**Overview of Recurrent Neural Networks**

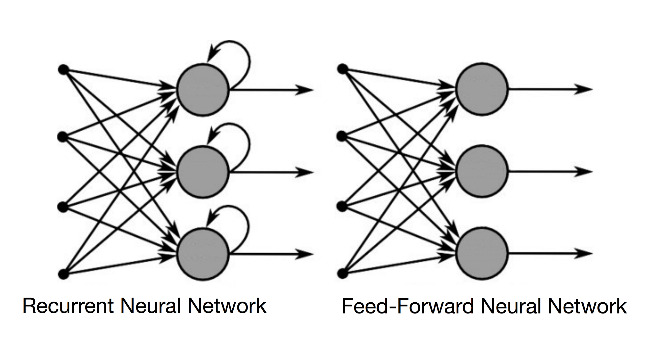
**Outline**

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.

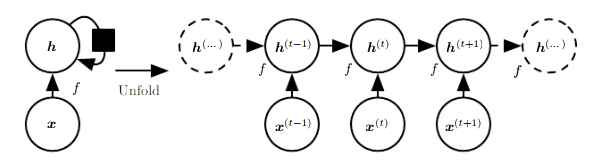
**Motivation, Architecture, and Training**

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.

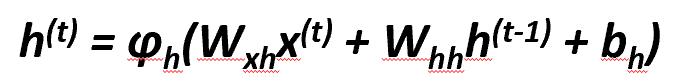
This lack of memory with FFNNs is something that RNNs attempt to fix in the following way: rather than the values of the hidden nodes of the FFNN being only affected by the values that feed into it (e.g. the network inputs or the values from the previous layer), hidden layers in RNNs are also affected 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 ‘*ht*’ is modified by not only the input ‘*x*’ but also the layer’s previous state ‘*ht+1*’, we can essentially ‘unfold’ this dependency with respect to time to get:

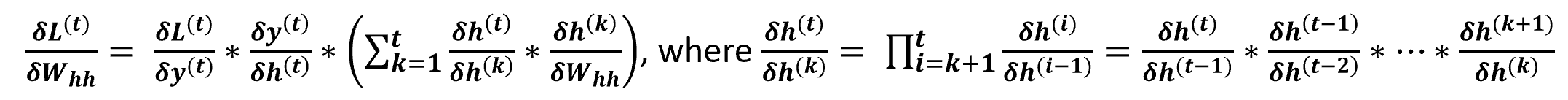


In other words, the output of a hidden node at time ‘*t*’ is given as ‘*ht*’ and is a function ‘*f*’ of the input at this time from the previous layer ‘*xt*’ and the output of the same layer at the previous time step ‘*ht-1*’. This can therefore be seen as the ‘memory’ aspect of an RNN: values from previous 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:

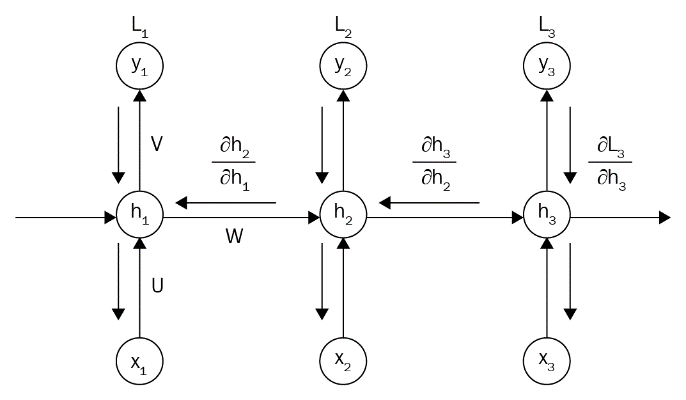


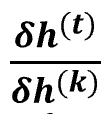
Note that the above function ‘ϕh’ is equivalent to ‘*f*’ in the above diagram, as in both cases they represent the activation function for layer ‘*h*’. We interpret this as the output of a hidden layer at time step ‘t’ being a combination of the previous hidden layer output and the current input to the 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 ‘*Wxh*’ that is learned through training to map from the previous layer’s values to the current one, but there is an additional weight matrix ‘*Whh*’ 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 ‘*Whh*’ matrix controls the influence each left-to-right arrow in the unfolded sequence image has on the state it points to, while the ‘*Wxh*’ 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 ‘*Whh*’ matrix via gradient descent:



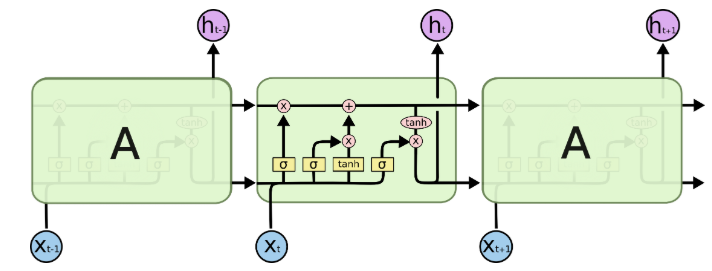
The idea is that the overall loss ‘L’ is the sum of all the loss functions from times ‘t=1’ to ‘t=T’, and since the loss at time ‘t’ is dependent on the hidden units at all previous time steps, the gradient is as seen above. Note that its chain rule structure is still very similar to standard backpropagation 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 ‘*Whh*’. Below, we can also see how these derivative values propagate backwards through time:



However, a large problem arises from the  terms having ‘t-k’ multiplications in the above equation, which therefore multiplies the weight matrix ‘*Whh*’ 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.

**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:

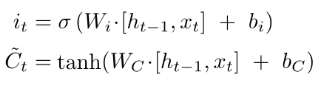


The central idea of this input is the horizontal line running through the top of the unit which allows the data to run along the unit relatively unchanged if required. The gates, represented in the above small yellow boxes within the central green box, allow the unit to let information in (we shall denote 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 ‘*ht-1*’ and ‘*xt*’ 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 ‘*ft*’ is given as:



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:

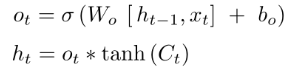


These two are multiplied together to get the new information that we wish to store in the cell, which replaces the old information that we have lost via the ‘forget gate’.

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 ‘*Ct-1*’ into the new cell state ‘*Ct*’. 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:



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 ‘*Ct*’ (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:



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.

**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).



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:



* 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’:

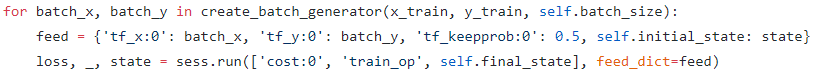


* These steps are defined within the ‘build()’ method for the ‘RNN’ class which is called upon by the constructor of the object at object creation. Hence, when we create an RNN object as…

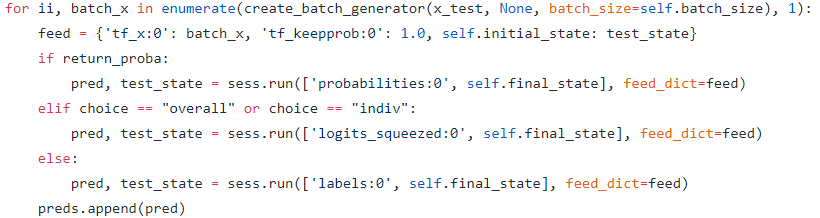


…it results in setting the attributes of the RNN object, including most of the hyperparameters that influence the architecture of the object, and calling the ‘build()’ method of the object that uses many of these hyperparameters. Thus, the above statement sets up the computational graph that defines 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:



* 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:



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.