LEARNING TO DECIDE A FORMAL LANGUAGE: A RECURRENT NEURAL NETWORK APPROACH

MSc Thesis *(Afstudeerscriptie)*

written by

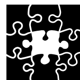
Krsto Proroković
(born March 12, 1993 in Kotor, Montenegro)

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:**
August 29th, 2018	Dr Floris Roelofsen
	Dr Tom Lentz
	Dr Miguel Rios Gaona
	Dr Elia Bruni
	Dieuwke Hupkes MSc



INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

We use recurrent neural networks (RNNs) for deciding locally k-testable languages. We show that, when used for deciding languages, RNNs fail to generalise to unseen examples. However, using attention greatly improves the generalisation. We then implement a differentiable version of the scanner used for deciding locally k-testable languages. We show that RNNs are able to store the k-factors in its memory but not arrange then as a look-up table which is necessary for deciding languages specified by multiple k-factors.

Acknowledgents

I would like to thank my supervisors Germán, Elia and Dieuwke (whose name sadly does not appear on the title page due to formal constraints) for all the guidance during the development of this thesis and for caring not only about my work, but also about my wellbeing and future.

I also thank my fellow Master of Logic colleagues for all the collaboration during my two years spent in Amsterdam.

I am grateful to Floris Roelofsen, Tom Lentz and Miguel Rios Gaona for finding time to read this thesis.

Lastly, I thank my family for their constant support.

Contents

1 Introduction **4**
1.1 Motivation . . . 4
1.2 Organisation of the Thesis . . . 4

2 Related Work **5**

3 Background **6**
3.1 Locally k-testable languages . . . 6
3.2 Recurrent Neural Networks . . . 8
3.2.1 Vanilla Recurrent Neural Networks . . . 8
3.2.2 Backpropagation Through Time and Vanishing Gradient . . . 11
3.2.3 Long-Short Term Memory Networks . . . 12
3.2.4 Gated Recurrent Units . . . 13
3.2.5 One-hot and Dense Encodings . . . 14
3.2.6 Encoder-Decoder Sequence-to-Sequence Models . . . 15
3.2.7 Attention Mechanism . . . 15
3.2.8 Layer Normalisation . . . 16

4 Experiments and Results **17**
4.1 Data . . . 17
4.2 Methods . . . 18
4.3 RNN Decider . . . 18
4.4 Adding Attention . . . 21
4.5 Decider as Scanner . . . 23
4.5.1 Last Hidden State of Encoder as Look-Up Table . . . 24
4.5.2 Can RNNs Store k-Factors? . . . 26
4.5.3 Using Positional Encodings . . . 28
4.5.4 Supervising LSTM Gates . . . 29

5 Conclusion and Future Work **32**
5.1 Register RNNs . . . 32

Bibliography **33**

Chapter 1

Introduction

1.1 Motivation

Learning the meaning of formal instructions from data is one of the fascinating aspects of human intelligence. For example, having seen the execution of several simple computer programs, humans can infer the semantics behind the if-then-else clause or for-loop. On the other hand, current state of the art machine learning methods would not be able to do so without extremely huge amount of data.

In this thesis we study a closely related problem - learning to decide a formal language from data. To see how these two are connected, think of regular expressions. One cannot decide whether a string belongs to the regular language specified by a certain regular expression, without understanding the meaning of regular language constructs such as conjunction or Kleene star. Therefore, solving the second problem would bring us a step closer to teaching machines to execute our instructions.

1.2 Organisation of the Thesis

In Chapter 2 we give an overview of the related work. In Chapter 3 we provide the background necessary to understand the experiments in Chapter 4. In Chapter 4 we will use recurrent neural networks for learning to decide a simple class of formal languages. In Chapter 5 we will summarise the results obtained in Chapter 4 and provide some future directions. We expect the reader to be familiar with linear algebra and formal language theory and to have seen some multivariable calculus. Many figures are intentionally hand-drawn using author's favourite uni-ball Signo 207 pen.

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 [20] requires domain specific knowledge about the programming languages and hand-crafted heuristics to speed up the underlying combinatorial search. Levin [16] devised an asymptotically optimal method for inverting functions on given output, albeit with a large constant factor. Hutter [11] 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 [25] 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 [19] used neural networks with external stack memory to parse simple context-free languages such as $a^n b^n$ and parenthesis balancing from both positive and negative examples. Wiles and Elman [30] used neural networks to learn sequences of the form $a^n b^n$ and generalise on a limited range of n. Joulin and Mikolov [14] 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 [31] used recurrent neural networks for learning to execute simple Python programs.

Chapter 3

Background

In this chapter we provide background material needed for understanding and reproducing results from Chapter 4. In first part we introduce locally k-testable languages - a simple class of languages that we will be dealing with later. In second part we introduce recurrent neural networks - statistical models for processing sequential data.

3.1 Locally k-testable languages

We start by providing several definitions from formal language theory. An alphabet $\mathcal{A}$ is a finite set of symbols. A string is a finite sequence of symbols drawn from the alphabet. A language is a set of strings over an alphabet. The number of symbols in a string is called the length of the string. Strings of length k are also called k-factors. A strictly k-local or SL_k definition is a set of k-factors. A string satisfies an SL_k definition if and only if every k-factor that occurs in the string is an element of that definition. The class of languages defined using SL_k definitions is called strictly k-local.

An algorithm decides a language if, given a string, outputs whether the string belongs to the language. A strictly k-local language can be decided using a scanner that contains a look-up table of k-factors: scanner walks through the string and checks whether every k-factor that occurs in the string occurs in the look-up table. Therefore, look-up table in this case corresponds to SL_k definition.

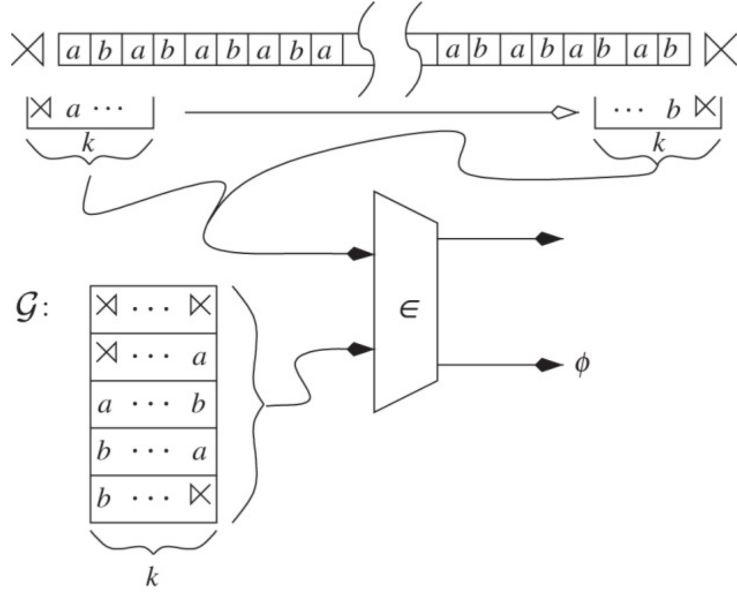


Figure 3.1: Scanner for strictly k-local language has a sliding window of size k that slices across the string and picks k-factors. Then, it checks whether each of these k-factors is included in a look-up table. Figure taken from [12].

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. For example, k-expression

$$\varphi(q_1, q_2, \ldots, q_F) = \neg q_1 \wedge \neg q_2 \wedge \ldots \wedge q_F \tag{3.1}$$

defines all strings that do not contain any of $q_1, q_2, \ldots, q_F$.

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.

