Recurrent neural networks for language modeling

Emil Sauer Lynge

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of the thesis is to explore the mechanisms and tools that enables efficient development of Recurrent Neural Networks, how to train them and what they can accomplish in regard to character level language modelling. Specifically Gated Recurrence Units and Long Short Term Memory are the focal point of the training and language modelling. Choice of data sets, hyper parameters and visualization methods, aims to reproduce parts of [KJL15]. More broadly RNN as a concept is explored through computational graphs and back propagation. Several concrete software tools written in python 3 is developed as part of the project, and discussed briefly in the thesis.

Summary (Danish)

Målet for denne afhandling er en udforskning af de mekanismer og værtøjer der muliggør efficient udvikling af Recurrent Neurale Netværk, hvorledes de optimeres og hvad de bibringer til sprogmodellering på bogstavniveau. Gated Recurrence Units and Long Short Term Memory er de specifikke RNN implementeringer, som optimering og sprogmodellering baserer sig på i afhandlingen. Valg af hyperparametre, data sæt og visualisering, sigter efter at reproducere dele af resultaterne fra [KJL15]. RNN som mere abstract koncept udforskes gennem beregningsgrafer og back propagation. I forbindelse med projektet er udviklet en håndfuld software værktøjer skrevet i Python 3 som kort beskrives i afhandlingen.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with recurrent neural networks, their architectures, training and application in character level language modelling.

The thesis consists of a detailed introduction to neural network python libraries, an extensive training suite encompassing LSTM and GRU networks and examples of what the resulting models can accomplish.

Lyngby, 19-02-2016

Emil Sauer Lynge

Acknowledgements

I would like to thank my advisor Ole Winther, whose patience, dedication and undeterred belief in me, has been absolutely paramount for the successful completion of this thesis. I would also like to thank friends, family and especially my partner Hilde for bearing the responsibilities I may have inadvertently shed from my shoulders during the most intense parts of this project.

I am grateful, and feel that I have been the recipient of a wholly undeserved amount of kindness.

Contents

Chapter 1

Introduction

Many real world problems face the issue of combining very localized prediction with contextual information. The next character in a sequence is heavily dependent on the immediately preceding characters, and on this input alone we can expect to robustly predict syllables, and maybe even whole words. But without *contextual* information we cannot expect these words or syllables to be coherent in any way. Context here meaning the position in a parapragh, whether the language is currently in first or third person, the tense of the sentence etc.

Providing a model with such contextual information, often takes the form of an n-gram, where data from previous time steps are included explicitly by using a sliding window, wherein several data points are concatenated into a single feature vector. This can quickly become a very computationally expensive way of expanding the temporal scope, as model training effort usually grow super linearly with length of the feature vector. Other methods might seek to transform previous data points into a lower dimensional space in order to fit a longer time window into fewer features. This however still leaves the problem of explicitly stating how far back relevant data might be found.

Recurrent Neural Networks introduces a way to draw on information, in principle, indefinitely far back in the past without an explicit sliding window. By recursively carrying over information from a hidden layer to itself at the next time step, a hidden state is maintained that can provide contextual information. This works as a fuzzy window into the past, where the manner and extent to which hidden states are maintained, defines the size of said window. Weights between hidden layers are the parameters that defines this behaviour and they are *trainable* rather than *tuneable*. In essence this means that stating what type of information to retain and for how long it should be retained is for the model to decide and only the complexity is left up to tuning.

RNNs however, are very dependent on the architecture of the implementation, which changes performance characteristics and model complexity. As early as 1997 new architectures were shown to vastly outperform the core RNN[HS97], with faster convergence of training outweighing the difficulties of complexly connected networks. On the other hand simplified architectures has also been shown, to some times provide the behaviour and performance of more complex counterparts, as in the case of GRU and LSTM[Chu+14][KJL15]. Improving RNN architectures is therefore less a case of introducing more parameters, and more recognizing which connections serve which purposes, and how to provide effective paths for errors to propagate to relevant parameters. Doing this requires an understanding of the inner mechanics of the network and the means to quickly, and with out side effects, play around with network connections.

With new tools and frameworks like Theano[Bas+12] and Lasagne [Die+15] it's increasingly feasible to generate a host of complex models. These tools hides the underlying complexity greatly, making RNN widely accessible. At the same time, complexity hidden may obscure crucial mechanics leaving a programmer with even less understanding of how to effectively implement a RNN. This leaves an interesting combination of apparent accessibility and actual required user skill.

1.1 Problem Statement

Recurrent Neural Networks is a very large family of models of varying complexity. They have very interesting character level prediction features as explored in detail in [KJL15]. I will reproduce some of these result documenting that my network implementation yields comparable results. Furthermore, I will explore the building blocks and training process required to create well performing RNNs. Lastly I will present a single-file program that showcase how such a network could be applied in real word applications.

CHAPTER 2

Data

To make any results comparable to the ones in [KJL15] I will be using the same two main data sources. They are simple text files that has been cleaned by Karparthy [Kar].

2.1 War and Peace

Novel by Leo Tolstoy with very poetic and distinct language. Abbreviated as WP.

Summary	
Pages	∼ 1,225
Line count	∼ 62,000
Word count	∼ 560,000
Characters	∼ 3,250,000
vocabulary	83

With more than 1000 pages this is a very long book, and as such is one of the more feasible targets for such a complex model that needs a lot of data to avoid overfitting. Surprisingly the novel scores relatively high in readability measures

[1], meaning that it should be a rather easy read. In general this means words can be expected to have few letters per word and few words per sentence.
By visual "inspection" it is also found that the author makes heavy use of direct speech. Direct speech is interesting as it has some very structured components:

- Starts and ends with quotation marks which have very little use outside direct speech and should be quite a robust delimiter.
- Tense will tend to be in present.
- Certain pronouns will be much more likely than usual eg. I, you, me etc.

Especially newline quotation marks and the likes were reported by [KJL15] to be well predicted by LSTM.

