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

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 NSAAs 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 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 uses 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 dystrophic, a protein that helps keeps 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."

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 difficult 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

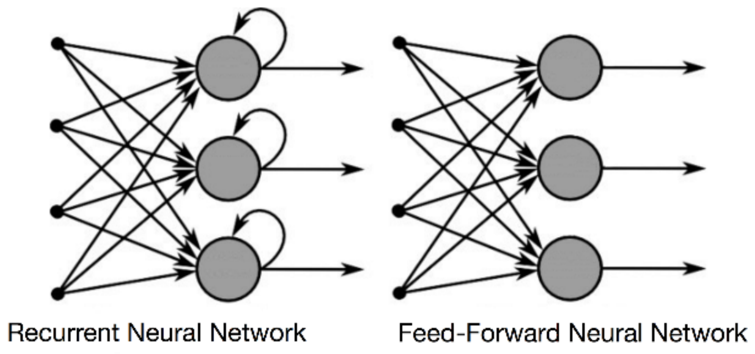
3.1 Outline

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.

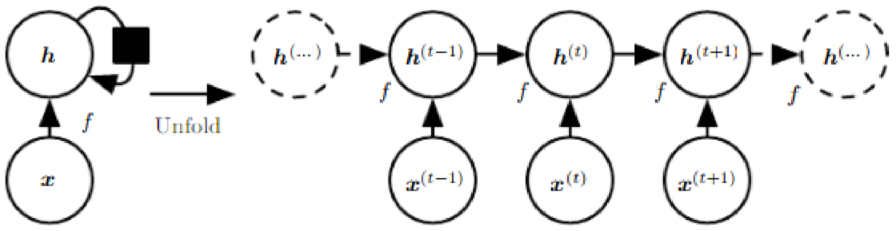
3.2 Motivation, Architecture, and Training

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 its 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:

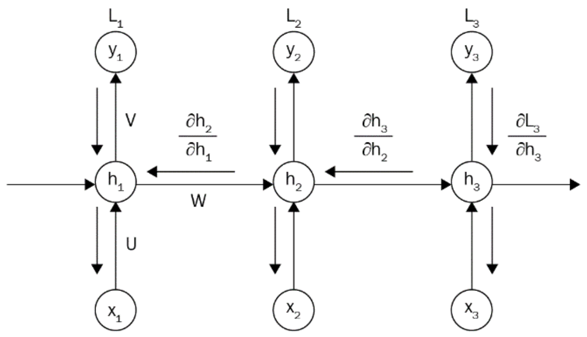
$$h^{(t)} = \varphi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

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:

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

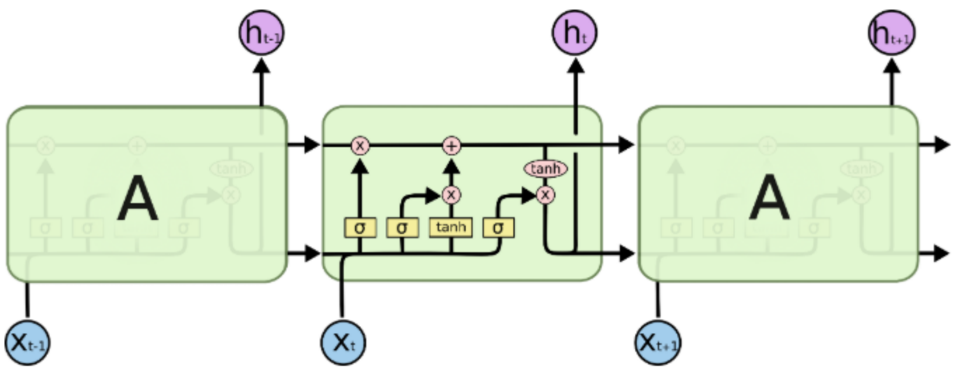
The idea is that the overall loss L is the sum of all the loss functions from times $t = 1$ to $t = T$, and since the loss at time t is dependent on the hidden units at all previous time steps, the gradient is as seen above. Note that its chain rule structure is still very similar to standard backpropagation 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:

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

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:

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

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:

$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

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 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

Part III

System Overview and Explanation

Chapter 4

Background

Chapter 5

Contribution

Chapter 6

Experimental Results

Chapter 7

Conclusion