1. Introduction

Subject investigated.

For Assignment 3 the theme chosen was the analysis of a neural model (Theme 2). The model investigated was the Long-Short-Term Memory Recurrent Neural Network (LSTM-RNN) model.

The focus of analysis was placed on understanding the model, particularly its architecture, rather than optimising the training regime. A rough version of the training algorithm most commonly used in LSTM networks, Back Propagation Through Time (BPTT), was examined. Training a network is potentially a more involved subject than implementing the model itself.

Investigation methodology

Sources used

Investigation began with an examination of available sources on LSTM networks. Information was primarily drawn from sources in the public domain (publicly available papers, Wikipedia articles, online tutorials). Various papers in academic journals were available via the University of Sussex library, but the material accessed was deemed sufficient for the task at hand and the library's resources were not used. A more thorough investigation would certainly require the use of such sources.

Practical (implementation)

Analysis proceeded along with a very simple implementation of an LSTM model in Python (lstm_rnn.py, included in the assignment deliverables). Some of the mathematical tools studied during lectures were also employed, particularly in understanding the structure of the network and the dimensions of its input and output vectors and weight matrices. The techniques employed included linear algebra, particularly matrix multiplication, the rules of which were used in understanding the relation between network inputs, hidden layer vectors and weight matrices, and network activations. Additionally, partial differentiation was employed for the purpose of error minimisation, as in the investigation of training algorithms.

A number of free and open-source development frameworks are available for machine-learning study and experimentation (including some specialising in Neural Networks). Those were found to be too high-level for the purpose of understanding the LSTM model at the fundamental level required for this report. Frameworks such as Matlab's Neural Network Toolbox, Theano [1] and Keras [2] were considered. A concession was made for numpy [3], a library for tensor manipulation and arithmetic included in the scientific programming package scipy [4], given that implementing such functionality from scratch was well outside the purpose of this assignment.

Experiments performed

A number of experiments were performed with the Python implementation described above. Experiments' results were plotted using the R language [5] (R script files are output by the lstm_rnn Python module)[1]. Some test data was created “by hand”, other produced using Swi-Prolog [6] and the library clpfd [7] (a library for Constraint Logic Programming in Finite Domains). Prolog was chosen for the trivial ease with which it produces permutations of lists of elements, both symbolic and numeric.

[1] scipy includes a plotting library, Matplotlib, but I am already familiar with plotting in R.

Results of the experiments, including logs and plots, are included with the assignment deliverables, as is the Python module used to generate each experiment's output and the test data generated or hand-crafted for the assignment. Each experiment's output is included in a separate directory for clarity.

Two sets of experiments were made: one while developing the VS-LSTM implementation, another using its "finalised" version. Experiments in the first set are not always informative of a correct LSTM model, so for the most part they were omitted from this report, however were relevant their results are discussed and plotted.

Motivation of investigation

The choice of subject was informed by a background in computer science and an interest in artificial intelligence, particularly employing AI techniques in software engineering. RNN networks can be seen as a model of computation and LSTM networks have been reported as being Turing complete [8]. It is plausible that such models may be used for automatic programming.

2. Subject matter

Recurrent Neural Networks in brief

The term "Recurrent Neural Network" (RNN) refers to a type of neural network where a hidden layer with a nonlinear activation has a connection to itself (see diagram, opposite). This recurrent layer stores an internal state of the network across several time-steps of the training process, therefore acting as the network's "memory". It is this recurrent connection that makes RNN interesting from a computation-theoretical pint of view, because it enables the model to store and alter its own state, as would a program. It also enables the network to identify relations between inputs encountered at different time steps, such as the dependencies between parts of speech in a sentence.

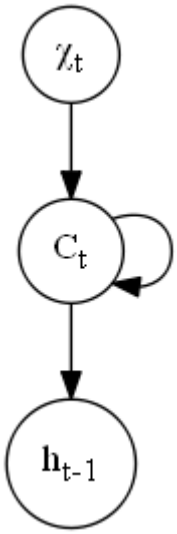

FIGURE 1: RNN

RNN are typically trained using sequences of input vectors, where each vector represents the state of the input signal at a given time step (the time step that corresponds to the index of that vector in the sequence). Accordingly, the activation of each output unit in the trained network represents a discrete time step of the output signal. This enables RNN to learn from dynamic data that changes over time.