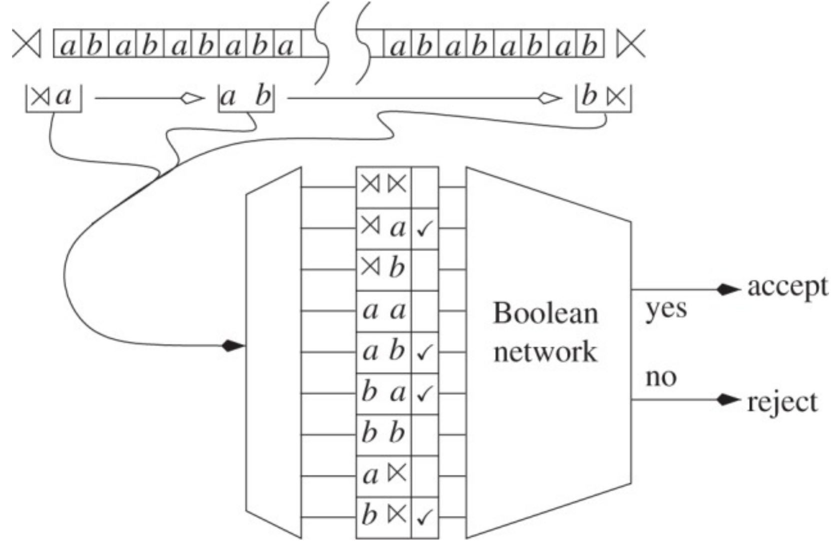


Figure 3.2: An LT_k automata tracks which k-factors occurred in a string and then passes this information to a Boolean network. The automata accepts a string if the output of the network is positive. Figure taken from [12]

3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) [23] are parametric models of computation for processing sequences loosely inspired by biological neural networks. They found applications in handwriting recognition [7], speech recognition [24], machine translation [18], etc. In this section we provide an introduction to RNNs including For a more detailed treatment we point the reader to [6].

3.2.1 Vanilla Recurrent Neural Networks

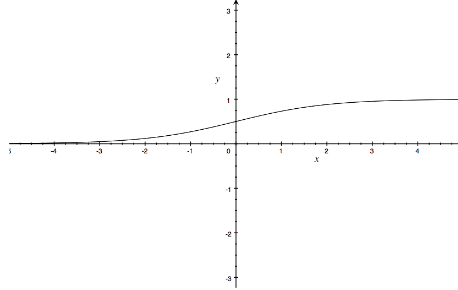
We start with a simple vanilla RNN model. We will denote the input sequence by $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$ where $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$ for every $t = 1, 2, \ldots, T$. The output of RNN will be another sequence $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T)}$ where $\mathbf{y}^{(t)}$ is a probability distribution on d_y outcomes for $t = 1, 2, \ldots, T$.

A vanilla RNN with hidden state dimension d_h is given with the following parameters:

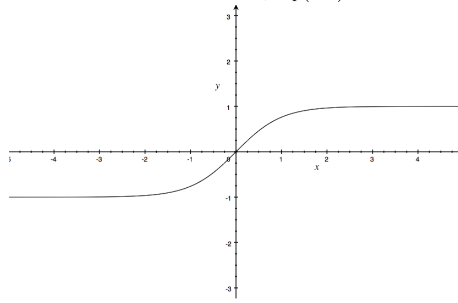
- Input to hidden connection weights $\mathbf{W}^{\mathrm{x}} \in \mathbb{R}^{d_h \times d_x}$
- Hidden to hidden connection weights $\mathbf{W}^{\mathrm{h}} \in \mathbb{R}^{d_h \times d_h}$
- Bias term $\mathbf{b}^{\mathrm{h}} \in \mathbb{R}^{d_h}$
- Activation function $\phi : \mathbb{R} \to \mathbb{R}$
- Hidden to output connection weights $\mathbf{W}^{\mathrm{y}} \in \mathbb{R}^{d_y \times d_h}$

- Bias term $\mathbf{b}^{\mathrm{y}} \in \mathbb{R}^{d_y}$
- Initial hidden state $\mathbf{h}^{(0)} \in \mathbb{R}^{d_h}$ (usually set to zero vector)

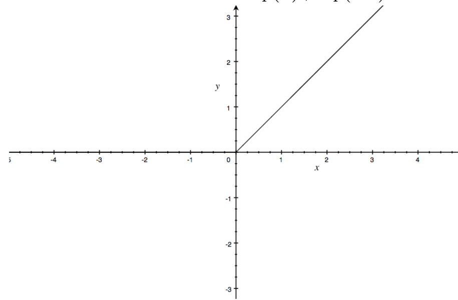
Connection weights model the strength of synapses and activation function models neuronal firing [17]. Most commonly used activation functions are sigmoid $\sigma(x) = 1/(1+\exp(-x))$, hyperbolic tangent $\tanh(x) = (\exp(x)-\exp(-x))/(\exp(x)+\exp(-x))$ and rectified linear unit $\mathrm{ReLU}(x) = \max\{0, x\}$.



(a) $\sigma(x) = \frac{1}{1+\exp(-x)}$



(b) $\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$



(c) $\mathrm{ReLU}(x) = \max\{0, x\}$

Figure 3.3: Most commonly used activation functions.

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

$$\mathbf{h}^{(t)} = \boldsymbol{\phi}\left(\mathbf{W}^{\mathrm{x}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{h}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{h}}\right) \tag{3.2}$$

$$\mathbf{y}^{(t)} = \mathrm{softmax}\left(\mathbf{W}^{\mathrm{y}}\,\mathbf{h}^{(t)} + \mathbf{b}^{\mathrm{y}}\right) \tag{3.3}$$

where $\boldsymbol{\phi}(\mathbf{z})_i = \phi(z_i)$ and $\text{softmax}(\mathbf{z})_i = \exp(z_i)/\sum_j \exp(z_j)$.

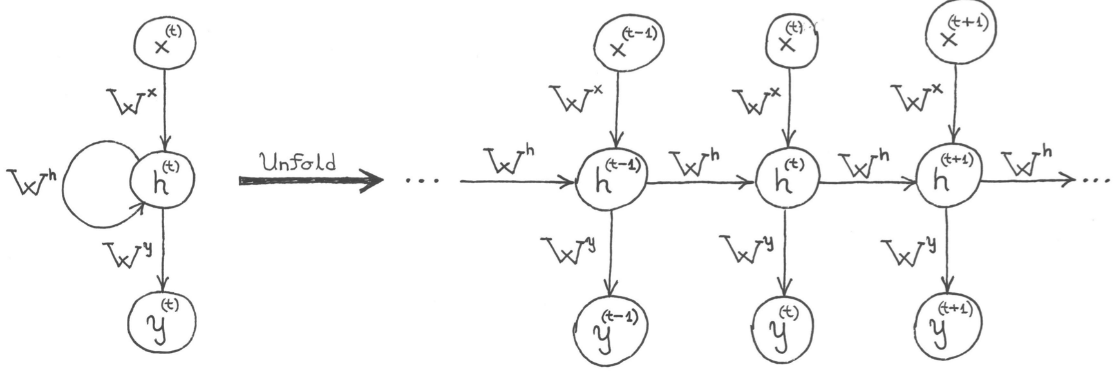


Figure 3.4: Computation graph of vanilla RNN. Bias terms and activations are omitted for simplicity.

Equation 3.2 describes what is called an RNN cell. Note that components of softmax sum up to one, so we can interpret it as a probability distribution. Here we will be mostly interested in sequence classification and thus we will only care about $\mathbf{y}^{(T)}$.

Example Consider a vanilla RNN given with the following parameters

$$\mathbf{W}^{\mathrm{x}} = \begin{bmatrix} 3 & -1 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad \mathbf{W}^{\mathrm{h}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{b}^{\mathrm{h}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{W}^{\mathrm{y}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{b}^{\mathrm{y}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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 $\mathbf{h}^{(t)}$ 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 [26].