It is important to note that the vocabulary size of 83 characters diverges from the 87 characters reported in [KJL15]. This is primarily because I have chosen to decode the characters rather than reading them in as raw bytes. The text file is UTF8 encoded, which means that non sc ASCII characters will take up two or more bytes per character, due to a special scheme where the first bits are used to encode the number of bytes the character consists off. In the text this applies to the following characters:

	UTF code point	bytes	occurrences
à	`U+00E0`	`c3 a0`	4
ä	`U+00E4`	`c3 a4`	1
é	`U+00E9`	`c3 a9`	1
ê	`U+00EA`	`c3 aa`	11
`BOM`	`U+FEFF`	`ef bb bf`	1

The byte `c3` in 4 first special characters recurs simply because they are close together in the character table. The `BOM` character is a special character that can occur as the very first character in a stream.
All in all these 5 characters represent 8 unique bytes thus accounting for 3 bytes "missing" in my vocabulary leaving 1 unaccounted for. The missing byte is the carriage return character `\r` which signals that enter has been pressed and the line should be "reset". In the data set this character **always** precedes the newline character `\n`. My implementation has been written in Python 3 which has default newline conversion that translates such `\r\n` byte pairs into a single `\n` newline character. I have chosen to keep this default behaviour for two reasons

[1]Checked by pasing text in here: http://www.editcentral.com/gwt1/EditCentral.html

- Their use in the text is effectively as a single character and treating them as two separate characters, would artificially increase model performance.
- A lot of the models had already been fully trained by the time I discovered what this "missing" byte was caused by. Rather than scrap those models along with the days and weeks of GPU time that trained them, it will just have to be accounted for in my analysis.

2.2 Linux Kernel

Source code written in C from the linux project. Abbreviated as LK.

Summary	
Functions	$\sim$ 8700
Variables[2]	$\sim$ 5000
Line count	$\sim$ 241,000
Word count	$\sim$ 760,000
Characters	$\sim$ 6,200,000
vocabulary	101

The data is a concatenation of all header and source files that constitutes the linux *kernel*. The kernel is the core of an operating system and handles access to the hardware such as CPUs, RAM, disks and other devices. The language features variable declaration, ";" to delimit statements and "" to encapsulate scopes which makes the text very structured. Additionally, the code is well indented and follows common C guidelines for formatting. Apart from raw code, the data set includes a lot of inline comments in natural language such as function documentation and license information.

[2]The number of *unique* variable names

CHAPTER 3

Theory

3.1 Nodes in networks

Visualization of networks and tools for computing their output and gradients, relies on an approach that seeks to map out any mathematical expression as a graph. This section is a short introduction to the terminology I will be using, how to decode computation graphs and what concrete tools can be used for node-centric computing.
This section assumes that the reader is familiar with the terms from basic graph theory, especially relating from directed graphs. A non-exhaustive graph glossary can be found in A.2.1.

3.1.1 Terminology

To discuss the architecture of RNNs I will use a node centered terminology, in which all parts of the network is represented as a node in a graph. On a broad level I will be referring to nodes as belonging to one of 3 classes:

- *Static* All leaf nodes in the graph. i.e. nodes for which the output is static and does not depend on an input. Note that this also includes weight matrices and other parameters as well as the raw data inputs.

- *Operator* All nodes that has both input and output. This type of node occurs every time some data is manipulated. The number of inputs depends on the operator type. e.g sigmoid transform takes only one input, matrix multiplication takes 2 inputs exactly and sum takes an arbitrary number of inputs.
- *Cost* An operator node with no parent that produces a scalar value. It takes as inputs all an estimated outputs and a ground truth static node and from these calculates some cost measure.

When discussing the THEANO library in 3.10 this way of looking at computations will become central.

As an illustration of this take the computation

$$\boldsymbol{X}\boldsymbol{W} + b = y \tag{3.1}$$

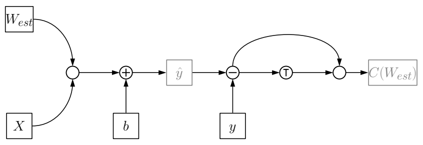
Suppose we have estimated a weight matrix $\boldsymbol{W}_{est}$ with known input $\boldsymbol{X}$, bias b and target value y and want to compute a cost based on the Sum of Squared Errors henceforth *SSE*.

$$C_{SSE}(\boldsymbol{W}_{est}) = (\boldsymbol{y}_{est} - y)^T (\boldsymbol{y}_{est} - y) \tag{3.2}$$

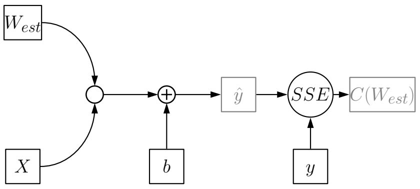
This can be computed by laying it out as a graph with 4 *static* input nodes representing $\boldsymbol{W}_{est}$, $\boldsymbol{X}$, b and y, and 6 *operator* nodes as seen in figure 3.1(a). The symbolism is as follows:

- Square nodes represents *data* and they **never** manipulate other data. The black squares are *static* nodes whilst light gray are "imaginary" nodes that are simply there to increase readability. They are imaginary because they do not really exist in the computation, but should be seen as labels on the output of an *operator* node.
- Round nodes are *operator* nodes. They have symbols that designate the type of operation they perform on the input. A full node symbol legend can be found in the appendix A.1

The last 3 operator nodes together comprise the SSE computation. The inner workings of such cost metrics is not necessarily relevant and for clarity it can



(a)



(b)

Figure 3.1: Node representation of a computation that calculates the sum of squared errors associated with $\boldsymbol{X}_{est}$

be reduced it to a single *cost* node in the graph as done in 3.1(b).

To obtain the resulting cost of the network, values of all *static* nodes are fed forward, by passing output data along the edges until it reaches the node for which we want to know the output. Along the way *operator* nodes may hold back inflowing data until all its parent nodes have passed along their output. This operation is called a *forward pass*, although in practice an implementation may very well start of traversing the graph backwards. The key part is that the data will always flow forwards in the graph.
In fact it is very convenient to implement the forwards pass by calling back through the graph recursively as seen in snippet 3.1.

3.1.2 Why nodes?

At first glance this node approach can seem overly convoluted, but it provides some important concepts that become very helpful when dealing with complex computation such as RNNs. I will very shortly layout the reasoning for this approach and in later sections go into detail with certain aspects.

```python
# node1.py
class ForwardNode:
    def __init__(self, *input_nodes):
        self.output = None
        self.input_nodes = input_nodes

    @property
    def in_values(self):
        return [inp for inp in self.input_nodes.forward()]

    def forward(self):
        if self.output is None:
            self.output = self.op(*self.in_values)
        return output
```

Snippet 3.1: Minimal node class that implements forward pass. `Node.op` is an abstract method that returns the output of node operation which any concrete implementation should define.