It is possible to train RNN to map an input sequence of length N to an output sequence of length M, where N is not necessarily the same as M [9]. For instance, a sequence of binary digits could be mapped to a decimal digit, or a sequence of words in a sentence could be mapped to a class of sentiment, or to a sentence in a different language with more or fewer words than the original.

These characteristics of RNN, to learn a mapping between arbitrary sequences and to keep an internal state between different time steps makes them seem appealing for tasks such as speech recognition, Natural Language Processing and unsegmented handwritten character recognition.

Additionally, the ability to output the next element in a sequence naturally suggests the ability to generate sequences. RNN have indeed been used for example to generate language [10].

RNN Limitations

RNN are trained using a version of the back-propagation algorithm adapted to their self-connected configuration and called Back-Propagation Through Time (BPTT). This is similar to "static" back-propagation, except that the input signal is propagated forward through the network along its

recurrent connection to itself over successive time-steps (known as "unfolding" or "unrolling" the network). The unfolded network can be considered a deep, feed-forward network, with the first self-connected instance as the input layer, the last instance as an output layer and each intermediary instance as a hidden layer. Seen like this, the unfolded network can be trained as a feed-forward network, by propagating error back over the self-recurrent connections, calculating the error with respect to each time-step and adjusting connection weights accordingly. Typically, every training iteration the network is presented with the whole input sequence up to the current time step; in this scheme an "epoch" passes when the network has seen an entire input sequence.

An issue that arises with this training scheme (which also affects deep feed-forward networks) is that, because the error is scaled relative to the derivative of each layer's activation, the more the layers the error travels through during back-propagation, the larger the scaling effect. If the derivative of the activation function is small, as is the case with sigmoid functions, the error will tend to vanish, slowing the learning process to the point where it becomes impractical to train the network. Conversely, if the derivative of the activation is high, the error will grow exponentially, eventually also becoming unmanageable. This is known as the "vanishing gradient" problem (or exploding gradient, when the gradient increases uncontrollably).

This issue is serious enough that, historically, RNN have proved incapable of learning sequences with long-term dependencies between their elements [11].

The LSTM RNN architecture

The Long-Short-Term Memory architecture was proposed as a solution to the vanishing gradient problem by Sepp Hochreiter and Jurgen Schmidhuber in papers going as far back as 1995, for example [8].

Central to this solution is a setup that the referenced paper calls the Constant Error Carrousel (CEC). This is a recurrent connection, as in non-LSTM RNN, but in this case the internal state has a linear activation and a constant weight of 1.0, therefore keeping the error gradient constant (and equal to 1). This would seem to also halt learning, even more comprehensively than a vanishing gradient, but the LSTM architecture also involves a set of "gate" units, additional layers with nonlinear activations that are permitted to access and modify the internal state layer. Essentially, rather than training its internal state layer directly an LSTM network learns patterns of activations of its gate layers that result in a hidden layer that can best represent the signal to be learned. In other words, the gates learn to open and close access to the memory, "protecting" or "exposing" it, as necessary to optimise learning.

The CEC and its attendant gates

The memory layer and the inputs of its attendant gates are each vectors of size equal to the size of the elements of the network's input sequence.

LSTM gates are neural units with nonlinear activation functions, selected to "squash" their input to within a small range such as (0,1). A typical example is the logistic function, but others are used, for instance tanh, ReLu, and so on. At each time-step, the output of each gate is put through a pointwise multiplication or summation with the contents of the network's internal state vector. The result is of masking or filtering the state vector, allowing only those values that are significantly larger than 0 to be output by the network's activation. Given that the hidden state layer has a recurrent connection to itself, this effect also extends across time steps, regulating the internal state for subsequent activations.

Different configurations of gates and connections to the memory layer are possible, however the scheme most commonly used involves three gates, each modulating the input signal, the network's overall activation and the hidden state layer. These are known by various names in the bibliography (usually "input", "output" and "forget" gate respectively). To avoid confusion with the networks overall activation and inputs, we will refer to them as the "remember", "recall" and "forget" gate respectively.

The gate and memory configuration outlined here are further elaborated in the immediately following section.