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

$$\begin{aligned}
\mathbf{h}_1^{(t)} &= \boldsymbol{\phi}\left(\mathbf{W}^{0,1}\,\mathbf{x}^{(t)} + \mathbf{W}^{1,1}\,\mathbf{h}_1^{(t-1)} + \mathbf{b}^1\right) \\
\mathbf{h}_2^{(t)} &= \boldsymbol{\phi}\left(\mathbf{W}^{1,2}\,\mathbf{h}_1^{(t)} + \mathbf{W}^{2,2}\,\mathbf{h}_2^{(t-1)} + \mathbf{b}^2\right) \\
&\ \ \vdots \\
\mathbf{h}_L^{(t)} &= \boldsymbol{\phi}\left(\mathbf{W}^{L-1,L}\,\mathbf{h}_L^{(t)} + \mathbf{W}^{L,L}\,\mathbf{h}_L^{(t-1)} + \mathbf{b}^L\right) \\
\mathbf{y}^{(t)} &= \text{softmax}\left(\mathbf{W}^{\mathrm{y}}\,\mathbf{h}_L^{(t)} + \mathbf{b}^{\mathrm{y}}\right)
\end{aligned}$$

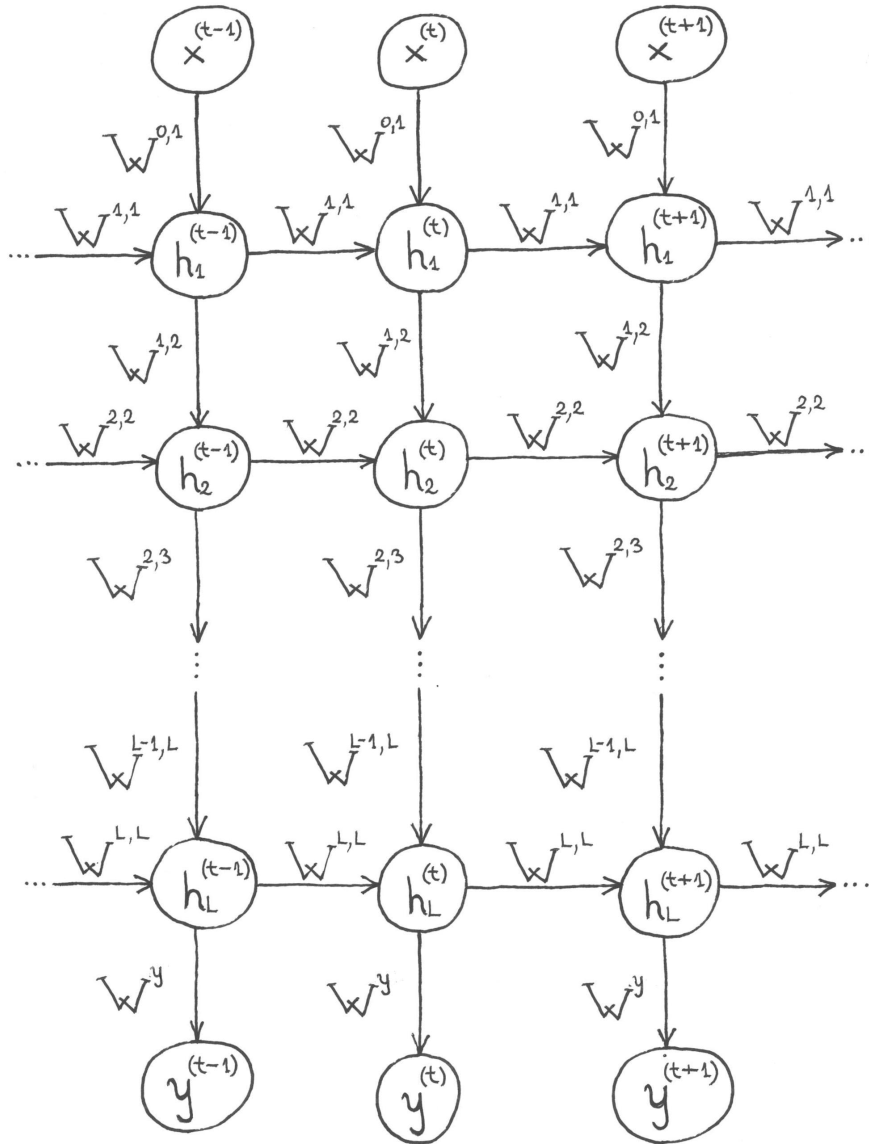


Figure 3.5: Computation graph of L-layer vanilla RNN.

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 the actual data. Suppose that we are given a

dataset $\boldsymbol{\mathcal{D}} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_n, \mathbf{y}_n)\}$. The network is trained by minimising

$$\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{D}}) = \frac{1}{n} \sum_{(\mathbf{x},\mathbf{y}) \in \boldsymbol{\mathcal{D}}} \ell(\mathbf{y}, \hat{\mathbf{y}}) \tag{3.4}$$

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

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i \; . \tag{3.5}$$

We train the network using mini-batch gradient descent. Dataset is split into B parts $\boldsymbol{\mathcal{D}}_1, \boldsymbol{\mathcal{D}}_2, \dots, \boldsymbol{\mathcal{D}}_B$ called batches. Parameters are updated in the following way: for $b = 1, 2, \dots, B$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{\mathcal{D}}_b) \tag{3.6}$$

η is a hyperparameter called learning rate. It needs to be small enough to ensure that training converges to a local minima, yet large enough so that the training is not too slow. Similarly, batch size needs to be small enough so that parameters are updated frequently, yet large enough so that the training is not too slow. More sophisticated update rules than 3.6 that automatically adapt η have been developed [4][15], but we will not go into them here. To calculate the gradient of a sequence model Backpropagation Through Time algorithm [29] is used.

Unfortunately, for vanilla RNNs this algorithm is not directly applicable. 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 [9][2]. For survey see [21].

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 [10] use additional memory cell $\mathbf{c}^{(t)}$ apart from $\mathbf{h}^{(t)}$. Here we will be using standard LSTM architecture introduced in [5] which uses three gates: input gate $\mathbf{i}^{(t)}$, forget gate $\mathbf{f}^{(t)}$ and output gate $\mathbf{o}^{(t)}$. Its cell is described

by the following set of equations

$$\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{xi}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{hi}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{i}}\right) \tag{3.7}$$

$$\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{xf}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{hf}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{f}}\right) \tag{3.8}$$

$$\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{xo}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{ho}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{o}}\right) \tag{3.9}$$

$$\tilde{\mathbf{c}}^{(t)} = \tanh\left(\mathbf{W}^{\mathrm{xc}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{hc}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{c}}\right) \tag{3.10}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)} \tag{3.11}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh\left(\mathbf{c}^{(t)}\right) \tag{3.12}$$

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

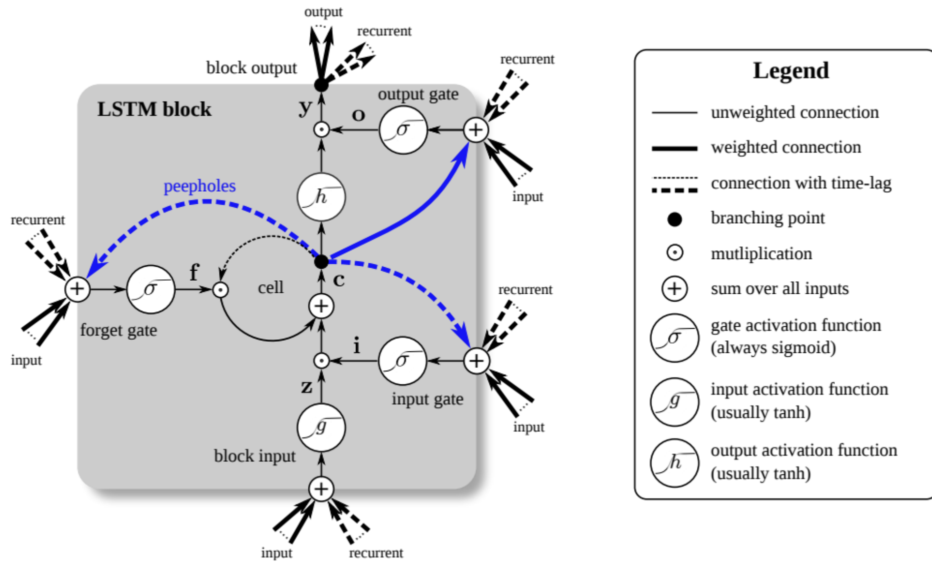


Figure 3.6: LSTM cell. Figure taken from [8].

