THESIS TITLE

MSc Thesis *(Afstudeerscriptie)*

written by

Krsto Proroković
(born March 12, 1993 in Kotor, Montenegro)

under the supervision of **Dr Germán Kruszewski** and **Dr Elia Bruni**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
Date of the defense goes here	Committee goes here

Institute for Logic, Language and Computation

Abstract

Abstract goes here

Acknowledgents

Contents

Chapter 1

Introduction

Chapter 2

Related Work

Learning to decide a formal language can be seen as an example of learning an algorithm from input/output examples. There are several approaches to this problem. One would be to induce a discrete program from finite set of instructions. Inductive logic programming [11] requires domain specific knowledge about the programming languages and hand-crafted heuristics to speed up the underlying combinatorial search. Levin [8] devised an asymptotically optimal method for inverting functions on given output, albeit with a large constant factor. Hutter [5] reduced the constant factor to less than 5 at the expense of introducing a large additive constant. These methods are not incremental and do not make use of machine learning. Schmidhuber [13] extended the principles of Levin's and Hutter's method and developed an asymptotically optimal, incremental method that learns from experience. However, the constants involved in the method are still large which limits its practical use.

The approach that we will take over here is based on training differentiable neural networks. To the best of our knowledge no one used neural networks for deciding a whole class of formal languages, rather a single language from positive and negative examples. Mozer and Das [10] used neural networks with external stack memory to parse simple context-free languages such as a^nb^n and parenthesis balancing from both positive and negative examples. Wiles and Elman [15] used neural networks to learn sequences of the form a^nb^n and generalise on a limited range of n. Joulin and Mikolov [6] used recurrent neural networks augmented with a trainable memory to predict the next symbol of the element of context-free language. Lastly, Zaremba and Sutskever [16] used recurrent neural networks for learning to execute simple Python programs.

Chapter 3

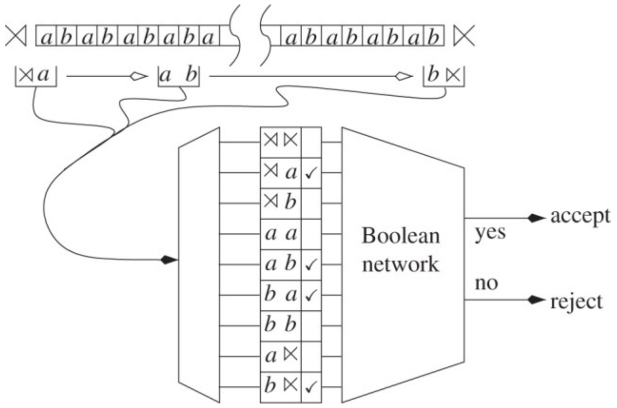
Background

In this chapter we ...

3.1 Locally k-testable languages

Fix an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. An SL_k definition is just a set of blocks of k adjacent symbols (called k-factors) drawn from the alphabet. A string satisfies the description if and only if every k-factor that occurs in the string is licensed by the definition. We can see k-factors as atomic properties of strings: a string satisfies a k-factor if and only if that factor occurs somewhere in the string. Then, we can build descriptions as propositional formulas over these atoms. We will call these formulas k-expressions. A k-expression defines the set of all strings that satisfy it. A language that is defined in this way is called a locally k-testable (LT_k) language.

A scanner for an LT_k language contains a table in which it records, for every k-factor over the alphabet, whether or not that k-factor has occurred somewhere in the string. It then feeds this information into a Boolean network which implements some k-expression. When the end of the string is reached, the automaton accepts or rejects the string depending on the output of the network.



3.2 Recurrent Neural Networks

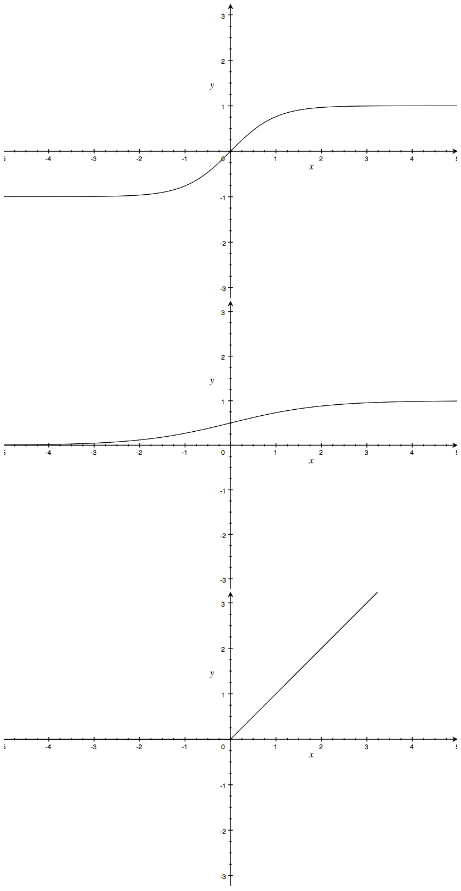
Recurrent neural networks (RNNs) [12] are parametric models of computation for processing sequences loosely inspired by biological neural networks. They found applications in handwriting recognition [reference], speech recognition [reference], machine translation [reference], etc. In this section we provide an introduction to RNNs and ... For a more detailed treatment we point the reader to [3].