A Very Simple LSTM architecture

Owing to the complexity of, but also the many papers published on, the subject by different authors, it is difficult to find a simple example of an LSTM network to study the architecture at an elementary level.

As discussed in an earlier section, a simplified LSTM model was implemented for this report, in an attempt to overcome this difficulty and isolate the essential elements of the architecture for the purose of studying it. It should be noted that this implementation was optimised for simplicity and clarity, rather than performance and completeness- for instance, the popular addition of "peephole" connections, reported to greatly increase performance [12] was omitted, although these are described and shown in the following diagram. That said, VS-LSTM is not the simplest LSTM architecture possible; the original Hochreiter and Schmidhuber paper [11] did not include a "forget" gate and a popular variant, the Gated Recurrent Unit network (GRU) combines the forget and input gates into an "update" gate and also combines the memory layer with the network's activation [13], for an overall simpler architecture. Despite this setup being simpler, it represents a more advanced model than the "standard" LSTM model and would be difficult to justify without a description of the standard model.

A diagram of the VS-LSTM model is shown below.

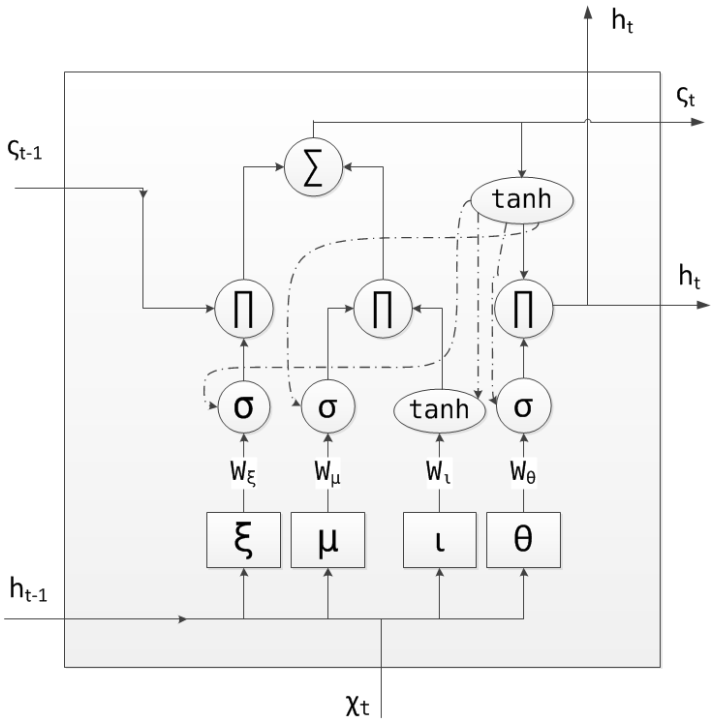

FIGURE 2: VS-LSTM CELL

Summary of notation and nomenclature

The diagram depicts the basic computational unit of the VS-LSTM architecture (roughly analogous to a flip-flop unit in a computer processor). We have referred to the LSTM architecture as a "network" above, while speaking of the model in general terms, however from now on the type of unit depicted in the above diagram will be referred to as one LTSM "memory cell".

Each memory cell connects a memory layer and its attendant gates. Several memory cells may be connected together to form the hidden layer of an LSTM network. An LSTM network therefore will typically consist of an input layer, an output layer and a hidden layer of connected LSTM cells. This hidden layer may be referred to as a memory "block". A connected block is shown in the diagram below.

LSTM cells extend connections to others in their block through two "ports", one for the cell's activation and one for the cell's memory, therefore allowing cells in a block to share their internal state, and also to combine their output in a single element of the output sequence. This enables the transformation of input sequences to outputs of possibly different size.

It should be noted that when "unfolding" a cell during training, the same connection scheme is employed except of course that the cell's memory is connected to itself in the next timestep. In the above diagram of the VS-LSTM model two output connections are shown leaving the network, depicting its ability to connect its output to a subsequent unit, or output it as part of the network's overall activation.