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

3.2.4 Gated Recurrent Units

Another gated recurrent architecture frequently used in practice are Gated Recurrent Units (GRU) [3]. Unlike LSTM, GRU gets rid of memory cell $\mathbf{c}^{(t)}$ by combining input and forget gates into a single one. The GRU cell is described by the following set of

equations

$$\mathbf{r}^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{xr}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{hr}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{r}}\right) \tag{3.13}$$

$$\mathbf{z}^{(t)} = \sigma\left(\mathbf{W}^{\mathrm{xz}}\,\mathbf{x}^{(t)} + \mathbf{W}^{\mathrm{hz}}\,\mathbf{h}^{(t-1)} + \mathbf{b}^{\mathrm{z}}\right) \tag{3.14}$$

$$\mathbf{n}^{(t)} = \tanh\left(\mathbf{W}^{\mathrm{xn}}\,\mathbf{x}^{(t)} + \mathbf{r}^{(t)} \odot \left(\mathbf{W}^{\mathrm{hn}}\,\mathbf{h}^{(t-1)}\right) + \mathbf{b}^{\mathrm{n}}\right) \tag{3.15}$$

$$\mathbf{h}^{(t)} = \left(\mathbf{1} - \mathbf{z}^{(t)}\right) \odot \mathbf{n}^{(t)} + \mathbf{z}^{(t)} \odot \mathbf{h}^{(t-1)} \tag{3.16}$$

where $\mathbf{1}$ is the all-ones vector $(1, 1, \dots, 1)^T$.

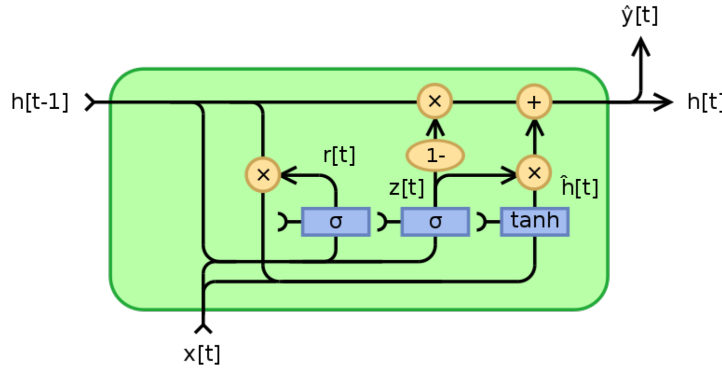


Figure 3.7: GRU cell. Figure taken from Wikipedia.

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 represent each a_i as the integer value 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.

In what follow by encoding of the string we will mean encodings of its symbols concatenated into a single vector.

3.2.6 Encoder-Decoder Sequence-to-Sequence Models

So far we only considered architectures that map an input sequence to an output sequence of the same length. However, for tasks like machine translation or question answering this is not appropriate. This issue is solved by using a sequence to sequence models [27] consisting of two RNNs called encoder and decoder. An encoder processes the input sequence $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T_x)}$ and outputs the context vector $\mathbf{c}$ (usually the final hidden state $\mathbf{h}^{(T_x)}$). A decoder is conditioned on the context vector, might take additional input (say, $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T_x)}$ again) and outputs the sequence $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T_y)}$.

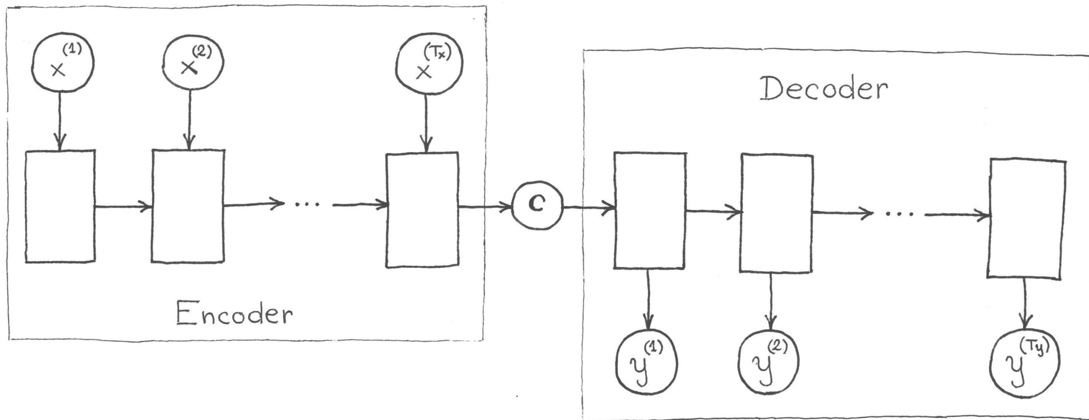


Figure 3.8: Encoder-decoder sequence-to-sequence architecture.

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 a potentially long input in a single context vector. This could definitely be an issue for sequence to sequence models introduced above. Attention mechanism allows us to obtain a distribution $\mathbf{a}$ on part of the input provided so far and focus on certain parts of it in a way we humans do. Here we will assume that decoder takes an input $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$ and we will be using the following attention model:

$$\mathbf{a}^{(t)} = \text{softmax}\left(\mathbf{W}^{\text{a}}\left[\mathbf{x}^{(t)}; \mathbf{h}^{(t-1)}\right]\right) \tag{3.17}$$

$$\mathbf{c}^{(t)} = \sum_{s} a_s^{(t)} \bar{\mathbf{h}}^{(s)} \tag{3.18}$$

$$\mathbf{h}^{(t)} = \text{ReLU}\left(\mathbf{W}^{\text{b}}\left[\mathbf{x}^{(t)}; \mathbf{h}^{(t-1)}\right]\right) \tag{3.19}$$

where $\bar{\mathbf{h}}^{(s)}$ is encoder hidden state and $\mathbf{h}^{(t)}$ is decoder hidden state.

3.2.8 Layer Normalisation

Training recurrent neural networks can be computationally expensive. Layer normalisation [1] reduces the training time by normalising the activities of neurons in a layer. It also helps stabilising the hidden state dynamics of RNNs and might improve their generalisation.

Layer normalisation is defined as a function $LN : \mathbb{R}^d \to \mathbb{R}^d$ with two vectors of parameters (inferred from data), gain $\boldsymbol{\alpha}$ and bias $\boldsymbol{\beta}$, in the following way:

$$LN(\mathbf{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{\mathbf{z} - \mathbf{m}}{s} \odot \boldsymbol{\alpha} + \boldsymbol{\beta} \tag{3.20}$$

where $m = \frac{1}{d}\sum_i z_i$, $\mathbf{m} = (m, m, \dots, m)^T$ and $s = \sqrt{\frac{1}{d}\sum_i (z_i - m)^2}$.

Applying layer normalisation to LSTMs replaces equations 3.7, 3.8, 3.9 and 3.12 with equations 3.21, 3.22, 3.23 and 3.24, respectively.

$$\mathbf{i}^{(t)} = \sigma\left(LN\left(\mathbf{W}^{\text{xi}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xi}}, \boldsymbol{\beta}^{\text{xi}}\right) + LN\left(\mathbf{W}^{\text{hi}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{hi}}, \boldsymbol{\beta}^{\text{hi}}\right) + \mathbf{b}^{\text{i}}\right) \tag{3.21}$$

$$\mathbf{f}^{(t)} = \sigma\left(LN\left(\mathbf{W}^{\text{xf}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xf}}, \boldsymbol{\beta}^{\text{xf}}\right) + LN\left(\mathbf{W}^{\text{hf}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{hf}}, \boldsymbol{\beta}^{\text{hf}}\right) + \mathbf{b}^{\text{f}}\right) \tag{3.22}$$

$$\mathbf{o}^{(t)} = \sigma\left(LN\left(\mathbf{W}^{\text{xo}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xo}}, \boldsymbol{\beta}^{\text{xo}}\right) + LN\left(\mathbf{W}^{\text{ho}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{ho}}, \boldsymbol{\beta}^{\text{ho}}\right) + \mathbf{b}^{\text{o}}\right) \tag{3.23}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh\left(LN\left(\mathbf{c}^{(t)}; \boldsymbol{\alpha}^{\text{h}}, \boldsymbol{\beta}^{\text{h}}\right)\right) \tag{3.24}$$