3.1.2.1 Abstraction

Calling something a node is akin to defining a function, and in doing so labelling a sequence of computations under a single name. In the case of the + node from figure 3.1 there is just the one computation in the sequence. But by treating a whole subcomponent as a composite node, as seen with the cost metric in 3.1b, we can hide several computations inside one node. The use of functions partly serves to avoid code-repetition, but more importantly it hides the underlying complexity allowing the programmer to think in abstract terms and focus on how concepts interact. In much the same way the abstract term of *cost node* allows for a generalized computation network, that illustrates how data interacts with the cost metric, without the concrete cost implementation cluttering the graph.
It is important to realize that nodes cannot just be perceived as glorified functions. The *type* of node indeed represents no more that what could be expressed by a function, but the input and output together with the underlying function is what constitutes the *node*.

Recurrent neural networks has many such complex interactions, both in low level context such as the inner workings of non-linearities, and on a high level, with several hidden states interacting across layers and time. With a node centered approach it is programmatically simple to swap out components and visualize how those components would affect the system.

3.1.3 Backpropagation

The ubiquitous method of training neural networks is by iteratively feeding forward output with given parameters and back-propagating error gradients that are then used for adjusting these. Programmatically defining such a network using node objects that inputs from, and outputs to, other node objects makes it easy to implement a computation of partial derivatives using the chain rule. As long as we can assert that a node object always computes the correct gradient given correct input any combination of interconnected node objects are also assured to compute correct gradients for the network as a whole. We can thus have relatively few so-called *unit tests* for every node class and be guaranteed correct gradients anywhere in any network. Additionally, node objects makes the bookkeeping simple as every object can simply store its own state, references to parents and children etc.

3.2 Automatic differentiation

As mentioned briefly in 3.1.3 calculating the gradient of an error wrt some parameter is the backbone of training neural networks. This section will explain how nodes in a computational network uses the chain rule for derivatives to compute the gradient for any parameter - a method called *Automatic Differentiation* (AD).

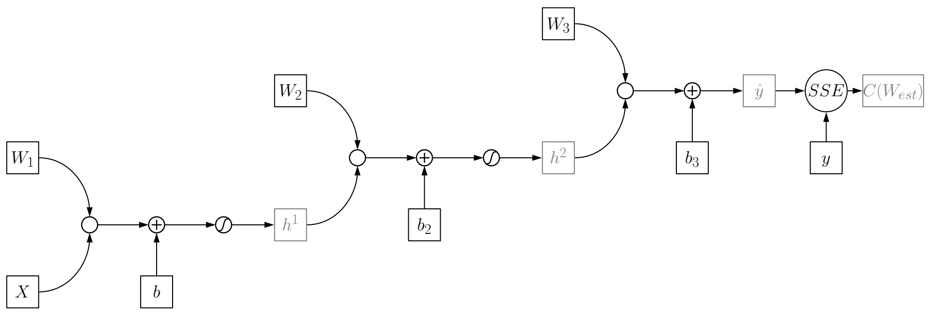
In 3.1.1 the cost is a sum of squared errors of a linear equation and the solution can be derived by Ordinary Least squares Estimation, OLE. In the case of neural network however, non-linearities are introduced that makes solutions space non-convex whereby such clear-cut solutions cannot be derived.

Consider the slightly more complex example in figure 3.2. This is a standard neural network with 2 hidden layers for which the output can be written as:

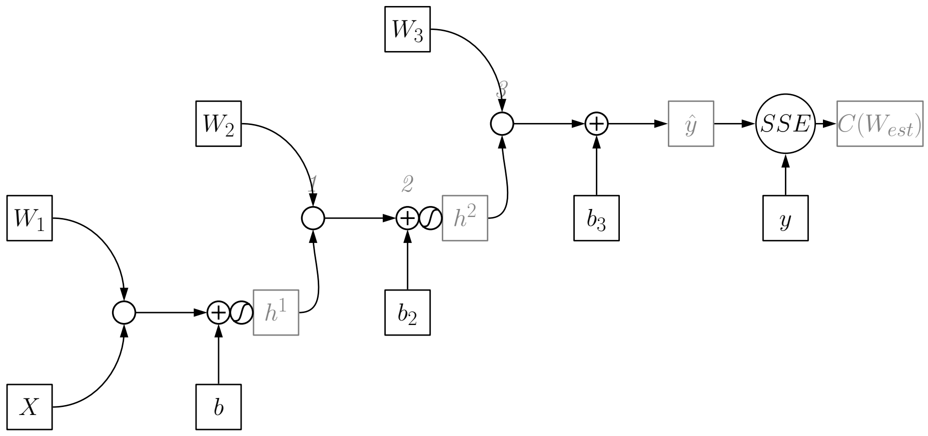
$$\hat{y} = \varphi[\varphi(\boldsymbol{X}\boldsymbol{W}_1 + b_1)\,\boldsymbol{W}_2 + b_2]\,\boldsymbol{W}_3 + b_3 \tag{3.3}$$

Where φ is some non-linear transformation e.g tanh. Given some estimate for the parameters $\boldsymbol{W}_l$ we now want to find some direction that minimizes the SSE cost function.

$$\frac{\partial C}{\partial \boldsymbol{W}_l} = (\hat{y} - y)^T\,(\hat{y} - y) \tag{3.4}$$



(a)



(b)

Figure 3.2: Node representation of a standard neural network with 2 hidden layers. 3.2b represents the same network as 3.2a with a slightly more compact way of showing simple chains of connectivity and has numbering on some, otherwise unnamed, operator nodes.

It is possible to write out the formulas for all parameter gradients and hardcode these into the optimizer program. There are some unfortunate downsides to such an approach:

- Copying errors could become a problem as the formulas may be several lines of code - especially for parameters such as $\boldsymbol{W}_1$ that lies "hidden" beneath 2 nonlinear transformations
- The resulting code would be very difficult to update as even small changes to the overall network architecture would require re-derivation of all formulas
- It might be very inefficient as some expression would appear in several of the formulas. Intermediate results could be stored for reuse, but this might introduce yet more complexity that makes updating even more difficult.