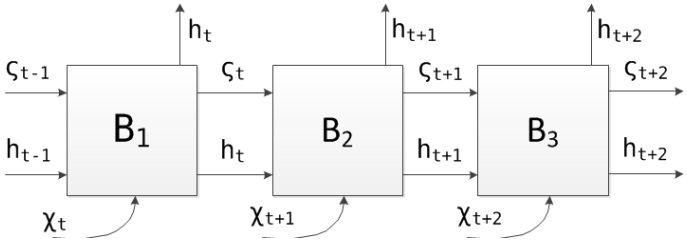

FIGURE 3: VS-LSTM BLOCK

In the above diagram of the VS-LSTM model, ι is the input layer, ξ, μ, θ are its attendant gates: the "forget" gate, "remember" gate [usually noted as the input gate in LSTM literature] and "recall" gate [usually output gate]; nodes annotated with a Greek σ (sigma) are non-linear layers with a logistic function activation in VS-LSTM; the two layers annotated with tanh are layers with a hyperbolic tangent activation; nodes marked with ∑ and ∏ denote point-wise operations on vectors; edges annotated with Wς, Wξ etc are weigthed connections of their respective layers; h□ is the block's activation layer (its output) at time-step t; ς□ is the output of the memory layer at time-step t; and χ□ is the cell's input at time-step t. The above notation and nomenclature will be used henceforth in this report.

Not shown in the diagram are four bias vectors of size equal to the cell's activation, one for each gate and one for the input node. Those are passed to each unit's activation function (this being the tanh node for the cell activation), as usual.

The activations of each layer are calculated as follows:

Each of the forget, remember and recall gate and the input layer calculate their activation by multiplying their input vector by their weight matrix and passing the result through their respective squashing function (logistic sigmoid for the gates, tanh for the input layer). If there is a previous layer's activation to take into account, this is included in the calculation by concatenating the activation and its weights with the corresponding input and weights of the activating layer:

$$\xi_t = \sigma(cons(W_{\xi_t} \cdot x_t \, , h_{t-1} \cdot W_{h_{t-1}}))$$

$$\mu_t = \sigma(cons(W\mu_t \cdot x_t\,, h_{t-1} \cdot W_{h_{t-1}}))$$

$$\iota_t = \tanh(cons(W\iota_t \cdot x_t\,, h_{t-1} \cdot W_{h_{t-1}}))$$

$$\theta_t = \sigma(cons(W_{\theta_t} \cdot x_t\,, h_{t-1} \cdot W_{h_{t-1}}))$$

The previous layer's memory is then put through a series of element-wise operations with each of the three gates' output to calculate the cell's activation and the value of the new memory layer. The cell's activation is first passed through its squashing function (tanh) to introduce nonlinearity and normalise its range:

$$\varsigma_t = (\xi_t * \varsigma_{t-1}) + (\mu_t * \iota_t)$$

$$h_t = \tanh(\varsigma_t)$$

Shown in the diagram above [DISAMBIGUATE] are "peephole" connections from the cell's memory to its attendant gates. In LSTM configurations where the memory layer projects connections to its attendant gates in subsequent layers, these connections will be modulated by the recall gate. When this is "closed", gates will receive no input over their connections from the memory, keeping them from modulating the memory. To overcome this limitation, "peepholes" are employed: weighted connections from the memory to its attendant gates allowing gates to "peek" at the memory vector's ungated values. Our VS-LSTM architecture does not include direct connections between memory and gates, therefore peepholes are not immediately relevant, however they seem popular enough that they were added in the diagram for completeness.

As discussed earlier the innovation of the LSTM architecture is the cell's memory unit (dubbed the Constant Error Carrousel). This recurrent unit allows the circulation of a constant error gradient through a memory block, which enables it to "remember" information learned many time-steps before the current- or, conversely, to let it dissipate if it's no longer relevant [this long-term retention ability gives the LSTM architecture its ability to learn to reproduce long sequences]. In the VS-LSTM implementation, the cell's memory is initialised to a vector of 1.0 values.

The nomenclature adopted in this report, of a "forget", "remember" and "recall" gate alludes to the modulation of the input signal and memory vector by the block's three gates in terms relating their function to the experience of human memory. When the cell's memory vector is multiplied to the activation of the forget gate, some of its elements may be reduced to 0, thus effectively being "forgotten". Accordingly, multiplying the input by the "remember" gate activation selects some elements as candidates for addition to the cell memory, therefore making a decision to "remember" them (or not); finally, multiplying the block's activation with the recall gate's activation decides whether the block will contribute its activation to the rest of the network's activation, effectively retrieving the memory from storage.