Similarly, applying layer normalisation to GRUs replaces equations 3.13, 3.14 and 3.15 with equations 3.25, 3.26 and 3.27, respectively.

$$\mathbf{r}^{(t)} = \sigma\left(LN\left(\mathbf{W}^{\text{xr}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xr}}, \boldsymbol{\beta}^{\text{xr}}\right) + LN\left(\mathbf{W}^{\text{hr}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{hr}}, \boldsymbol{\beta}^{\text{hr}}\right) + \mathbf{b}^{\text{r}}\right) \tag{3.25}$$

$$\mathbf{z}^{(t)} = \sigma\left(LN\left(\mathbf{W}^{\text{xz}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xz}}, \boldsymbol{\beta}^{\text{xz}}\right) + LN\left(\mathbf{W}^{\text{hz}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{hz}}, \boldsymbol{\beta}^{\text{hz}}\right) + \mathbf{b}^{\text{z}}\right) \tag{3.26}$$

$$\mathbf{n}^{(t)} = \tanh\left(LN\left(\mathbf{W}^{\text{xn}}\,\mathbf{x}^{(t)}; \boldsymbol{\alpha}^{\text{xn}}, \boldsymbol{\beta}^{\text{xn}}\right) + \mathbf{r}^{(t)} \odot LN\left(\mathbf{W}^{\text{hn}}\,\mathbf{h}^{(t-1)}; \boldsymbol{\alpha}^{\text{hn}}, \boldsymbol{\beta}^{\text{hn}}\right) + \mathbf{b}^{\text{n}}\right) \tag{3.27}$$

Chapter 4

Experiments and Results

In this chapter we use RNNs for deciding a particular class of LT_k languages. In first section we describe the data that we will be working with. Then we define an architecture that we will be using throughout the chapter. We start by showing how RNNs fail to generalise and then experiment with a differentiable version of scanner from Chapter 3. All experiments will be carried out using PyTorch neural network library [22].

4.1 Data

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 models which achieve high accuracy on the test set will capture the semantics behind 3-factors.

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 an SL_k definition 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. This corresponds to the k-expression specified by equation 3.1.

4.2 Methods

Similarly to sequence to sequence models (see 3.2.6) we will be using two statistical models called encoder and decider. An encoder takes SL_k definition as input and outputs the context vector $\mathbf{c}$ 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.

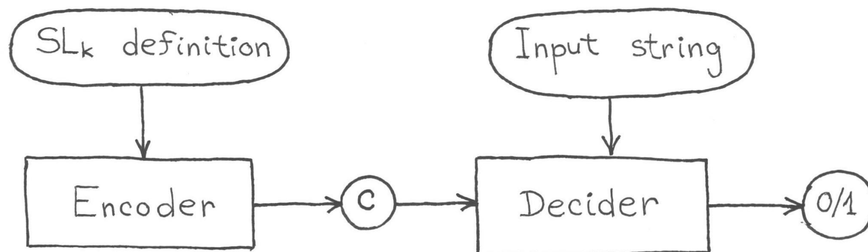


Figure 4.1: Encoder-decider architecture.

In the following, encoder will always be an RNN. 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 [15].

4.3 RNN Decider

In this section both encoder and decider will be RNNs with the same number of layers and hidden units per layer. 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. For each choice of hyperparameters we perform 10 different runs with 50 epochs

each. The maximal test accuracy over all runs and epochs on FACTOR1 and FACTOR5 are reported in tables 4.1 and 4.2.

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: Results on FACTOR1

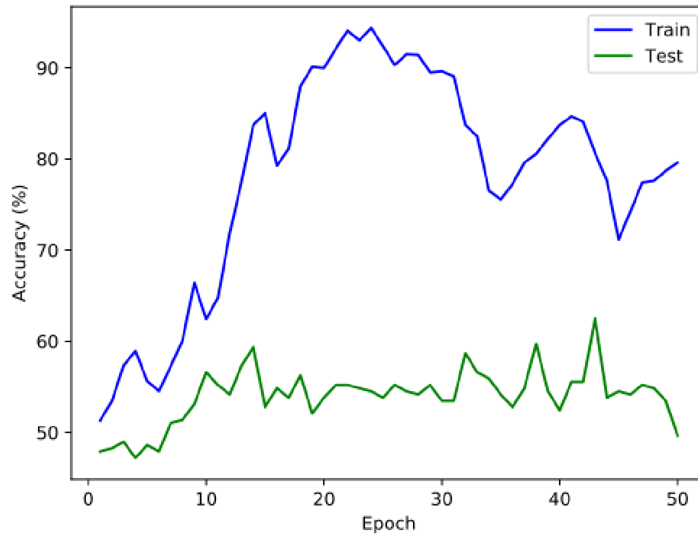


Figure 4.2: The evoultion of train and test accuracy of model for which the maximum is achieved (two layer GRU with 100 hidden units). Applying layer normalisation to this model did not improve the value reported in the table above.

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: Results on FACTOR5

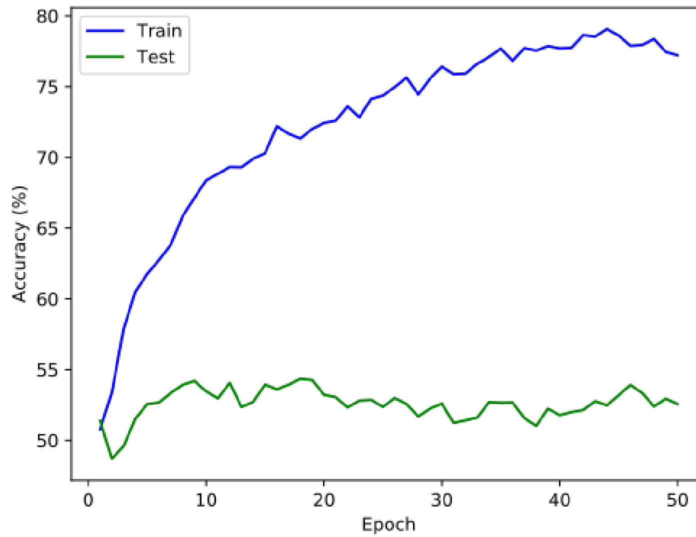


Figure 4.3: The evoultion of train and test accuracy of model for which the maximum is achieved (single layer GRU with 100 hidden units). Applying layer normalisation to this model did not improve the value reported in the table above.

We see that RNNs fail to generalise to unseen languages.

4.4 Adding Attention

Now we add the attention as described in 3.2.7 to the models studied in the previous section. Again, the maximal test accuracy over all the runs and epochs is taken. The results are displayed in Table 4.3 and Table 4.4.

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: Results on FACTOR1

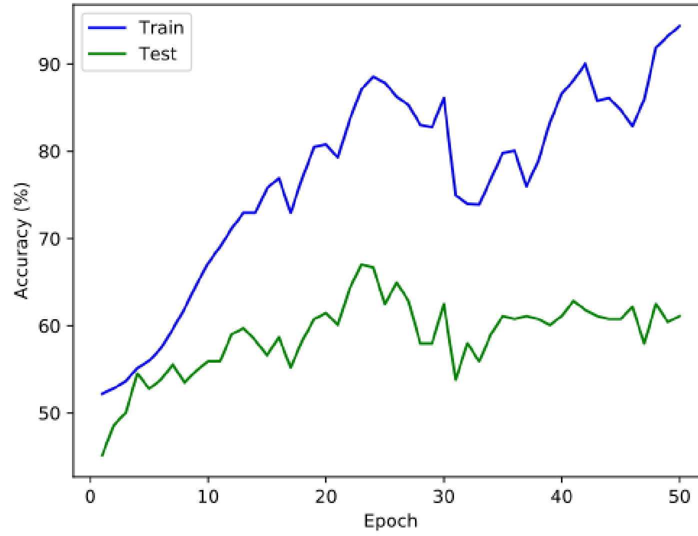


Figure 4.4: The evoultion of train and test accuracy of model for which the maximum is achieved (single layer GRU with 20 hidden units). Applying layer normalisation to this model did not improve the value reported in the table above.