A simpler approach is to exploit the chain rule which states that

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial x} \tag{3.5}$$

Using this we could as an example define the gradient of the cost for wrt $\boldsymbol{W}_2$ as

$$\frac{\partial C}{\partial \boldsymbol{W}_2} = \frac{\partial C}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \boldsymbol{W}_2} \tag{3.6}$$

Which can be expanded to a product over the partial derivatives for every single operation from C to $\boldsymbol{W}_2$

$$= \frac{\partial C}{\partial \hat{y} - y}\frac{\partial \hat{y} - y}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \boldsymbol{h}^2\boldsymbol{W}_3}\frac{\partial \boldsymbol{h}^2\boldsymbol{W}_3}{\partial \boldsymbol{h}^2}\frac{\partial \boldsymbol{h}^2}{\partial \boldsymbol{h}^1\boldsymbol{W}_2 + b_2}\frac{\partial \boldsymbol{h}^1\boldsymbol{W}_2 + b_2}{\partial \boldsymbol{h}^1\boldsymbol{W}_2}\frac{\partial \boldsymbol{h}^1\boldsymbol{W}_2}{\partial \boldsymbol{W}_2} \tag{3.7}$$

Where the partial derivatives are

$$\frac{\partial C}{\partial \hat{y} - y} = \frac{\partial}{\partial \hat{y} - y} \left(\hat{y} - y\right)^T \left(\hat{y} - y\right) = 2\left(\hat{y} - y\right) \tag{3.8}$$

$$\frac{\partial \hat{y} - y}{\partial \hat{y}} = \boldsymbol{I} \tag{3.9}$$

$$\frac{\partial \hat{y}}{\partial \boldsymbol{h}^2 \boldsymbol{W}_3} = \frac{\partial}{\partial \boldsymbol{h}^2 \boldsymbol{W}_3} \boldsymbol{h}^2 \boldsymbol{W}_3 + b_3 = \boldsymbol{I} \tag{3.10}$$

$$\frac{\partial \boldsymbol{h}^2 \boldsymbol{W}_3}{\partial \boldsymbol{h}^2} = \boldsymbol{W}_3 \tag{3.11}$$

$$\frac{\partial \boldsymbol{h}^2}{\partial \boldsymbol{h}^1 \boldsymbol{W}_2 + b_2} = \frac{\partial}{\partial \boldsymbol{h}^1 \boldsymbol{W}_2 + b_2} \varphi\big(\boldsymbol{h}^1 \boldsymbol{W}_2 + b_2\big) = \big(1 - \tan^2\big[\boldsymbol{h}^1 \boldsymbol{W}_2 + b_2\big]\big) \tag{3.12}$$

$$\frac{\partial \boldsymbol{h}^1 \boldsymbol{W}_2 + b_2}{\partial \boldsymbol{h}^1 \boldsymbol{W}_2} = \boldsymbol{I} \tag{3.13}$$

$$\frac{\partial \boldsymbol{h}^1 \boldsymbol{W}_2}{\partial \boldsymbol{W}_2} = \boldsymbol{h}^1 \tag{3.14}$$

The product of these, yields the result (leaving out the ones matrices and vectors)

$$\frac{\partial C}{\partial \boldsymbol{W}_2} = 2\left(\hat{y} - y\right) \boldsymbol{W}_3 \big(1 - \tan^2\big[\boldsymbol{h}^1 \boldsymbol{W}_2 + b_2\big]\big) \boldsymbol{h}^1 \tag{3.15}$$

Additionally the gradient for every node in between $\boldsymbol{W}_2$ and SSE can be written as products between intermediate results and partial derivatives. To clarify the relationship between equations and the computational graph in figure 3.2, any node that is not explicitly named will be represented as N_j where j is the number given to it in the graph.

$$\frac{\partial C}{\partial SSE} = 1 \tag{3.16}$$

$$\frac{\partial C}{\partial \hat{y}} = \frac{\partial C}{\partial SSE} \frac{\partial SSE}{\partial \hat{y}} = 1 \cdot 2\left(\hat{y} - y\right) = 2\left(\hat{y} - y\right) \tag{3.17}$$

$$\frac{\partial C}{\partial N_3} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial N_3} = 2\left(\hat{y} - y\right) \boldsymbol{I} = 2\left(\hat{y} - y\right) \tag{3.18}$$

$$\frac{\partial C}{\partial \boldsymbol{h}^2} = \frac{\partial C}{\partial N_3} \frac{\partial N_3}{\partial \boldsymbol{h}^2} = 2\left(\hat{y} - y\right) \boldsymbol{W}_3 \tag{3.19}$$

$$\frac{\partial C}{\partial N_2} = \frac{\partial C}{\partial \boldsymbol{h}^2} \frac{\partial \boldsymbol{h}^2}{\partial N_2} = 2\left(\hat{y} - y\right) \boldsymbol{W}_3 \big(1 - \tan^2\left[N_2\right]\big) \tag{3.20}$$

$$= 2\left(\hat{y} - y\right) \boldsymbol{W}_3 \big(1 - \tan^2\big[\boldsymbol{h}^1 \boldsymbol{W}_2 + b_2\big]\big) \tag{3.21}$$

Note that $\partial C / \partial SSE = 1$ because SSE *is* the cost function, and computationally this can just be seen as seeding the algorithm.

```
# node2.py
class BackwardNode(ForwardNode):
    def __init__(self, parent, *input_nodes):
        self.output = None
        self.grad = None
        self.input_nodes = input_nodes
        self.parent = parent

    def backwards(self, input_node):
        if self.grad is None:
            self.grad = self.parent.backwards(self)
        wrt_idx = self.input_nodes.index(input_node)
        dout_din = self.pdev(wrt_idx, self.output, *self.in_values)
        return self.grad * dout_din
```

Snippet 3.2: Minimal node class that implements backwards pass. `Node.pdev` is an abstract method that returns the partial derivative of node output wrt some input index, which any concrete implementation should define. Note that this class inherits from `ForwardNode` from snippet 3.1

An algorithm using above procedure does not need to expand the formula as done in (3.21). If the proper bookkeeping is done in the forward pass (See figure 3.1), stored output can simply be used directly without worrying about how that value came to be.
In this way cost gradients can efficiently be calculated for any node, by recursively propagating ${}^{\partial C}/_{\partial SSE}$ backwards through the network. At every node ${}^{\partial N_{out}}/_{\partial N_{in}}$ is computed using stored outputs, and the chain rule is applied by multiplying with ${}^{\partial C}/_{\partial N_{out}}$ received from the parent node to yield ${}^{\partial C}/_{\partial N_{in}}$. A minimal example of such a node object can be seen in figure 3.2.