3. Investigation

A complication that arises from the investigation of a complex subject with practical tools (and also with theoretical ones) is that it is hard to know at any time during the investigation whether one is on firm ground or completely off course. In the context of this investigation of LSTM using the simple Python implementation detailed above, when faced with counter-intuitive observations, it was difficult to know whether to attribute them to the nature of the model, or to the implementation. Even when observations matched expectations it was difficult to tell why.

Nevertheless, a number of experiments were performed and their results are presented here. However they should be treated with the understanding that conclusions are speculative, rather than authoritary. In general, experiments but also the report as a whole, generated more questions that it answered. These will have to be investigated further, in my own time and during the rest of the course. Below are some of the experiments attempted and my speculative analysis of the results.

Testing rig

Experiments were executed using a test module, experiment_n.py. This serves both as a record of experiments' settings and also to allow each experiment to be reproduced easily. Refer to the module's comments for instructions to reproduce experiments.

Experiments attempted

Experiment 2: Question 1[2]

This experiment looks at question 1 from the assignment specification, assuming a single neuron with no connections from a previous layer, and with constant input.

In the context of the LSTM model, a single neuron is an entire cell; a 0-connection to the previous layer means that there is no previous input, nor previous memory to concatenate with the current time step's input; and a constant sensory input means that there is a single element of the training sequence presented to the network throughout the experiment. A concession must be made to the fact that unfolding the network to allow training must necessarily activate the recurrent connection.

The network is trained on a sequence of bit-vectors of four bits each, representing the binary numerals for the numbers 1 and 2. The target of each training step is the next number in the sequence. In other words, the network has to learn to alternate two binary numbers.

In this and subsequent experiments training on binary numerals represented as bit-vectors, the output of the network is interpreted as a vector of (unnormalised) probabilities that a corresponding element in the target vector is "high" (ie, 1). The maximum element of the activation vector is taken to mark the position of a high digit in the target vector. In a slight variation of the more typical "one-of-k" encoding scheme, we allow more than one digit to be high, as long as it is equal to the maximum value in the activation vector. This allows the network to learn a vector of all-high digits and makes for a more compact representation than one-of-k encoding, but also possibly increases the difficulty of the learning task.

Experiment 3: Question 2

Question 2 asks to vary the network's parameters, particularly its learning rate and time constants and also its input, and see what happens. LSTM RNN do not have a time constant, as CT RNN do. The experiment performed instead varied the learning rate, starting at 0.01 and increasing by .10 every Era for 8 Eras (an Era lasting for 80 epochs).

Training was done against a sequence of 4-bit vectors representing the binary numerals from 1 to 9, looping back to 1 at 9. Encoding was the same as in Experiment 2.

Experiment 5: Question 6

Question 6 asks to connect two neurons together to investigate the stability of the system. In a non-LSTM network, we would expect some instability from connecting multiple layers together, because

[2] Some experiments were dropped between editing this document; numbering is consistent but two numbers are skipped.

of the vanishing gradients phenomenon. Vanishing gradients should not be the case in VS-LSTM, but exploding gradients are still a possibility.

The binary counting task on the dataset from Experiment 3 was also used here.

Experiment 6: Counting in binary

In this experiment we train the network to recognise the next element in a sequence of 9 binary numerals represented as 4-bit vectors. Each input is a vector of 4 bits. The target is the next number in the sequence. The sequence starts at 0001 and "loops back" at 1001

Experiment 7: Learning the Reber grammar

The Reber grammar is an artificial grammar popular in grammar induction and recurrent networks [REFERENCE-1]. Its lexicon consists of strings of arbitrary length composed of the characters: B,T,S,X,P,V,E. Each is mapped to a bit-vector of length seven. This time, encoding follows a typical "one-of-k" (where k=7) encoding scheme, outlined below:

B	1000000
T	0100000
S	0010000
X	0001000
P	0000100
V	0000010
E	0000001

The network is trained on sequences of such bit vectors. The target is always the next element in the sequence.