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: Results on FACTOR5

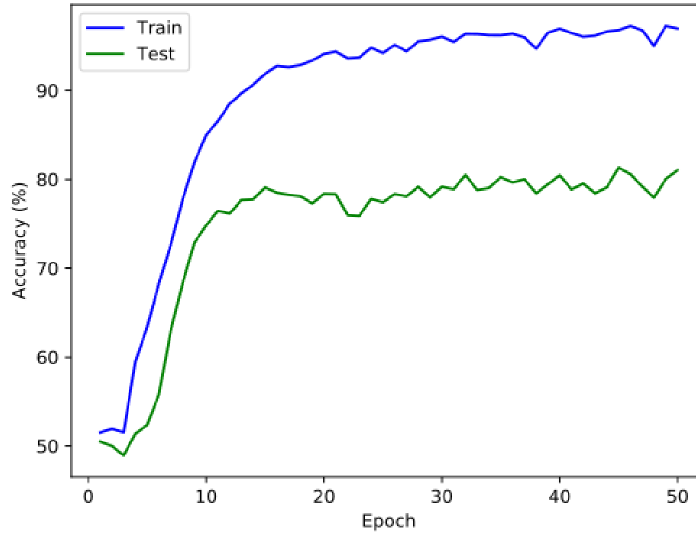


Figure 4.5: The evoultion of train and test accuracy of model for which the maximum is achieved (single layer GRU with 20 hidden units). Applying layer normalisation to this model did not improve the value reported in the table above.

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%. Therefore, RNNs fail to capture the semantics behind k-factors.

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. Note that such a structure is sufficient for deciding languages that we will be working with.

Suppose that we are given a k-locally testable language specified by F k-factors. The scanner takes a context vector $\mathbf{c}$ and splits it into F parts of equal size $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_F$. Then it takes the encodings of k-factors from the input string and uses dot product to check if there is a match between one of them and one of $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_F$. The maximum over all dot products is taken, fed to a linear layer and the result is passed to sigmoid activation. Last three steps corresponds to Boolean network implementing k-expression specified by 3.1.

The algorithm below formally describes such a scanner.

Algorithm: Scanner

Input: context vector $\mathbf{c}$, string x of length n

1 Split context vector $\mathbf{c}$ into F parts of equal size $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_F$.
2 **for** $i = 1$ **to** $n - k + 1$ **do**
3 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.
4 $m_{ij} \leftarrow \langle \bar{\mathbf{x}}_i, \mathbf{c}_j \rangle$
5 **end**
6 $m \leftarrow \max_{i,j} m_{ij}$
7 **return** $\sigma(am + b)$

Parameters a and b are inferred from the data.

To verify that the proposed model works correctly, we hardcode the construction of a look-up table. We do this by taking the encodings of 3-factors and concatenating them to obtain the context vector $\mathbf{c}$ that we pass to the scanner. For an example, see Figure 4.6. On both Factor1 and Factor5 we obtain the test accuracy of 100%.

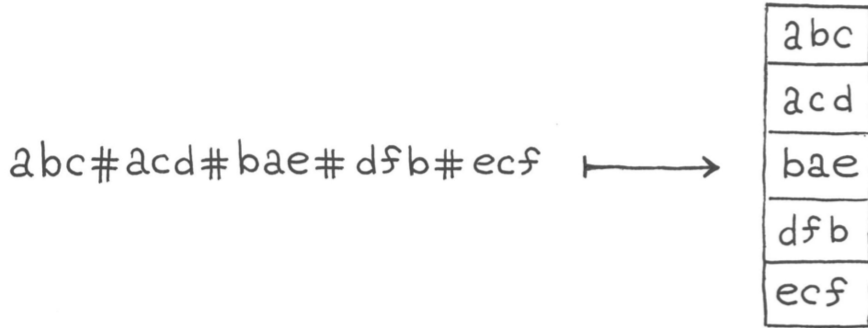


Figure 4.6: Hardcoding look-up table construction. Here by abc we mean the encoding of 3-factor "abc" (i.e. encodings of "a", "b" and "c" concatenated in a single vector). Look-up table is constructed by concatenating the encodings of 3-factors which in this case correspond to $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_5$. If the dimension of symbol encoding is equal to d, then the dimension of whole look-up table is equal to $15d$ (five strings with three symbols each).

Therefore, the learning task reduces to learning to construct a look-up table from SL_k definition.

4.5.1 Last Hidden State of Encoder as Look-Up Table

Here we will use only the last 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 runs for every embedding dimension $d \in \{5, 10, 20\}$ and learning rate $\eta \in \{0.01, 0.001\}$. We take values of d and η for which the test accuracy is the highest on average across all 100 runs; in this case $d = 10$ and $\eta = 0.01$. We only consider 52 runs for which the training accuracy was at least 95% and plot the distribution of the corresponding test accuracy. The results are displayed in Figure 4.7.

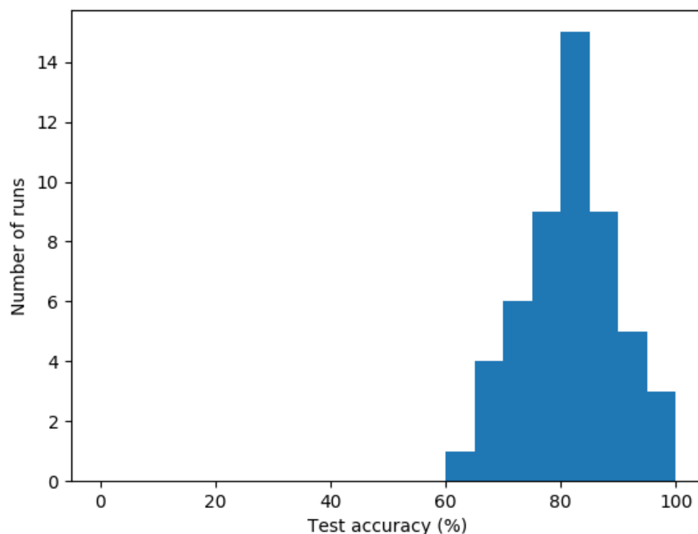


Figure 4.7: Test accuracy distribution of 52 runs that achieved at least 95% on the training subset of FACTOR1. We observe a bell shaped histogram centered around 83%.

As we find that test accuracy distribution differs only slightly between different values of d and η, we take the same values that we used on FACTOR1 (i.e. $d = 10$ and $\eta = 0.01$) and use them on FACTOR5. We perform 50 runs with 200 epochs each. We only consider 17 runs for which the training accuracy was at least 95% and plot the distribution of the corresponding test accuracy. Unfortunately, none of these runs generalises to unseen languages. The results are displayed in Figure 4.8.

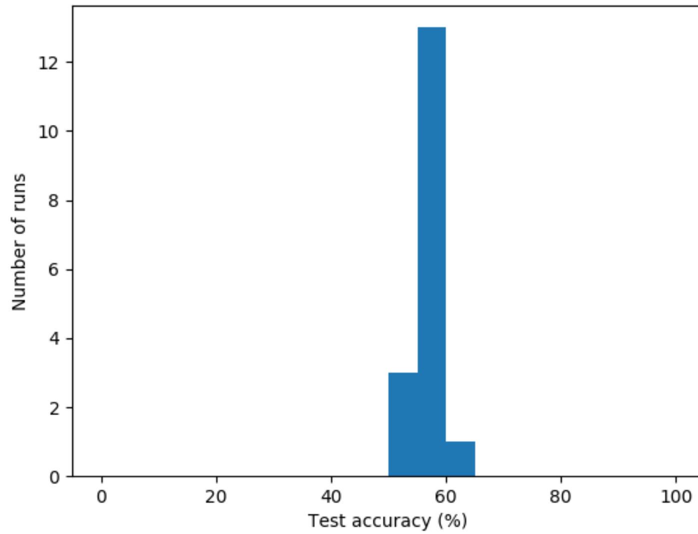


Figure 4.8: Test accuracy distribution of 17 runs that achieved at least 95% on the training subset of FACTOR5. No model generalises to unseen languages.

