0% | 34 | 31 |
| 58.82% | HC | D | 75.0% | 25.0% | 34 | 23 |
| 52.94% | D | D | 91.74% | 8.26% | 19 | 25 |
| 23.53% | D | D | 77.23% | 22.77% | 19 | 28 |
| 41.18% | D | D | 87.86% | 12.05% | 19 | 29 |
| 64.71% | D | D | 100.0% | 0.0% | 22 | 25 |
| 41.18% | D | D | 80.94% | 19.06% | 22 | 28 |
| 41.18% | D | D | 73.75% | 26.25% | 22 | 25 |
| 82.35% | D | D | 94.84% | 5.16% | 27 | 27 |
| 82.35% | D | D | 69.53% | 30.47% | 27 | 26 |
| 70.59% | D | HC | 63.12% | 36.88% | 27 | 27 |
| 94.12% | D | HC | 22.66% | 77.34% | 33 | 27 |
| 94.12% | D | D | 79.69% | 20.31% | 33 | 26 |
| 88.24% | D | D | 81.25% | 18.75% | 33 | 26 |
| 100.0% | HC | HC | 16.8% | 83.2% | 34 | 31 |
| 52.94% | HC | D | 85.94% | 14.06% | 34 | 26 |
| 100.0% | HC | HC | 13.28% | 86.72% | 34 | 32 |

The final document that we make heavy use of in the system is the 'model_predictions.csv' document. This is where any assessments made by the 'model_predictor.py' script is stored (and by extension 'test_altdirs.py' which calls 'model_predictor.py' numerous times) and as such corresponds to the model predictions sets that are discussed later in the 'Experiments and Results Discussion' chapter. As all the information that we wish to write to the file is known by the 'model_predictor.py' at run time, this processed is automated via the script itself and does not require the user to enter information into the table manually. As the primary differences between the two documents have already been outlined, it just remains to outline the types of information that is recorded in the table along with how we make use of this when assessing various model setups and investigating performance of different variations of subject files in the model predictions sets.

Each assessment made by 'model_predictor.py' writes one row of information to this table. This contains the name of the subject we are assessing on, along with extra strings that signify different types of models that subject was assessed on or any other preprocessing steps that were taken for the subject file(s) (see the relevant model prediction set descriptions to see to what each of these additional strings correspond). The name of the directory that the subject file(s) are sourced from is also included, along with any other directories that act as alternative directories from which to source files (see model prediction set 1 for an example of this). We also include the different measurements extracted from this subject's file(s) that were used to act as input to the necessary models (the exact models that are chosen therefore depend on the types of measurement that are extracted from the subject file in question).

The rest of the columns contain the assessed results of the subject for all the output types we are testing the subject on (which is generally the 'overall', 'acts', and 'dhc' output types). For example, the 'Predicted 'D/HC Label"' reflects the results of the assessing the subject on models built for the 'dhc' output type, while the 'True 'Overall NSAA Score"' and 'Predicted 'Overall NSAA Score"' reflects the results of assessing on models built for the 'overall' output type. Therefore, we use all the information from subject assessments on various types of models to create tables comparing different subjects and setups (as shown in many model predictions sets) or to be used via 'graph_creator.py', which can access 'model_predictions.csv' in a similar way to 'RNN Results.xlsx' in order to create graphs over various rows of the table, based on the arguments passed to 'graph_creator.py'.

Chapter 8

Background

Chapter 9

Contribution

Chapter 10

Experimental Results

Chapter 11

Conclusion