3.2.1 Vanilla Recurrent Neural Networks

We start with a simple vanilla RNN model. A vanilla RNN is given with the following parameters:

- Input to hidden connection weights $\mathbf{W}^{\mathbf{x}} \in \mathbb{R}^{d_h \times d_x}$
- Hidden to hidden connection weights $\mathbf{W}^{\mathbf{h}} \in \mathbb{R}^{d_h \times d_h}$
- Bias term $\mathbf{b}^{\mathbf{h}} \in \mathbb{R}^{d_h}$
- Activation function $\phi : \mathbb{R} \to \mathbb{R}$
- Hidden to output connection weights $W_y \in \mathbb{R}^{d_y \times d_h}$
- Bias term $\mathbf{b}^{\mathbf{y}} \in \mathbb{R}^{d_y}$
- Initial hidden state $h^{(0)} \in \mathbb{R}^{d_h}$ (usually set to zero vector)

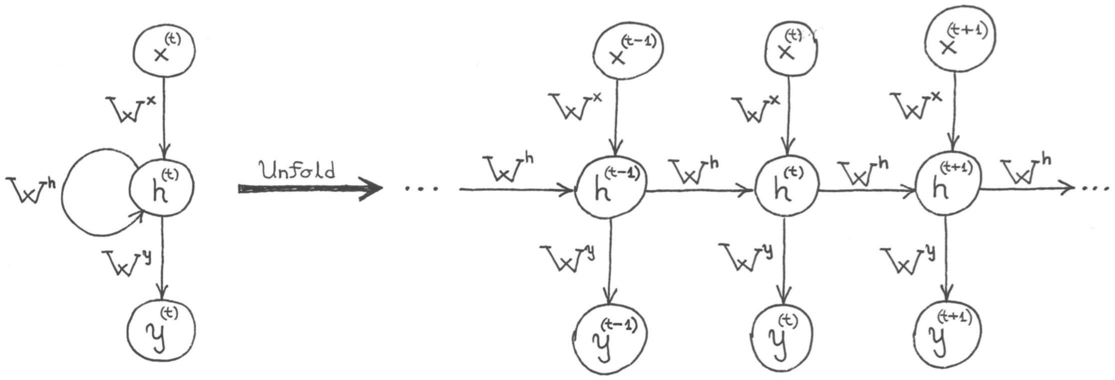
Connection weights model the strength of synapses and activation function models neuronal firing [9]. Most commonly used activation functions are sigmoid $\sigma(x) = 1/(1 + \exp(-x))$, hyperbolic tangent $\tanh(x) = (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x))$ and rectified linear unit $\mathrm{ReLU}(x) = \max\{0, x\}$.



The computation is performed in the following way: for $t = 1, 2, \ldots, T$

$$\mathbf{h}^{(t)} = \boldsymbol{\phi}\left(\mathbf{W}^{\mathbf{h}}\,\mathbf{h}^{(t-1)} + \mathbf{W}^{\mathbf{x}}\,\mathbf{x}^{(t)} + \mathbf{b}^{\mathbf{h}}\right)$$
$$\mathbf{y}^{(t)} = \mathrm{softmax}\left(\mathbf{W}^{\mathbf{y}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathbf{y}}\right)$$

where $\boldsymbol{\phi}(v)_i = \phi(v_i)$ and $\mathrm{softmax}(v)_i = \exp(v_i)/\sum_j \exp(v_j)$.



Note that components of $y^{(t)}$ sum up to one, so we can interpret it as a probability distribution. Here we will be mostly interested in sequence classification, i.e. we will only care about $y^{(T)}$.

Example Consider a vanilla RNN given with the following parameters

$$W_x = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad W_h = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad b_h = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad W_y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad b_y = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and $\phi(z) = z$. It is easy to see that $y_1^{(T)} < y_2^{(T)}$ if and only if $\sum_t (3x_1^{(t)} - x_2^{(t)}) < \sum_t 2x_3^{(t)}$. Note that in this case $h^{(t)}$ accumulates the results of the computation on the input encountered so far.

While the previous example displays a rather simple function, vanilla RNNs are universal in the sense that any function computable by a Turing machine can be computed by such a network [reference Siegelmann and Sontag].

We can also stack several RNN cells in order to exploit compositionality and obtain more powerful RNNs. Such RNNs are called multilayer or deep RNNs. The computation performed by an L layer vanilla RNN is given with

$$\begin{aligned}
\mathbf{h}_1^{(t)} &= \phi\left(\mathbf{W}^{\mathbf{h}_1\mathbf{h}_1}\, \mathbf{h}_1^{(t-1)} + \mathbf{W}^{\mathbf{x}\mathbf{h}_1}\, \mathbf{x}^{(t)} + \mathbf{b}^{\mathbf{h}_1}\right) \\
\mathbf{h}_2^{(t)} &= \phi\left(\mathbf{W}^{\mathbf{h}_2\mathbf{h}_2}\, \mathbf{h}_2^{(t-1)} + \mathbf{W}^{\mathbf{h}_1\mathbf{h}_2}\, \mathbf{h}_1^{(t)} + \mathbf{b}^{\mathbf{h}_2}\right) \\
&\vdots \\
\mathbf{h}_L^{(t)} &= \phi\left(\mathbf{W}^{\mathbf{h}_L\mathbf{h}_L}\, \mathbf{h}_L^{(t-1)} + \mathbf{W}^{\mathbf{h}_{L-1}\mathbf{h}_L}\, \mathbf{x}^{(t)} + \mathbf{b}^{\mathbf{h}_L}\right) \\
\mathbf{y}^{(t)} &= \text{softmax}\left(\mathbf{W}^{\mathbf{y}}\, \mathbf{h}_L^{(t-1)} + \mathbf{b}^{\mathbf{y}}\right)
\end{aligned}$$

Observe that we will need L initial states $\mathbf{h}_1^{(0)}, \mathbf{h}_2^{(0)}, \ldots, \mathbf{h}_L^{(0)}$.