4.5.2 Can RNNs Store k-Factors?

Here we show that RNN is able to learn to store sufficiently similar copies of k-factors in its memory. We do this in the following way: we feed the encodings of symbols in SL_k definition to the RNN. We concatenate the hidden states corresponding to processing the symbols of 3-factors (ignoring hidden states corresponding to "#") and obtain the context vector which we pass to the scanner. For an example, see Figure 4.9.

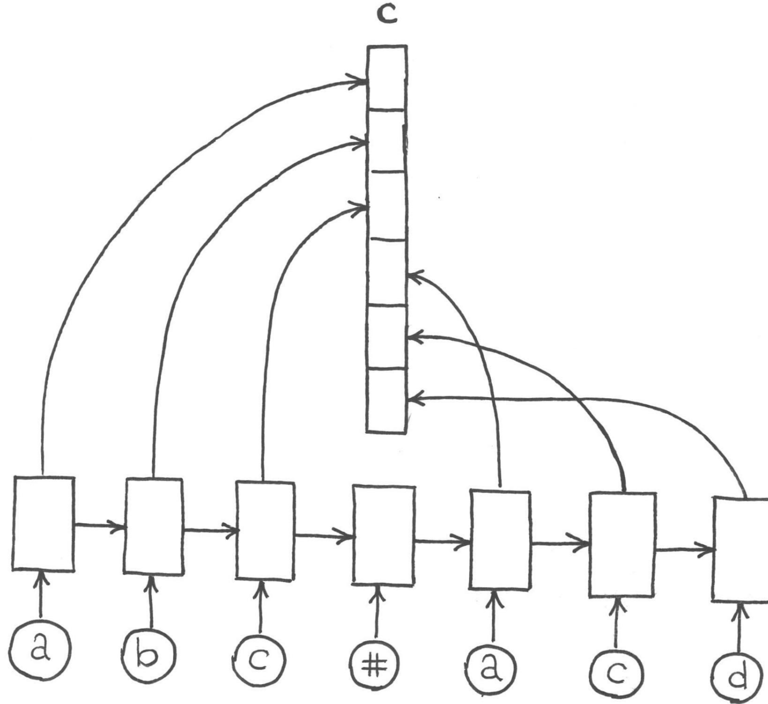


Figure 4.9: The encodings of symbols of 3-factors are processed by an RNN and the corresponding hidden states are concatenated to construct a context vector **c**. If the model recognises 3-factors correctly, then **c** will be a perfect look-up table.

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 perform 100 runs for every embedding dimension $d \in \{5, 10, 20\}$ and learning rate $\eta \in \{0.01, 0.001\}$. We take values of d and η for which the test accuracy is the highest on average across all 100 runs; in this case $d = 10$ and $\eta = 0.01$. Out of 100 runs, 50 of them achieved 100% accuracy on both training and test subset. The other 50 achieved less than 75% accuracy on the training subset, so we will not consider them.

For FACTOR5 we take the same values of d and η (i.e. $d = 10$ and $\eta = 0.01$) and perform 50 runs with 200 epochs each. Out of 50 runs, 21 of them achieved 100% accuracy on both training and test subset. The other 29 achieved less than 65% accuracy on the training subset, so we will not consider them.

We see that models that achieve 100% accuracy on training set necessarily generalise to unseen languages. We conclude that LSTMs are capable of storing k-factors in their memory.

4.5.3 Using Positional Encodings

One of the problems of approach used in 4.5.1 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. For example, if RNN stores the information from processing the first symbol in first d hidden units, then it should store the information from processing the second symbol in second d hidden units, and so on. Therefore, the information about the position of symbol in a definition would be useful for RNN.

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. We use additional embedding layer also of size d for encoding positions.

On FACTOR1 we train the model for 100 epochs. Again, the test accuracy highly depends on initial values of parameters. We perform 100 runs for every embedding dimension $d \in \{5, 10, 20\}$ and learning rate $\eta \in \{0.01, 0.001\}$. We take values of d and η for which the test accuracy is the highest on average across all 100 runs; in this case $d = 20$ and $\eta = 0.01$. We only consider the 54 runs for which the training accuracy was at least 95% and plot the distribution of corresponding test accuracy. The results are displayed in Figure 4.10.

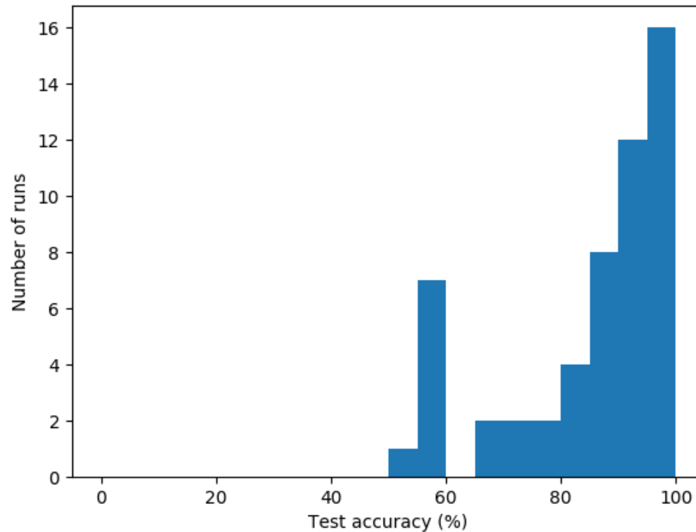


Figure 4.10: Test accuracy distribution of 52 runs that achieved at least 95% on the training subset of FACTOR1. Many of them generalise to unseen languages.

In comparison with 4.5.1we observe improvement in generalisation.

As we find that test accuracy distribution differs only slightly between different values of d and η, we take the same values that we used on FACTOR1 (i.e. $d = 20$ and $\eta = 0.01$) and use them on FACTOR5. We perform 50 runs with 200 epochs each. We only consider 14 runs for which the training accuracy was at least 95% and plot the distribution of the corresponding test accuracy. Unfortunately none of the runs generalises to unseen languages. The results are displayed in Figure 4.11.

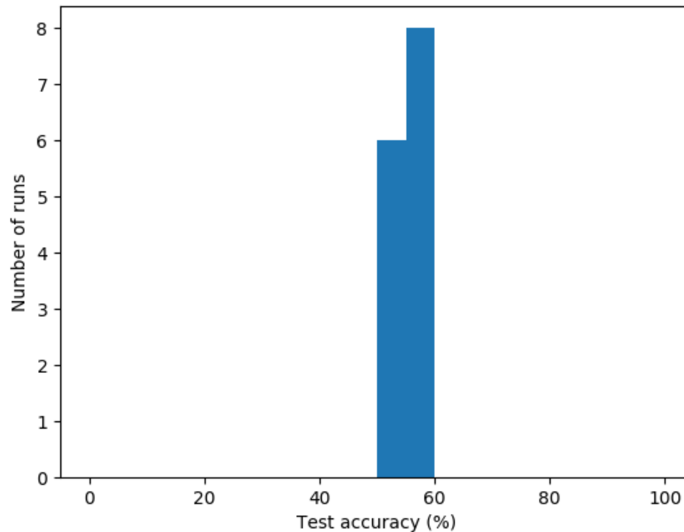


Figure 4.11: Test accuracy distribution of 17 runs that achieved at least 95% on the training subset of FACTOR5. No model generalises to unseen languages.

4.5.4 Supervising LSTM Gates

Here we force the LSTM with positional encodings 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 of hidden units, second symbol in the second third of hidden units and third symbol in last third of hidden units. This is achieved by setting the first third of input gate activations to be close to 1 and rest close to 0 on the first time step; second third of input gate activations to be close to 1 and rest close to 0 on the second time step; last third of input gate activations to be close to 1 and rest close to 0 on the third time step. Also, for forget gates we want the first third activations to be close to 1 on second time step; first and second third activations to be close to 1 on third time step (so that it remembers the first two symbols). The idea is naturally extended for more than one 3-factor. We will use cross entropy to measure the error in gate activations. The loss associated with

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 perform 100 runs for every embedding dimension $d \in \{5, 10, 20\}$, learning rate $\eta \in \{0.01, 0.001\}$ and $\gamma \in \{2, 1, 0.5, 0.1\}$. We take values of d, η and γ for which the test accuracy is the highest on average across all 100 runs; in this case $d = 5$, $\eta = 0.01$ and $\gamma = 0.1$. We only consider 51 runs for which the training accuracy was at least 95% and plot the distribution of the corresponding test accuracy. The results are displayed in Figure 4.12.