3.2.1 Dealing with multiple outputs

Whilst the code example in 3.2 will work for the network in figure 3.2 it does so only because no nodes outputs to more than 1 other node. Consider instead the network in 3.3 where the hidden layers share the same weighting matrix. From 3.2 it is known how to compute ${}^{\partial C}/_{\partial N_1}$ and using this value the gradient of the

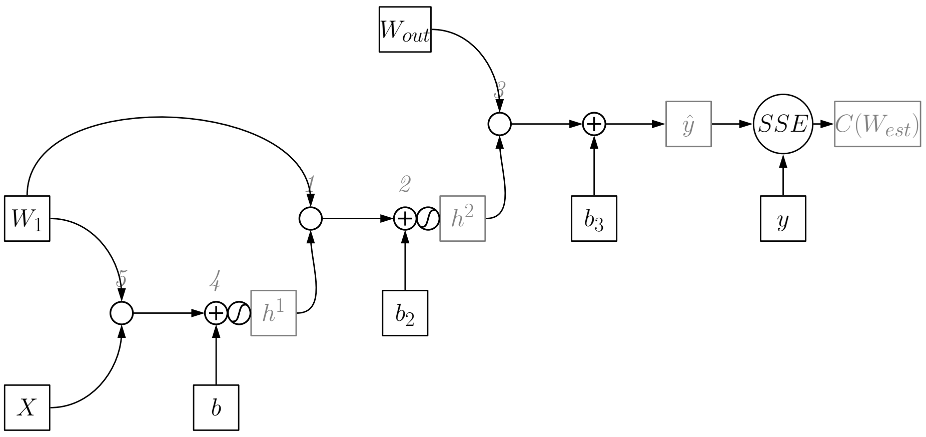


Figure 3.3: Node representation of an unusual neural network with 2 hidden layers. Note reuse of $\boldsymbol{W}_1$ which is generally a bad idea, but illustrates a simple case where a node has several paths to C

cost wrt $\boldsymbol{W}_1$ can be written as

$$\frac{\partial C}{\partial \boldsymbol{W}_1} = \frac{\partial C}{\partial N_1}\frac{\partial N_1}{\partial \boldsymbol{W}_1} \tag{3.22}$$

Where

$$\frac{\partial N_1}{\partial \boldsymbol{W}_1} = \frac{\partial}{\partial \boldsymbol{W}_1}\boldsymbol{W}_1\boldsymbol{h}^1 \tag{3.23}$$

Remember that $\boldsymbol{h}^1$ is a function of $\boldsymbol{W}_1$ so the product rule applies

$$\frac{\partial N_1}{\partial \boldsymbol{W}_1} = \left(\frac{\partial}{\partial \boldsymbol{W}_1}\boldsymbol{W}_1\right)\boldsymbol{h}^1 + \boldsymbol{W}_1\left(\frac{\partial}{\partial \boldsymbol{W}_1}\boldsymbol{h}^1\right) \tag{3.24}$$

Where

$$\frac{\partial}{\partial \boldsymbol{W}_1}\boldsymbol{h}^1 = \frac{\partial \boldsymbol{h}^1}{\partial \boldsymbol{W}_1} \tag{3.25}$$

$$= \frac{\partial \boldsymbol{h}^1}{\partial N_4}\frac{\partial N_4}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1} \tag{3.26}$$

Combining equations 3.22, 3.24 and 3.26

$$\frac{\partial C}{\partial \boldsymbol{W}_1} = \frac{\partial C}{\partial N_1}\left[\left(\frac{\partial}{\partial \boldsymbol{W}_1}\boldsymbol{W}_1\right)\boldsymbol{h}^1 + \boldsymbol{W}_1\left(\frac{\partial \boldsymbol{h}^1}{\partial N_4}\frac{\partial N_4}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1}\right)\right] \tag{3.27}$$

$$= \frac{\partial C}{\partial N_1}\boldsymbol{h}^1 + \frac{\partial C}{\partial N_1}\boldsymbol{W}_1\left(\frac{\partial \boldsymbol{h}^1}{\partial N_4}\frac{\partial N_4}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1}\right) \tag{3.28}$$

Realizing that $\boldsymbol{W}_1 = \frac{\partial N_1}{\partial \boldsymbol{h}^1}$

$$= \frac{\partial C}{\partial N_1}\boldsymbol{h}^1 + \frac{\partial C}{\partial N_1}\frac{\partial N_1}{\partial \boldsymbol{h}^1}\frac{\partial \boldsymbol{h}^1}{\partial N_4}\frac{\partial N_4}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1} = \frac{\partial C}{\partial N_1}\boldsymbol{h}^1 + \frac{\partial C}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1} \tag{3.29}$$

And $\boldsymbol{h}^1 = \frac{\partial N_1}{\partial \boldsymbol{W}_1}$ if $\boldsymbol{h}^1$ is regarded as independent of $\boldsymbol{W}_1$

$$\sim \frac{\partial C}{\partial N_1}\frac{\partial N_1}{\partial \boldsymbol{W}_1} + \frac{\partial C}{\partial N_5}\frac{\partial N_5}{\partial \boldsymbol{W}_1} \tag{3.30}$$

Recognize that this result is sum over derivatives attained from the outgoing connections using the method in 3.2.

Generally it turns out that a real valued (**not symbolic**) cost gradients for a node can be computed by such accumulation over derivatives from outgoing edges.

$$\overline{N}_j = \sum^{\forall_k \in \mathbb{P}_{N_j}} \left(\overline{N}_k g(N_k, N_j)\right) \tag{3.31}$$

Where

- $\mathbb{P}_{N_j}$ is the set of nodes that are parents to node j
- $\overline{N}$ is the computed value of the cost gradient wrt N
- $g(N_k, N_j)$ is the partial derivative ${}^{\partial N_k}/_{\partial N_j}$ computed under the assumption that no other children of N_k is dependent on N_j

This means that a node class need only implement a method for computing $g(self, child)$ with no regard to how the rest of the network is connected. A slight rewrite to snippet (3.2) implementing this can be seen in snippet 3.3

3.2.2 Concrete implementations

In the code snippets 3.1, 3.2 and 3.3 it is noted that a *concrete* implementation should contain certain methods. This is because those snippets contains constructs called *Abstract Base Classes* (ABC). An ABC represents an *interface* that guarantees any concrete implementation hereof will exhibit some defined behaviour. It can be seen as a template with "blank" methods that needs filling out. In the case of 3.1 a template for a forward node is defined, and for this

```
# node3.py
class BackwardNode2(ForwardNode):
    def __init__(self, *input_nodes):
        self.output = None
        self.grad = None
        self.input_nodes = input_nodes
        self.parents = list()
        for inp in self.input_nodes:
            inp.register_parent(self)

    def register_parent(self, parent):
        self.parents.append(parent)

    def backwards(self, input_node):
        if self.grad is None:
            self.grad = sum(p.backwards(self) for p in self.parents)
        wrt_idx = self.input_nodes.index(input_node)
        dout_din = self.pdev(wrt_idx, self.output, *self.in_values)
        return self.grad * dout_din
```

Snippet 3.3: Minimal node class that implements backwards pass *that allows multiple parents*. `Node.pdev` is an abstract method that returns the partial derivative of node output wrt some input index, which any concrete implementation should define. Note that this class inherits from `ForwardNode` from snippet 3.1

```python
# node4.py
class MatMultNode(BackwardNode2):
    def pdev(self, wrt_idx, output, left, right):
        if wrt_idx == 0:
            return right.T
        else:
            return left.T

    def op(self, left, right):
        return left @ right
```

Snippet 3.4: Minimal node class that implements backwards and forward pass for a matrix multiplication operator node. The `@` is a Python3.5 infix operator that does matrix multiplication such that these expressions are equivalent: `A @ B == dot(A, B)`. Note that this class inherits from `BackwardNode2` from figure 3.4

.

class to work a method `ForwardNode.op` that returns the result of some operation must be written. A class that inherits from an ABC and fills out the blank methods is called a *concrete implementation*.