3.2.2 Backpropagation Through Time and Vanishing Gradient

In practice we need to estimate the parameters of RNN from the data. The way this is done is by minimising the average of a certain loss function which measures how far are the outputs of the network from actual data. Suppose that we are given a dataset $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$. The quantity that we are supposed to minimise is

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(\mathbf{y}_i, \hat{\mathbf{y}}_i)$$

where θ are parameters of the network and $\hat{\mathbf{y}}_i$ is the output of the network on input $\mathbf{x}_i$. The loss function that we will use here is the cross entropy loss given with

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i \,.$$

In order to minimise the loss function we need to calculate its gradient with respect to the parameters of the network. This is achieved by Backpropagation Through Time algorithm [14].

Unfortunately, for vanilla RNNs this does not solve the optimisation problem. The reason is that is that the gradient of the loss function may vanish or explode over many time steps. This problem has been studied independently by separate researchers [Hochreiter][Bengio]. For survey see [paper by Pascanu and Bengio].

The problem of exploding gradient can be solved by gradient clipping: if the norm of the gradient exceeds a certain threshold, then normalise it and multiply it by that threshold. However, the vanishing gradient remains a problem for vanilla RNNs which makes their optimisation difficult in practice.

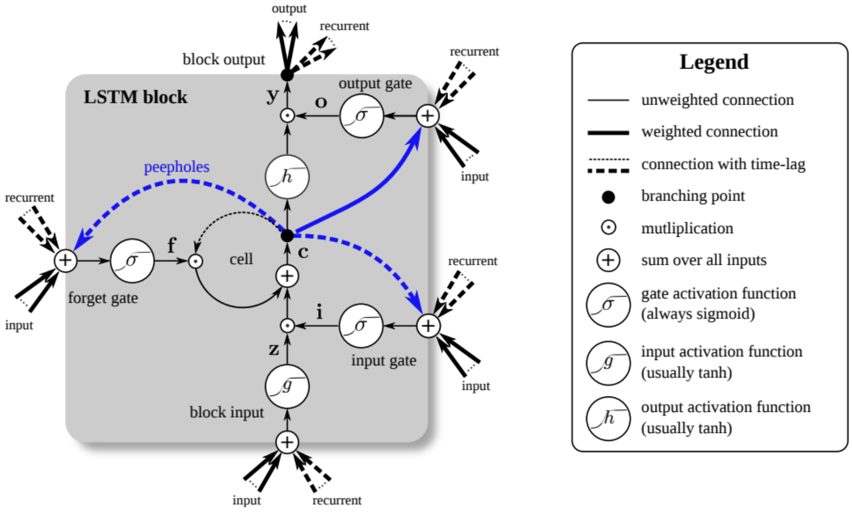
3.2.3 Long-Short Term Memory Networks

One way to deal with vanishing gradient is to use gated RNNs whose connection weights may change at each time step. Multiplicative gates allow RNNs to control the way information is stored and retrieved. Thus, they are able to create paths through time that have derivatives that neither vanish nor explode. Gates in RNNs can be seen as differentiable version of logic gates in digital circuits.

Long short term memory (LSTM) networks [4] use additional memory cell $c^{(t)}$ apart from $h^{(t)}$. Here we will be using standard LSTM architecture introduced in [2] which uses four gates: forget gate $f^{(t)}$, input gate $i^{(t)}$, output gate $o^{(t)}$ and input modulation gate $g^{(t)}$. Its cell is described by the following set of equations

$$\begin{aligned}
\begin{bmatrix} f^{(t)} \\ i^{(t)} \\ o^{(t)} \\ g^{(t)} \end{bmatrix} &= W_x x^{(t)} + W_h h^{(t)} + b \\
c^{(t)} &= \sigma(f^{(t)}) \odot c^{(t-1)} + \sigma(i^{(t)}) \odot \tanh\left(g^{(t)}\right) \\
h^{(t)} &= \sigma(o^{(t)}) \odot \tanh\left(c^{(t)}\right)
\end{aligned}$$

where $\odot$ is component-wise or Hadamard product.



Just like vanilla RNN cells, LSTM cells can be stacked in order to obtain more powerful multilayer LSTM networks.

LSTM networks have been successfully applied in many contexts such as handwriting recognition [reference] and generation, language modeling [reference], machine translation [reference], image captioning [reference], parsing [reference], etc.

3.2.4 Gated Recurrent Units

Another gated recurrent architecture frequently used in practice are Gated Recurrent Units (GRU) [1]. Unlike LSTM, GRU gets rid of memory cell $c^{(t)}$ by combining input and forget gates into a single one. The GRU cell is described by the following set of equations

$$\begin{aligned}
r^{(t)} &= \sigma(W_{xr}x_t + W_{hr}h^{(t-1)} + b_r) \\
z^{(t)} &= \sigma(W_{xr}x_t + W_{hz}h^{(t-1)} + b_z) \\
n^{(t)} &= \tanh\Big(W_{xn}x_t + r_t(W_{hn}h^{(t-1)} + b_n)\Big) \\
h^{(t)} &= (1 - z_t)n_t + z_t h_{t-1}
\end{aligned}$$

GRU figure goes here.

Again, GRU cells can be stacked, yielding multilayer GRU networks.

3.2.5 One-hot and Dense Encodings

Note that RNNs work exclusively with numerical data, i.e. real numbers. Suppose that we are given an alphabet $\mathcal{A} = \{a_1, a_2, \dots a_k\}$. One way to encode it would be to map each a_i to i. However, this is not a good encoding as the network would need to learn that there is no ordering between alphabet elements. A better way to encode the alphabet that avoids this problem is to assign a one-hot vector to every element of the alphabet, i.e.

$$a_1 \mapsto \begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix} \quad a_2 \mapsto \begin{bmatrix}0\\1\\\vdots\\0\end{bmatrix} \quad \dots \quad a_k \mapsto \begin{bmatrix}0\\0\\\vdots\\1\end{bmatrix}$$

One problem with this encoding is that k may be huge while one-hot vectors remain sparse. For example, in natural language processing k would be the number of words which is usually several tens of thousands. This problem is solved by using embedding layer which learns to map alphabet elements to dense, low-dimensional encodings.