Experiment 8: Learning the embedded Reber grammar

The embedded Reber grammar allows two types of strings: one that begins with the characters BT and ends in the characters TE, and one that begins with BP and ends in PE. Both paths allow a Reber string of an arbitrary number of characters between the start and end characters. The restriction on the beginning and end characters of the grammar creates a long-term dependency that cannot be learned by RNN (or indeed, feed-forward networks). LSTM should be able to learn it however.

The purpose of training here is to produce a network that, given a Reber grammar string beginning with, for instance, "BT" and following with an arbitrary Reber string, will correctly finish the sequence with "TE".

Conclusions

What was learned - Insights.

Overall, two major insights were gained from this investigation, both of which are insights about Artificial Neural Networks as a whole, rather than LSTM in particular. One insight is that ANN are "just" optimisation, only involving a system of functions rather than a single function.

The other insight concerns the observation that the size of a layer's weight matrix is the product of the sizes of the layer's inputs and outputs:

$$|W| = |X| \cdot |Y|$$

And this relation holds for any type of inputs and outputs, including scalars and vectors. For instance, with LSTM networks the input to the network is a sequence of vectors, therefore the size of weight matrices is given by the above formula, with X,Y as vectors.

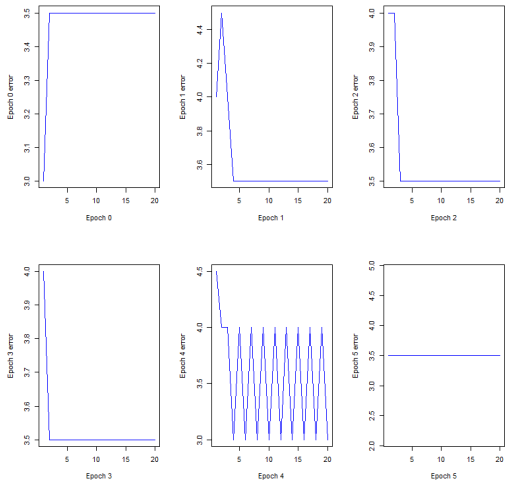

FIGURE 4: PATTERNS OBSREVED

In addition to that, it's possible to perform transformations between inputs and outputs, by taking the above relation into account. This might have uses in dimensionality reduction, for instance.

Neither of these observations is likely to be news to an experienced ANN researcher, of course and they are probably obvious with hindsight. However, for this report they had to be learned the "hard" way, particularly the second insight which was gained after long and arduous fiddling with weight and input and output configurations.

The first insight, concerning the "just optimisation" nature of ANN was reached after observing the plotting of experiments' results. In such figures, four patterns were routinely observed:

a) An oscillation
b) A flat line
c) Exponential decay, occasionally with a distinct growth spike
d) Exponential growth, occasionally with a sudden dip

All these are patterns reminiscent of the behaviour of gradient descent and hill-climbing, observed during our previous assignments. Indeed, training of the VS-LSTM model, using a (crude) version of BPTT uses gradient descent to optimise a measure of error of the network's activation against a target value. It is therefore probably safe to carry over the insights gained from that earlier assignment to the current subject matter.

According to these earlier insights then, the four patterns observed can be explained by the deterministic nature of Gradient Descent:

An oscillation is a sign that the algorithm is "stuck" on a local optimum, which in this case is a minimum (since we are minimising error). Learning cannot improve from this position, because the deterministic algorithm cannot free itself from the local optimum trap.

A flat line is a sign that the algorithm is stuck on a 0-gradient area of the error function- a flat (multidimensional) plane from which its performance can neither improve nor deteriorate. Learning will also cease; the network has "nowhere to go" and can't find anything to learn.

Exponential decay means that the network has successfully identified a gradient that leads to a low point in the error function- this is usually accompanied by a good learning outcome, though not always (if randomly initialised weights and bias values happen to be in a very high error elevation, even exponential decay may not reach the bottom of the error's function in any reasonable time).

Exponential growth is the odd one out. This is not something that I observed in the Gradient Descent/Hill Climb assingment. The gradient descent algorithm implemented for VS-LSTM (as for the earlier assignemnt) does not allow for a movement to a higher elevation.