To exemplify how a concrete node would work snippet 3.4 shows the concrete implementation of 3.3 for a matrix multiplication node.

These examples are of course far too simple for practical use. For more in-depth base classes and concrete implementations that covers most normal operators, look at the library created as part of this project call nodecentricRNN[1].

3.3 Recurrent Neural Network

Recurrent Neural Networks (RNN) is a broad family of neural networks that carry over information from earlier timesteps implicitly by including the hidden layer from $t-1$ as input when computing the layer at t.
This makes it possible for input at some t to be used at a later time, in principle it can be carried over indefinitely. In practice however there will be restriction on how far gradients will be backpropagated, until they vanish or explode and have to be dropped to prevent them from dominating the computations. Furthermore information will only be carried over insofar the significance/value of

[1]`https://github.com/emillynge/nodecentricRNN`

the information preserves some kind of "magnitude" at later timesteps.

The exact mechanism for using hidden layers recurrently is varied and several methods has been proposed since recurrent nets was introduced [Jor86]. Most notably LSTM 3.8 and later GRU has improved the results of RNN.

3.4 Central concepts

This section will start of by introducing the simplest conceivable kind of recurrent neural network, a so-called *vanilla* network. Concepts central to recurrent networks will then be explained using this simple network as base to which the concepts and methods can be applied.

3.4.1 Vanilla RNN

One of the simplest type of recurrent neural networks is the Elman Network proposed in 1990 [Elm90]. It differs from a normal neural network by having a context layer that acts as input to the hidden layer. This context layer is simply the hidden layer at $t-1$. This means that every hidden vector $\boldsymbol{h}_t^l$ at layer l and time step t is dependent on the hidden vector $\boldsymbol{h}_t^{l-1}$ from the layer below and $\boldsymbol{h}_{t-1}^l$ which is the context from the previous time step.

$$\boldsymbol{h}_t^l = \sigma\Big(\boldsymbol{\Theta}_l^{con}\boldsymbol{h}_{t-1}^l + \boldsymbol{\Theta}_l^{lay}\boldsymbol{h}_t^{l-1}\Big) \tag{3.32}$$

where $\boldsymbol{\Theta}_l^{con}$ is the context weight matrix for layer l and $\boldsymbol{\Theta}_l^{lay}$ is the layer weight matrix from layer $l-1$ to l. Note that the hidden vector at $t=0$ is always the zero vector and at $l=0$ it is the input vector $\boldsymbol{x}_t$

In the most basic implementation the topmost hidden vector is used as output.

$$\hat{\boldsymbol{y}}_t = \boldsymbol{h}_t^L \tag{3.33}$$

but more often the output will be a *projection* of the topmost layer.

$$\hat{\boldsymbol{y}}_t = \varphi\big(\boldsymbol{\Theta}^{out}\boldsymbol{h}_t^L\big) \tag{3.34}$$

where the transformation φ depends on the application and desired type of output. The projection makes it possible to adjust the number of hidden units in the recurrent layers, independently of the shape of the output.

3.4.2 Skip connections

Deep networks with several recurrent layers, makes for a long distance between input and output which may result in *vanishing gradients*. A way to mitigate this problem is to include so-called "skip connections"[Gra13]
A skip connection is a connection that bypasses layers by either passing input $\boldsymbol{x}$ to a layer directly or allowing a layer to output directly to the output projection $\hat{\boldsymbol{y}}$. Including such connections changes the hidden update function to

$$\boldsymbol{h}_t^l = \sigma\Big(\boldsymbol{\Theta}_l^{con}\boldsymbol{h}_{t-1}^l + \boldsymbol{\Theta}_l^{lay}\boldsymbol{h}_t^{l-1} + \boldsymbol{\Theta}_l^{in}\boldsymbol{x}_t\Big) \tag{3.35}$$

and the projection to

$$\hat{\boldsymbol{y}}_t = \varphi\left(\sum_{l=1}^{L}(\boldsymbol{\Theta}_l^{out}\boldsymbol{h}_t^l)\right) \tag{3.36}$$

In figure 3.4 a graphical representation of the network can be seen. It is fairly cluttered, mostly because of the weights, and a more compact graph where these weight are simply labels on the edges can be seen in figure 3.5
Skip connections are marked with red for output and blue for input. From the graph it is easy to realize that information from an input can flow to any output node at a later time step, even though the only explicit input is x_t. This makes it possible to solve complex problems, that rely on past information such as the prototypical XOR problem. The main advantage is that this can be done without having to increase the dimensionality of input vectors. Past input flows forward implicitly via contexts instead of explicitly including them as input at later time steps.

3.4.3 Character level network

To work with character level prediction the network has to handle input and output that is categorical. This is done using so called *one hot* encoding where every character c_k is treated as a class of its own and the input that *codes* for class k is a unit vector $\mathbf{e}_k$ of length N_v where

$$\mathrm{e}_i = \begin{cases} 0 & \text{for} \quad i = k \\ 1 & \text{for} \quad i \neq k \end{cases} \tag{3.37}$$