3.2.6 Encoder-Decoder Sequence to Sequence Models

3.2.7 Attention Mechanism

When processing the input, RNNs store all the memory in the hidden state. However, it may be hard to compress potentially long input in a single context vector. This could definitely be an issue if the hidden state is passed as input to another network which is common in sequence to sequence learning. For example, the family of models used for machine translation has one RNN called encoder to process the input in one language and pass it to the other RNN called decoder which is supposed to output a sequence in a different language.

Attention mechanism allows us to obtain a distribution on part of the input provided so far and focus on certain parts of it in a way we humans do. The attentional hidden state is defined as

$$\widetilde{h}^{(t)} = \tanh\Big(W_c[c^{(t)}; h^{(t)}]\Big)$$

where $c^{(t)}$ is the context vector and $[;]$ denotes concatenation. We will be using an attention model in which the alignment vector $a^{(t)}$ is obtained by comparing the current hidden state $h^{(t)}$ with each source hidden state $\bar{h}^{(s)}$

$$a^{(t)} = \text{align}(h^{(t)}, \bar{h}^{(s)}) = \text{softmax}(\text{score}(h^{(t)}, \bar{h}^{(s)}))\,.$$

Attention figure goes here.

3.2.8 Layer Normalization

Chapter 4

Experiments and Results

In this chapter we ...

4.1 Environment

We will be using two datasets called Factor1 and Factor5. Factor1 contains 4 positive and 4 negative examples of length 18 from each of 216 locally 3-testable languages specified by one 3-factor. The dataset is split into a train subset containing 180 languages with corresponding positive and negative examples and a test subset containing 36 languages with corresponding positive and negative examples. Factor5 contains 4 positive and 4 negative examples of length 18 from each of 5000 locally 3-testable languages specified by five 3-factors. The dataset is split into a train subset containing 4000 languages with corresponding positive and negative examples and a test subset containing 1000 languages with corresponding positive and negative examples. In both cases, not a single 3-factor that occurs in the train subset set occurs in the test subset. This ensures that the test accuracy measures how well the trained model generalises out of the sample.

A typical line in a Factor5 dataset looks like one of the following two.

```
abc#acd#bae#dfb#ecf    abaddfecbcffaebcad    1
abc#acd#bae#dfb#ecf    abaddfecbcacdebcad    0
```

The first part represents a 3-locally testable language specified by five 3-factors “abc”, “acd”, “bae”, “dfb” and “ecf”. The second part represents an input string (of length 18) and the third part whether the input string belongs to the language specified by the first part. For example, in the first line above string “abaddfecbcffaebcad” belongs to the language, while in the second line string “abaddfecbcacdebcad” does not belong to the language because it contains “acd” as a substring.

4.2 Methods

Similarly to sequence to sequence models (Section 3.x) we will be using two statistical models called encoder and decider. An encoder takes language representation as input and outputs the context vector **c** which acts as a vector description of language. A decider takes the context vector from encoder and string as input and outputs the probability that the string belongs to the language encoded in that context vector.

Encoder-decider architecture figure goes here.

In the following, encoder will always be an RNN whose final hidden state will be the context vector. The model of the decider will vary across experiments. Both encoder and decider are trained by minimising the mean cross entropy between the actual value of the string belonging to language (0 or 1) and predicted probability that the input string belongs to the language. For optimisation we will be using Adam optimisation algorithm [7].

4.3 RNN Decider

In this section both encoder and decider will be RNNs. We will be using LSTMs and GRUs with 1 or 2 layers and 20, 50 or 100 hidden units per layer. Embeddings of size 10 are trained jointly with the model. The maximal test accuracy over the period of 50 epochs is reported in Table 4.1.

RNN	Number of layers	Number of hidden units	Test accuracy
LSTM	1	20	58.68%
LSTM	1	50	59.72%
LSTM	1	100	57.99%
LSTM	2	20	58.68%
LSTM	2	50	57.99%
LSTM	2	100	55.9%
GRU	1	20	60.42%
GRU	1	50	58.33%
GRU	1	100	61.46%
GRU	2	20	61.81%
GRU	2	50	59.03%
GRU	2	100	62.5%

Table 4.1: Factor1

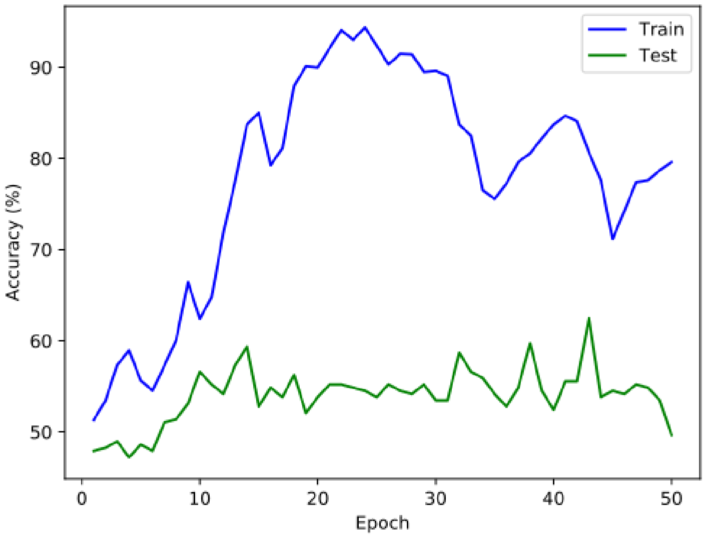


Figure 4.1: Evolution of train and test accuracy of model for which the maximum is achieved.