I have still not resolved this at the time of submitting this assignment, but my current intuition is that it must be an example of the "exploding gradients" issue. Indeed, looking at the logs of one experiment that displayed exponential growth, I noticed that the absolute values of the negative-valued weights and bias terms were extremely high (in the thousands) while the values of layer activations were very small (showing as negative zeroes in the log, configured to display no more than two decimal points, to improve legibility).

```
Time step 12 ======================
    Stimulus:           [0, 0, 0, 0, 0, 0, 1]
    remember_decision: [-0. -0. -0. -0. -0. -0. -0.]
    self.memory         [-0. -0. -0. -0. -0. -0. -0.]
    forget_decision:    [-0. -0. -0. -0. -0. -0. -0.]
    self.memory         [-0. -0. -0. -0. -0. -0. -0.]
    tanh(self.memory)   [-0. -0. -0. -0. -0. -0. -0.]
    recall_decision:    [-0. -0. -0. -0. -0. -0. -0.]
Cell weights ======================
Wi:     [[-1231.29 -2389.99     1.23     1.14 -1821.92 -1248.54 -1234.22]
Wm:     [[-1232.06 -2391.83    -0.95     1.5  -1824.05 -1248.48 -1234.49]
Wf:     [[-1232.01 -2390.48     0.55    -0.17 -1821.68 -1249.34 -1235.02]
Wr:     [[-1231.16 -2392.09     0.38    -0.99 -1823.28 -1251.12 -1234.05]
Cell bias terms ======================
bi:     [-1233.25 -2391.49    -0.38    -0.8  -1821.89 -1250.25 -1235.22]
bm:     [-1231.54 -2392.8     -0.61    -1.72 -1822.43 -1249.25 -1233.76]
bf:     [-1231.82 -2392.92    -0.13    -1.02 -1822.83 -1249.45 -1235.2 ]
br:     [-1232.53 -2389.44    -1.09     0.95 -1822.23 -1250.95 -1234.78]
Prediction:  [1, 1, 1, 1, 1, 1, 1]
Actual:      [0, 0, 0, 0, 0, 0, 0]
Step Error:  1.0
Epoch error: 11.2857142857
```

FIGURE 5: NEGATIVE WEIGHT SATURATION

Note: log is redacted beyond the first row of weight matrices.

At the same time, the overall activation of the cell was abnormally high, displaying all-1s:

```
   Activation for [0, 0, 0, 0, 1, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [1, 0, 0, 0, 0, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 1, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 1, 0, 0, 0, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 1, 0, 0, 0, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 1, 0, 0, 0, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 1, 0, 0, 0, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 0, 1, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 0, 1, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 0, 0, 1]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 1, 0, 0]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 0, 0, 1]: [1, 1, 1, 1, 1, 1, 1]
   Activation for [0, 0, 0, 0, 0, 0, 0]: [0, 0, 0, 1, 0, 0, 0]
```

FIGURE 6: ACTIVATION

This could only happen if all elements of each activation vector were the same value. In turn, for this to happen, network weights must have reached absolute negative saturation. It seems the network was able to find the maximum of the error function, if nothing else (this was also confirmed by the plot of the error in that experiment).

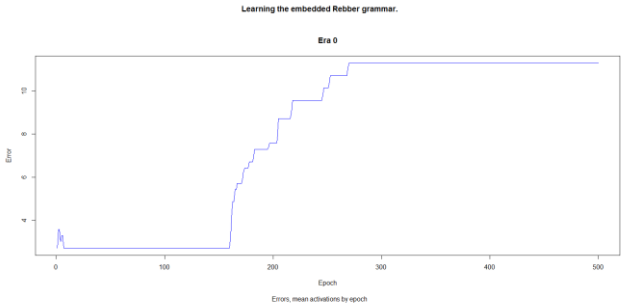

FIGURE 7:POSSIBLY EXPLODING GRADIENT

This may be a result of an error in programming, or design of the VS-LSTM model. Alternatively, it is the result of error gradients becoming very high, causing weight and bias values to become very low and in turn activation values also becoming very low.

This is not implausible, after all exploding gradients are reported as a problem that still affects LSTM networks, unlike vanishing gradients. Indeed, the bibliography suggests that clipping of gradients is useful to keep error down and avoid this situation.