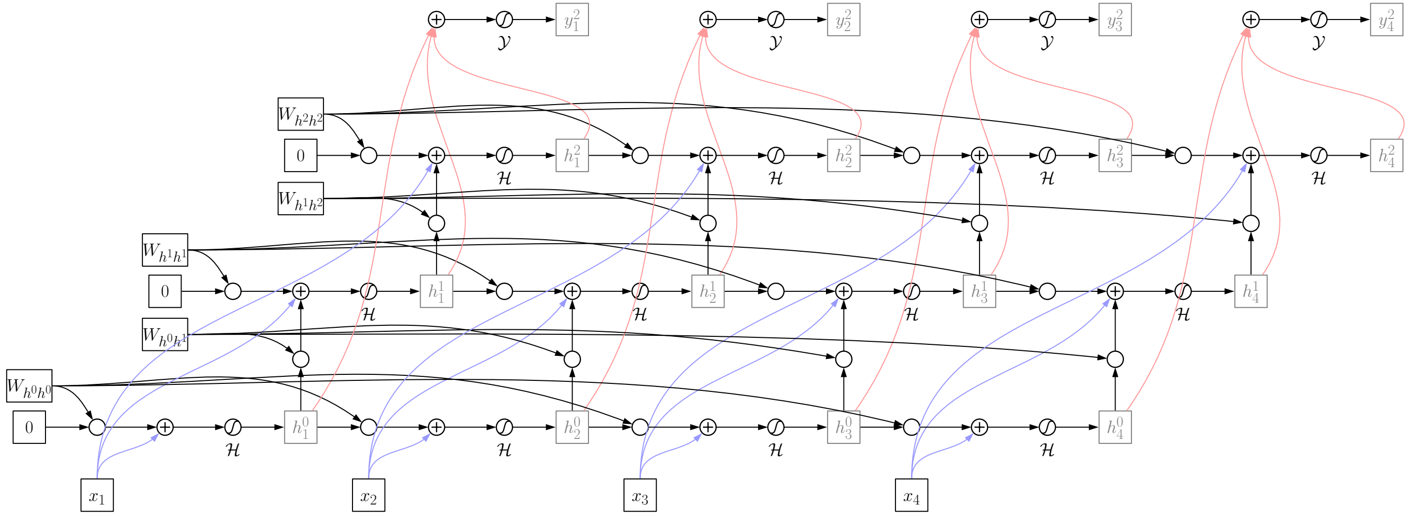


Figure 3.4: A model of a vanilla RNN. All information flows from left to right, and bottom to top. Blue lines are data from raw input, red are data flowing to output

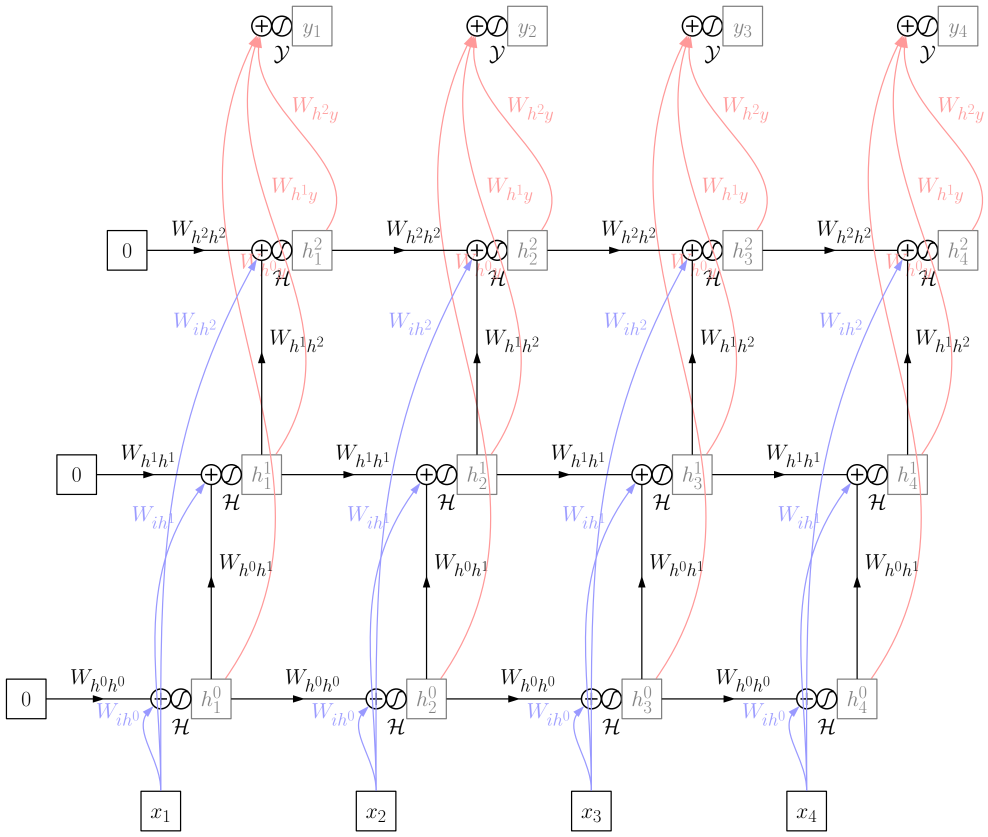


Figure 3.5: A model of a vanilla RNN. All information flows from left to right, and bottom to top. Non-linear nodes are labelled with the type of transform. Edge labels denote what linear transformation is performed on the data flowing along it, if any such is applicable. Blue lines are data from raw input, red are data flowing to output

N_v is the size of the *vocabulary* which is the collection of possible characters that the model can handle.
To obtain output the softmax function is used[Gra13] as projection transformation φ in (3.34, 3.36)

$$\sigma(\boldsymbol{z})_k = \frac{e^{z_k}}{\sum_j^{N_v} e^{z_j}} = \hat{\boldsymbol{y}}_k \tag{3.38}$$

From the formula it is apparent that softmax guarantees: 1) non-negative elements in $\hat{\boldsymbol{y}}$. 2) $\hat{\boldsymbol{y}}$ has a sum of 1. Therefore softmax is a valid probability density function and $\hat{\boldsymbol{y}}$ can be seen as a discrete probability distribution where

$$P(\boldsymbol{y}_t = \mathbf{e}_k | x_t) = \sigma(\boldsymbol{z})_k = \hat{\boldsymbol{y}}_k \tag{3.39}$$

This distribution is called the *coding distribution.* To associate this coding distribution with a scalar cost *categorical crossentropy* is used

$$C(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{N_v} (y_i \log(\hat{y}_i)) \tag{3.40}$$

$$= -\log(\hat{y}_k) \qquad |\boldsymbol{e}_k = \boldsymbol{y} \tag{3.41}$$

The cost is works for feeding a single letter through the network, but usually whole sequences of input/output is evaluated at once for efficiency and ease training of hidden to hidden weights.

$$\boldsymbol{X} = \begin{vmatrix} \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_T \end{vmatrix} \qquad \boldsymbol{Y} = \begin{vmatrix} \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_T \end{vmatrix} \tag{3.42}$$

which gives the following cost metric

$$C\left(\hat{\boldsymbol{Y}}, \boldsymbol{Y}\right) = -\sum_{t=1}^{T} \left(\sum_{i=1}^{N_v} (y_{ti} \log(\hat{y}_{ti})) \right) \tag{3.43}$$

$$= -\sum_{t=1}^{T} (\log(\hat{y}_{tk}) \qquad |\boldsymbol{y}_t = c_k) \tag{3.44}$$

$$= -\log\left(\prod_{t=1}^{T} P(\boldsymbol{y}_t = c_k | \boldsymbol{x}_t) \right) \tag{3.45}$$

This translates to the negative log-likelihood of correctly predicting the output sequence $\boldsymbol{Y}$

3.4.4 Prediction

When discussing prediction in section one must distinguish between the output projection $\hat{\boldsymbol{y}}$ which is a coding distribution and a *prediction* $\hat{\mathbf{e}}$ which is a one-hot

vector.
The prediction is generated by mapping the $\hat{\boldsymbol{y}} \to \hat{\mathrm{e}}$ using one of the following methods.