RNN	Number of layers	Number of hidden units	Test accuracy
LSTM	1	20	52.51%
LSTM	1	50	52.6%
LSTM	1	100	51.59%
LSTM	2	20	52.48%
LSTM	2	50	52.52%
LSTM	2	100	52.14%
GRU	1	20	53.35%
GRU	1	50	54.35%
GRU	1	100	52.41%
GRU	2	20	53.42%
GRU	2	50	52.48%
GRU	2	100	54.01%

Table 4.2: Factor5

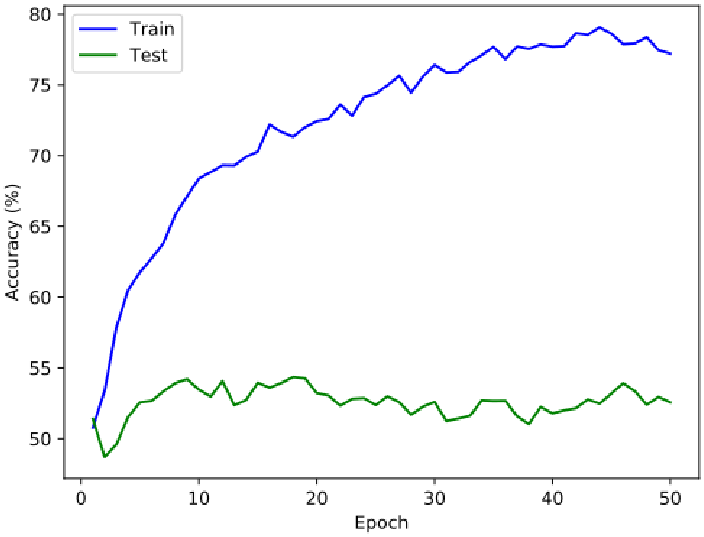


Figure 4.2: Evolution of train and test accuracy of model for which the maximum is achieved.

We see that RNNs fail to generalise to unseen 3-factors.

4.4 Adding Attention

Now we add the attention as described in Section 4.x to the models studied in the previous section. Again, the maximal test accuracy over the period of 50 epochs is taken. The results are displayed in Table 4.2.

RNN	Number of layers	Number of hidden units	Test accuracy
LSTM	1	20	58.33%
LSTM	1	50	58.68%
LSTM	1	100	56.6%
LSTM	2	20	56.34%
LSTM	2	50	59.03%
LSTM	2	100	58.33%
GRU	1	20	67.01%
GRU	1	50	64.24%
GRU	1	100	59.72%
GRU	2	20	62.5%
GRU	2	50	59.72%
GRU	2	100	61.11%

Table 4.3: Factor5

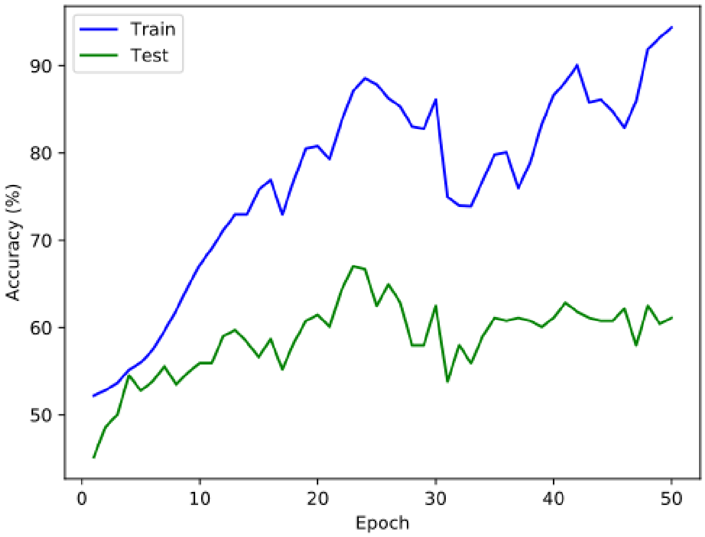


Figure 4.3: Evolution of train and test accuracy of model for which the maximum is achieved.

RNN	Number of layers	Number of hidden units	Test accuracy
LSTM	1	20	80.86%
LSTM	1	50	80.23%
LSTM	1	100	72.56%
LSTM	2	20	65.33%
LSTM	2	50	65.26%
LSTM	2	100	64.96%
GRU	1	20	81.33%
GRU	1	50	78.97%
GRU	1	100	79.75%
GRU	2	20	73.56%
GRU	2	50	67.26%
GRU	2	100	66.29%

Table 4.4: Factor5

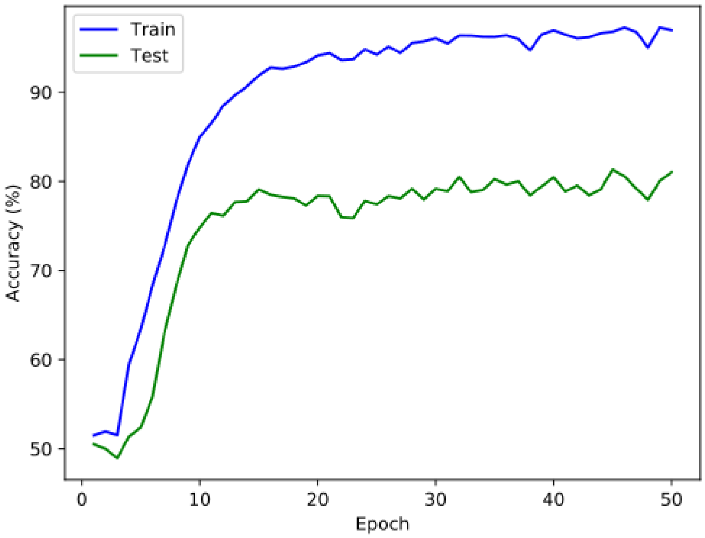


Figure 4.4: Evolution of train and test accuracy of model for which the maximum is achieved.