However, I find this confusing, because my understanding so far is that the use of the tanh function to "squash" gate activations to a range of [-1,1] would also result in an error gradient similarly squashed to a small range, therefore keeping it from exploding. I may have misunderstood this.

What was learned - general conclusions

It seems that overall, even given a crude implementation of BPTT and other defects, it seems that the network did manage to minimise its error function, at least some of the time. On the other hand, training times were not trivial, even for the trivial problems examined. It seems then that training LSTM RNN is hard, if good result are required.

What was not learned - open questions

Repeated allusion was made in this report to the fact that the implementation of the BPTT algorithm in VS-LSTM is "crude". In fact it is one of several instances of either mistakes that were not realised in time and therefore remain unresolved in the source code delivered, or uncertainty in the configuration of the LSTM architecture that remains after the brief examination of the publicly available bibliography for this report.

These include:

a) The aforementioned implementation of BPTT. Specifically, the current implementation feeds the entire training sequence to the unfolded network at once. My understanding of BPTT is that, at each time step, the sequence so-far must be presented to the unfolded network, which in fact is unfolded a number of steps equal to the size of the sequence so-far.

b) The implementation of network unfolding and stacking is incorrect in that the activation of the previous layer is only fed forward to the input node at each time step. The correct configuration is for

the previous activation vector to be concatenated to the input node's input but also to each of the three gates.

c) Investigation of different architecture configurations, for instance different activation functions, "peephole" connections, more connections between the various layers and gates and so on.

These open questions will have to explored further in the future.

Bibliography

[1] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard and Y. Bengio, “Theano: new features and speed improvements,” in *Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop*, 2012.

[2] F. Chollet, “Keras,” [Online]. Available: http://keras.io/. [Accessed 24 December 2015].

[3] Anon, “Numpy.org,” [Online]. Available: http://www.numpy.org/. [Accessed 24 December 2015].

[4] Anon, “Scipy.org,” [Online]. Available: http://www.scipy.org/. [Accessed 24 December 2015].

[5] R. C. Team, “R: A language and environment for statistical computing,” R Foundation for Statistical Computing., Vienna, Austria, 2014.

[6] J. Wielemaker, T. Schrijvers, M. Triska and T. Lager, “SWI-Prolog,” *Theory and Practice of Logic Programming,* vol. 12, no. 1-2, pp. 67-96, 2012.

[7] M. Triska, “The Finite Domain Constraint Solver of SWI-Prolog,” in *Eleventh International Symposium on Functional and Logic Programming (FLOPS)*, 2012.

[8] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation,* vol. 9, no. 8, pp. 1735-1780, 1997.

[9] A. Karpathy, “The Unreasonable Effectiveness of Recurrent Neural Networks,” 21 May 2015. [Online]. Available: https://github.com/karpathy/char-rnn. [Accessed 24 December 2015].

[10] A. Graves, “Generating Sequences With Recurrent Neural Networks,” Department of Computer Science University of Toronto, Toronto.

[11] S. Hochreiter, Y. Bengio, P. Frasconi and J. Schmidhuber, “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,” 2001.

[12] D. D. Monner and J. A. Reggia, “A generalized LSTM-like training algorithm for second-order recurrent neural networks.,” *Neural Networks,* vol. 25, no. 1, pp. 70-83, 2012.

[13] C. Olah, “Understanding LSTM Networks,” [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed 24 December 2015].

[14] A. Graves, G. Wayne and I. Danhielka, “Neural Turing Machines,” 2014.

[15] C. Herta, “LSTM (Long Short Term Memory),” [Online]. Available: http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/LSTM.php. [Accessed 24 December 2015].

[16] U. S. o. C. S. Fred Cummins, “Rebber grammar,” [Online]. Available: http://cogsci.ucd.ie/Connectionism/Exercises/Exercise3.php. [Accessed 24 December 2015].

[17] W. U. Unknown, “Reber grammar and embedded Reber grammar,” [Online]. Available: https://www.willamette.edu/~gorr/classes/cs449/reber.html. [Accessed 24 December 2015].

[18] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio, “Theano: a CPU and GPU Math Expression Compiler,” in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Austin, TX, 2010.