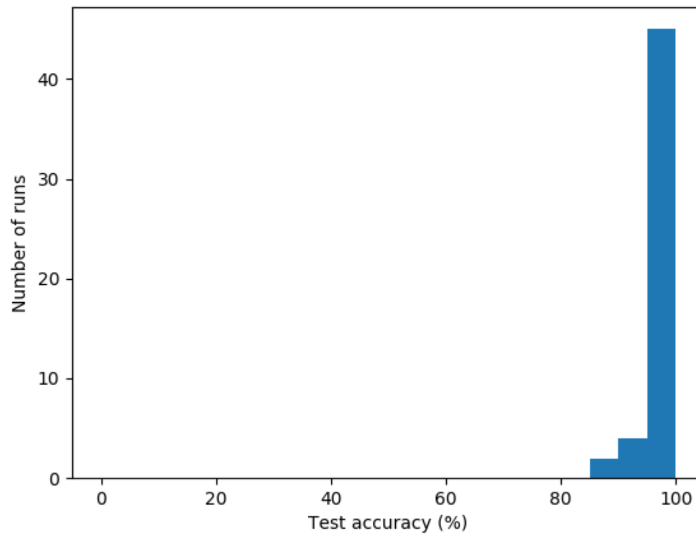


Figure 4.12: Test accuracy distribution of 51 runs that achieved at least 95% on the training subset of Factor1. Most of them generalise to unseen languages.

Therefore, supervising the gate activations further improves generalisation.

To convince ourselves that the model has learned to use gates in a desired way we plot the gate activations in Figure 4.13.

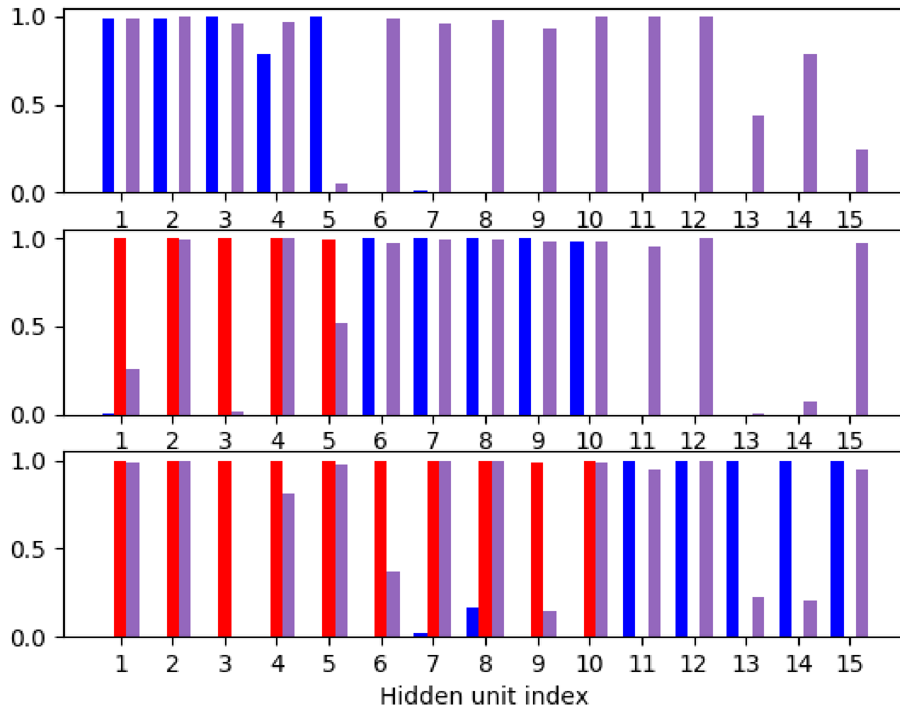


Figure 4.13: Gate activations on FACTOR1 dataset. Blue bars denote input gates, red bars forget gates and purple bars output gates. The uppermost plot corresponds to $t = 1$. We see that model learned to store the first symbol in first 5 hidden units. Second plot corresponds to $t = 2$. We see that model learned to store the second symbol in second 5 hidden units, while remembering the first one in first hidden units. Last plot corresponds to $t = 3$. We see that model learned to store the thirds symbol in third 5 hidden units while remembering the first two in first 10 hidden units.

As we find that test accuracy distribution differs only slightly between different values of d, η and γ, we take the same values that we used on FACTOR1 (i.e. $d = 5$, $\eta = 0.01$ and $\gamma = 0.1$) and use them on FACTOR5. We perform 50 runs with 200 epochs each. Only 2 out of 50 runs obtain train accuracy higher than 95%. Unfortunately, neither of them gets beyond 60% accuracy on the test subset.

Chapter 5

Conclusion and Future Work

In this chapter we summarise some of the obtained results and propose some future directions.

In 4.2 we defined an encoder-decider architecture for deciding formal languages. We used RNNs for both encoder and decider and saw that they fail to generalise to unseen languages. While adding attention improves the test accuracy, in principle RNNs do not capture the semantics of k-factors.

Then we implemented a differentiable version of scanner and used it as a decider while keeping encoder to be an RNN. In that way we reduced the task of deciding a language to the task of constructing a look-up table. We saw that LSTM is able to store k-factors in its memory, but has trouble arranging five of them. We used positional encodings and supervised the gates of the LSTM to aid it in construction of a look-up table. While this improved the generalisation on FACTOR1, none of this worked for FACTOR5 dataset.

Therefore, the problem we considered is far from solved.

5.1 Register RNNs

Here we propose another model that might be useful for our task. Instead of using a single hidden state, we use a memory consisting of several registers. At each time step, a model selects one of the registers where to write and then treat it as a hidden state of an RNN.

However the selection procedure is not differentiable and therefore has to be trained using techniques like reinforcement learning [28] or Gumbel softmax [13]. Another issue is, whether this model would be applicable to any other problem than the one we are considering here.

Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[3] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. URL: http://www.aclweb.org/anthology/D14-1179.

[4] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[5] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (2000), pp. 2451–2471.

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

[7] Alex Graves et al. "A novel connectionist system for unconstrained handwriting recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 31.5 (2009), pp. 855–868.

[8] Klaus Greff et al. "LSTM: A search space odyssey". In: *IEEE transactions on neural networks and learning systems* 28.10 (2017), pp. 2222–2232.

[9] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: *Diploma, Technische Universität München* 91.1 (1991).

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

[11] 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.

[12] Gerhard Jäger and James Rogers. "Formal language theory: refining the Chomsky hierarchy". In: *Philosophical Transactions of the Royal Society of London B: Biological Sciences* 367.1598 (2012), pp. 1956–1970.

[13] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical reparameterization with gumbel-softmax". In: *arXiv preprint arXiv:1611.01144* (2016).

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

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

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

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

[18] Minh-Thang Luong et al. "Addressing the rare word problem in neural machine translation". In: *arXiv preprint arXiv:1410.8206* (2014).

[19] 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.

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

[21] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*. 2013, pp. 1310–1318.

[22] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[23] 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.

[24] Haşim Sak, Andrew Senior, and Françoise Beaufays. "Long short-term memory recurrent neural network architectures for large scale acoustic modeling". In: *Fifteenth annual conference of the international speech communication association*. 2014.

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

[26] Hava T Siegelmann and Eduardo D Sontag. “Turing computability with neural nets”. In: *Applied Mathematics Letters* 4.6 (1991), pp. 77–80.

[27] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems.* 2014, pp. 3104–3112.

[28] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction.* 1998.

[29] 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.

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

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