We see that using attention greatly improves generalisation of RNNs. However, the performance is still far away from perfect, i.e. test accuracy of 100%.

4.5 Decider as Scanner

In this section we provide some useful bias to the model by constraining the decider to a differentiable version of a scanner that contains a look-up table as such structure is sufficient to decide a locally k-testable language.

4.5.1 Implementing Scanner

Here we show how a scanner that operates on vector encodings of strings can be implemented. Suppose that we are given a k-locally testable language specified by F k-factors. The following algorithm describes such a scanner.

Algorithm: Scanner

```
Input: context vector $\mathbf{c}$, string $x$ of length $n$
Split context vector $\mathbf{c}$ into $F$ parts of equal size $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_F$.
for $i = 1$ to $n - k + 1$ do
    Take encodings of $x_i, x_{i+1}, \ldots, x_{i+k-1}$ and concatenate them. Let $\bar{\mathbf{x}}_i$ be the resulting vector.
    $m_{ij} \leftarrow \langle \bar{\mathbf{x}}_i, \mathbf{c}_j \rangle$
end
$m \leftarrow \max_{i,j} m_{ij}$
return $\sigma(m)$
```

In other words, scanner takes a context vector and input string and uses dot product to check if there is a match between parts of context vector and substrings of input string of length k.

Scanner figure goes here.

Hence, the goal of encoder is to construct a look-up table by arranging the encodings of k-factors in the context vector $\mathbf{c}$.

To verify that the proposed model works correctly, we take the encodings of 3-factors in Factor1 and Factor5 and concatenate them to obtain the context vector that we pass to the scanner. On both datasets we obtain the test accuracy of 100%.

4.5.2 Passing Representations to LSTM

Here we will use only the final hidden state of the RNN encoder as the context vector. This tests whether an RNN can arrange the copies of k-factors in its memory. We will use LSTM with hidden size equal to $3d$ times the number of 3-factors (i.e. $3d$ for Factor1 and $15d$ for Factor5), where d is the embedding dimension. On Factor1 we train the model for 100 epochs. We find out that the test accuracy highly depends on initial values of parameters. To study this, we perform 100 experiments with different values of embedding dimension d and learning rate λ. The results are displayed in Figure 4.5.

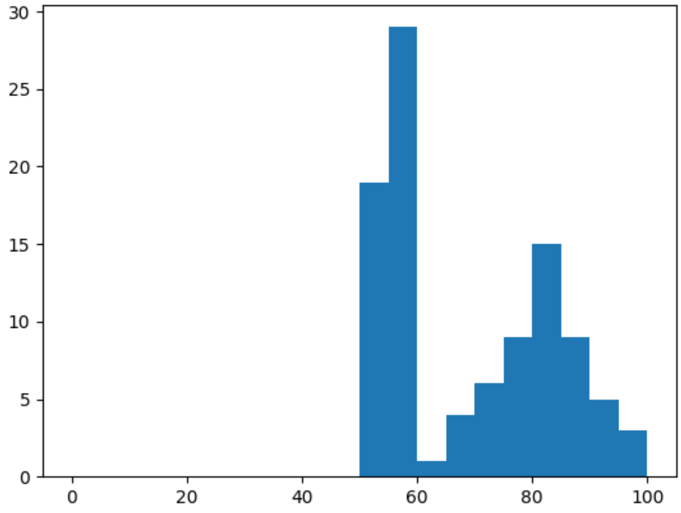


Figure 4.5: Factor1

For Factor5 we take the values of d and λ for which the model generalised most times on Factor1 and perform 50 experiments with 200 epochs each. Unfortunately, none of the runs generalises to unseen 3-factors. The results are displayed in Figure 4.6.

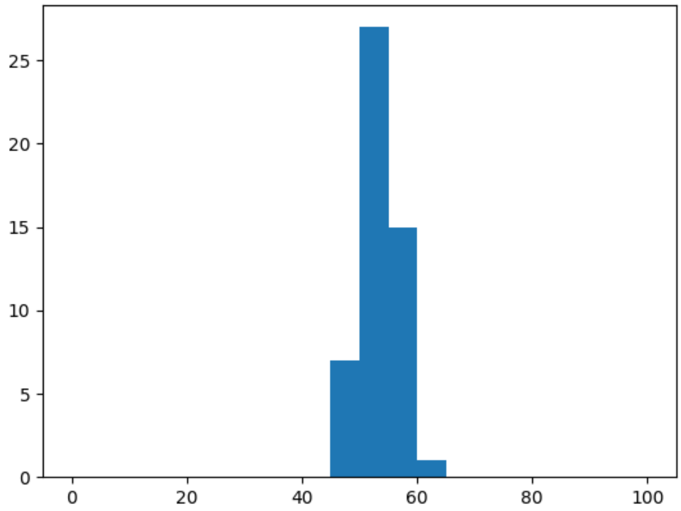


Figure 4.6: Factor5

4.5.3 Passing factors to LSTM

Here we show that RNN is able to learn to store sufficiently similar copies of factors in its memory. We do this in the following way: we feed the encodings of symbols from language representation to the RNN. We concatenate the hidden states corresponding to 3-factors (ignoring hidden states corresponding to "#") and obtain the context vector which we pass to the scanner. We use LSTM with hidden size equal to the embedding dimension d. On FACTOR1 we train the model for 100 epochs. Again, the test accuracy highly depends on initial values of parameters. We run 100 experiments with different values of embedding size d and learning rate λ. The results are displayed in Figure 4.7.

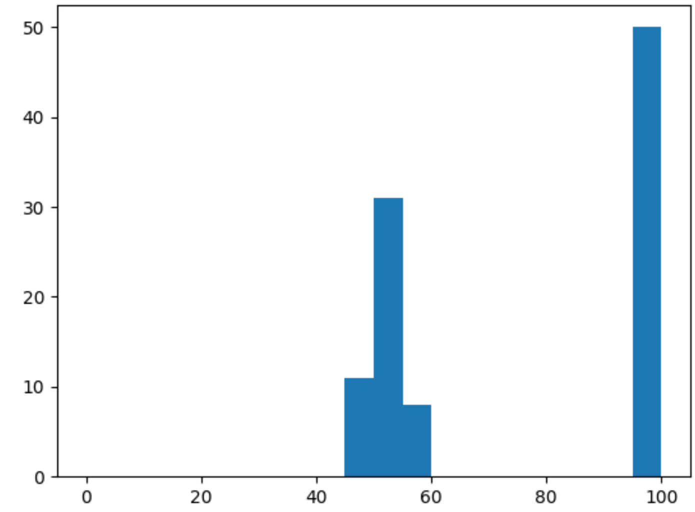


Figure 4.7: Factor1

For FACTOR5 we take the values of d and λ for which the model generalised most times on FACTOR1 and perform 50 experiments with 200 epochs each. The results are displayed in Figure 4.8.

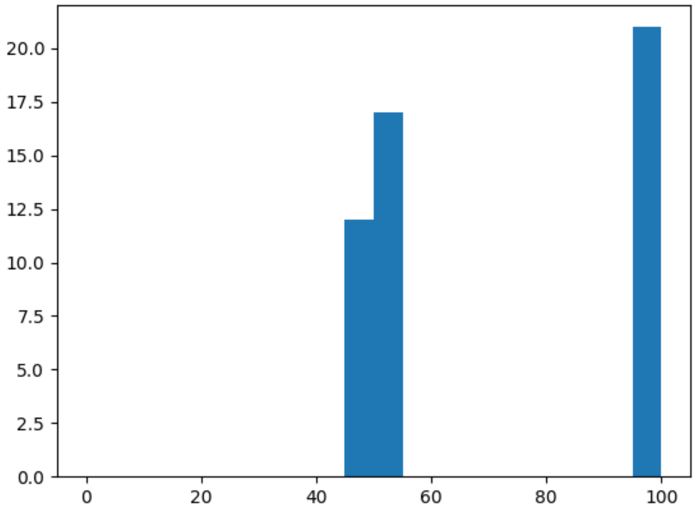


Figure 4.8: Factor5

On both datasets the model is able to achieve the test accuracy of 100%.

4.5.4 Using Positional Encodings

One of the problems of approach used in 4.5.2 is that the whole hidden state is reserved for storing the encodings of 3-factors. In that way RNN does not know how to store the information at each time step. This naturally motivates the use of positional encodings, i.e. instead of feeding the encoding of the symbol to the RNN, we feed it the encoding of the symbol concatenated with the encoding of the symbol position in language representation. As in 4.5.2 we will use LSTM with hidden size equal to $3d$ times the number of 3-factors (i.e. $3d$ for Factor1 and $15d$ for Factor5), where d is the embedding dimension. On Factor1 we train the model for 100 epochs. Again, the test accuracy highly depends on initial values of parameters. We run 100 experiments with different values of embedding size d and learning rate λ. The results are displayed in Figure 4.9.

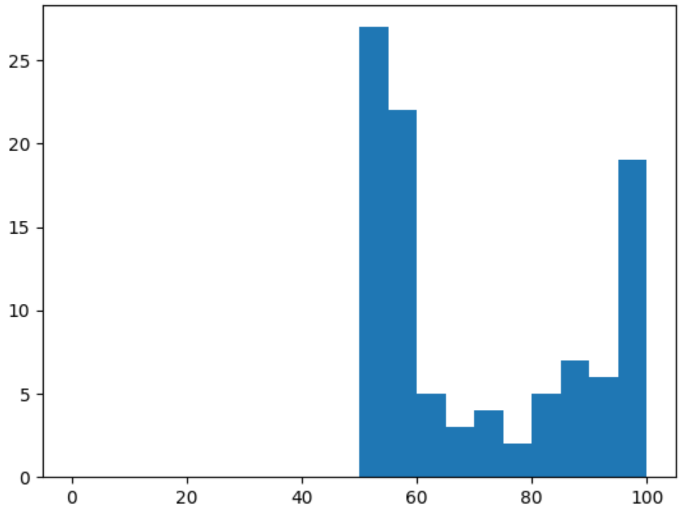


Figure 4.9: Factor1

For Factor5 we take the values of d and λ for which the model generalised most times on Factor1 and perform 50 experiments with 200 epochs each. The results are displayed in Figure 4.10.

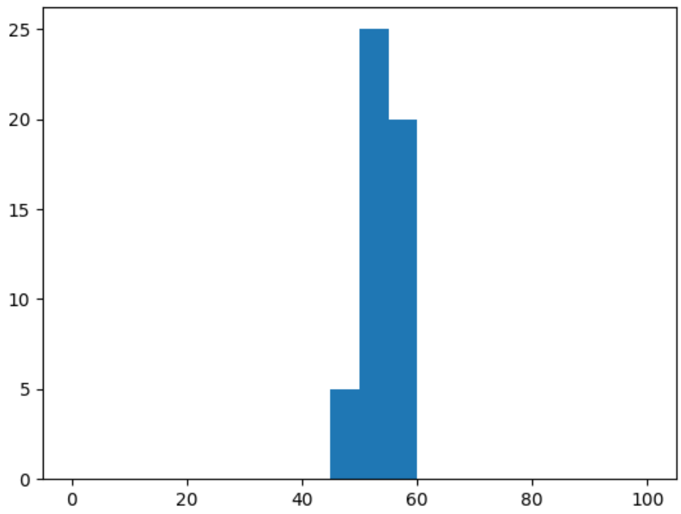


Figure 4.10: Factor1

4.5.5 Supervising LSTM Gates

Here we force the LSTM to store the information at the right location by providing supervision on its gate activations at each time step. In the case of Factor1, we want to store the first symbol in first third hidden units, second symbol in the second third hidden units and third symbol in last third hidden units. This is achieved by setting the first third input gates to be close to 1 and rest close to 0 on first time step; second third input gates to be close to 1 and rest close to 0 on second time step; last third input gates to be close to 1 and rest close to 0 on third time step. Similarly, for forget gates we want first third to be close to 1 on second time step; first and second third to be close to 1 on third time step. The idea is naturally extended for more 3-factors. We will use cross entropy to measure the error in gate activations. The loss associated to the gates will be multiplied by a hyperparameter γ and added to the original loss. On Factor1 we train the model for 100 epochs. Again, the test accuracy highly depends on initial values of parameters. We run 100 experiments with different values of embedding size d and learning rate λ. The results are displayed in Figure 4.11.

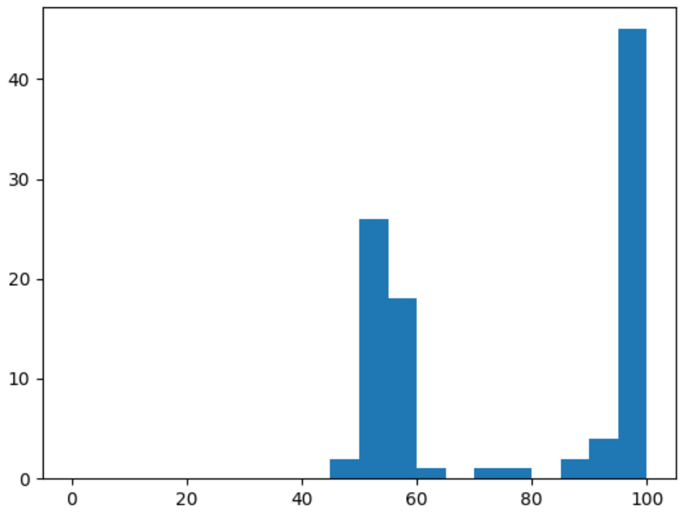


Figure 4.11: Factor1

For Factor5 we take the values of d, λ and γ for which the model generalised most times on Factor1 and perform 50 experiments with 200 epochs each. The results are displayed in Figure 4.10.

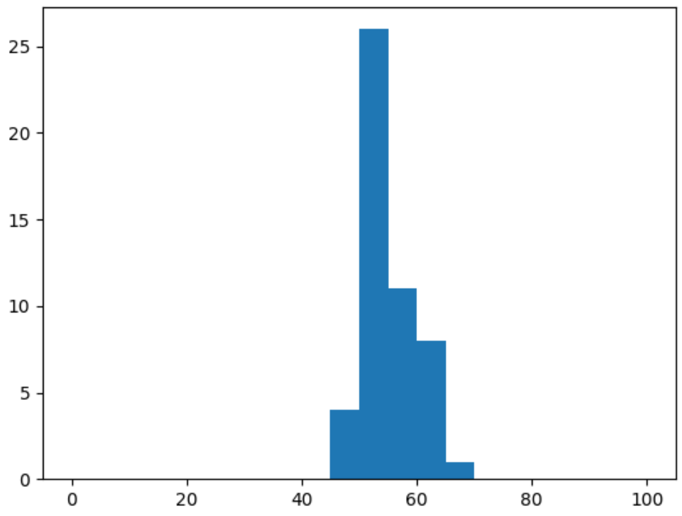


Figure 4.12: Factor1

Chapter 5

Discussion and Future Work

Bibliography

[1] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[2] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999).

[3] Alex Graves. “Supervised sequence labelling”. In: *Supervised sequence labelling with recurrent neural networks*. Springer, 2012, pp. 5–13.

[4] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[5] Marcus Hutter. “The fastest and shortest algorithm for all well-defined problems”. In: *International Journal of Foundations of Computer Science* 13.03 (2002), pp. 431–443.

[6] Armand Joulin and Tomas Mikolov. “Inferring algorithmic patterns with stack-augmented recurrent nets”. In: *Advances in neural information processing systems*. 2015, pp. 190–198.

[7] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[8] Leonid Anatolevich Levin. “Universal sequential search problems”. In: *Problemy Peredachi Informatsii* 9.3 (1973), pp. 115–116.

[9] Rodolfo Llinas. “Neuron”. In: *Scholarpedia* 3.8 (2008). revision #91570, p. 1490. DOI: 10.4249/scholarpedia.1490.

[10] Michael C Mozer and Sreerupa Das. “A connectionist symbol manipulator that discovers the structure of context-free languages”. In: *Advances in neural information processing systems*. 1993, pp. 863–870.

[11] Stephen Muggleton. “Inductive logic programming”. In: *New generation computing* 8.4 (1991), pp. 295–318.

[12] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[13] Jürgen Schmidhuber. “Optimal ordered problem solver”. In: *Machine Learning* 54.3 (2004), pp. 211–254.

[14] Paul J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[15] Janet Wiles and Jeff Elman. “Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks”. In:

[16] Wojciech Zaremba and Ilya Sutskever. “Learning to execute”. In: *arXiv preprint arXiv:1410.4615* (2014).