Maximum likelihood sets the highest probability in $\hat{y}$ to 1 whilst zeroing all other elements.

$$\hat{\mathrm{e}}_i = \begin{cases} 1 & \text{for} \quad y_i = \max(\hat{\boldsymbol{y}}) \\ 0 & \quad \text{otherwise} \end{cases} \tag{3.46}$$

Random draw chooses character c_k with probability $\hat{y}_k$. There are 3 steps in this process

1) A random number drawn from the uniform distribution $r \sim \mathcal{U}(0, 1)$

2) A cumulative distribution generated from $\hat{\boldsymbol{y}}$

$$\hat{y}^{cum}_{k+1} = \sum_{i=1}^{k} \hat{y}_i \qquad\qquad \hat{y}^{cum}_1 = 0 \tag{3.47}$$

3) The prediction vector $\hat{\mathbf{e}}_{t+1}$ is generated using

$$\hat{\mathrm{e}}_i = \begin{cases} 1 & \text{for} \quad \hat{y}^{cum}_i \leq r < \hat{y}^{cum}_{i+1} \\ 0 & \quad \text{otherwise} \end{cases} \tag{3.48}$$

Random draw with temperature chooses character in the same ways a random draw, but uses a slightly different nonlinearity than (3.38). This method is therefore not just a mapping, but also change the way $\hat{\boldsymbol{y}}$ is produced by using this slightly modified softmax function.

$$\sigma(\boldsymbol{z})_k = \frac{e^{z_k/\tau}}{\sum_j^{N_v} e^{z_j/\tau}} = \hat{y}_k \tag{3.49}$$

Where τ is a *temperature* parameter that controls how strongly to prefer highly probable characters.
For $\tau = 1$ the parameter has no effect
When the temperature approaches zero $\tau \to 0^+$, the selection is the same as maximum likelihood.
When the temperature goes to infinity $\tau \to \infty$ the selection is completely random with the values in $\hat{\boldsymbol{y}}$ having no influence on the outcome.

3.4.5 Cold start vs Warm up

In 3.4.1 the Elman network was defined to have a zero valued hidden vector at $t = 0$ and this is usually the case for implementations of RNN. This initial state can also be treated as a parameter to be learned. Another approach is to seed the model with a sequence $\boldsymbol{s}_{seed}$ by feeding it through the network before feeding in the real input. This will change hidden state such that it is more representative of a state "in use".

Wether to use warm up depends the training data available and the concrete application of the trained model. In this project the training data is one large contiguous text string, making the start up phase a very small part of the whole. Normal operation therefore relies on a state that is "in use", which is why some warm up will be used to kick start a text generation

Note that I do not consider transferring a hidden state from the end of a mini-batch to the start of another, as being warm up.

3.5 Text generation

In principle a RNN can input any time-series $\boldsymbol{X}$ and be trained to predict a time series $\boldsymbol{Y}$ from a completely different domain or source. For example one might input an aggregate of financial data at time t and try to predict the price of a specific stock at $t+1$. One of the more interesting uses of RNN is however to predict the next element in the input sequence. For the above example this would be analogous to predicting the stock price at time $t+1$ given the stock price at time t. To create such a network the output is defined as

$$y_t = x_{t+1} \tag{3.50}$$

In this project the input is a text string with length $T+1$ and the analogous task would be to predict the character c^{t+1} given c^t. This string has a one-hot matrix representation $\boldsymbol{S}$ where the columns are characters

$$\boldsymbol{s}_t = \mathbf{e}_k | c^t = c_k \tag{3.51}$$

Input and output is the defined as

$$\boldsymbol{y}_t = \boldsymbol{s}_{t+1} \qquad \boldsymbol{x}_t = \boldsymbol{s}_t \qquad \text{for} \quad = 1...T \tag{3.52}$$

And the cost of feeding a string sequence $\boldsymbol{S}$ into the network is

$$C(\boldsymbol{S}) = -\log\left(\prod_{t=1}^{T} P\big(\boldsymbol{y}_t = \boldsymbol{s}_{t+1} | \boldsymbol{s}_t\big)\right) \tag{3.53}$$

The negative log-likelihood of correctly predicting the next character in the sequence over all time steps.

Predicting a single character is of very limited use, but since the sole input to the network at any time step t is a single character, it is possible to feed the character prediction $\hat{\mathbf{e}}_t$ back into the network as $\boldsymbol{x}_{t+1}$ which in turn will produce a new prediction $\hat{\mathbf{e}}_{t+1}$. This can be repeated indefinitely such that sequences of arbitrary length can be generated. In figure 3.6 a network can be seen that generates 4 new letters given a single character as input.

In practice, it is the best strategy is to define some sentinel $\mathbf{e}_{sen}$, that terminates the character generation.

3.6 Optimizing Weights

In sections 3.1.3 and 3.4.1 the methods for obtaining error gradients has been covered. What remains is a mechanism for using the gradients to adjust model parameters in such way, as to improve model performance. The core loop is the following:

- Forward propagate data to obtain cost measure.
- Backpropagate derivatives wrt obtained cost, to all parameter nodes.
- For every parameter subtract the gradient scaled by some factor. Different update strategies mainly focus on optimal choice of this step size.

Usually this core loop will be extended updating inputs to the network in order to cycle through a data set, as well as wrapping the loop in another loop that repeats training for a number of epochs. See snippet 3.5 for a simple implementation of a 1 epoch training function. The exact method of determining and applying the step size is called the *update* function.

3.6.1 RMSprop

In this project, an update method called Running Mean Squared RMSprop[TH12] is used. It uses a per gradient, running average calculate a factor used for determining step size for every minibatch. A step size based on a running average will smooth out sudden changes in gradients so the step size is more robust

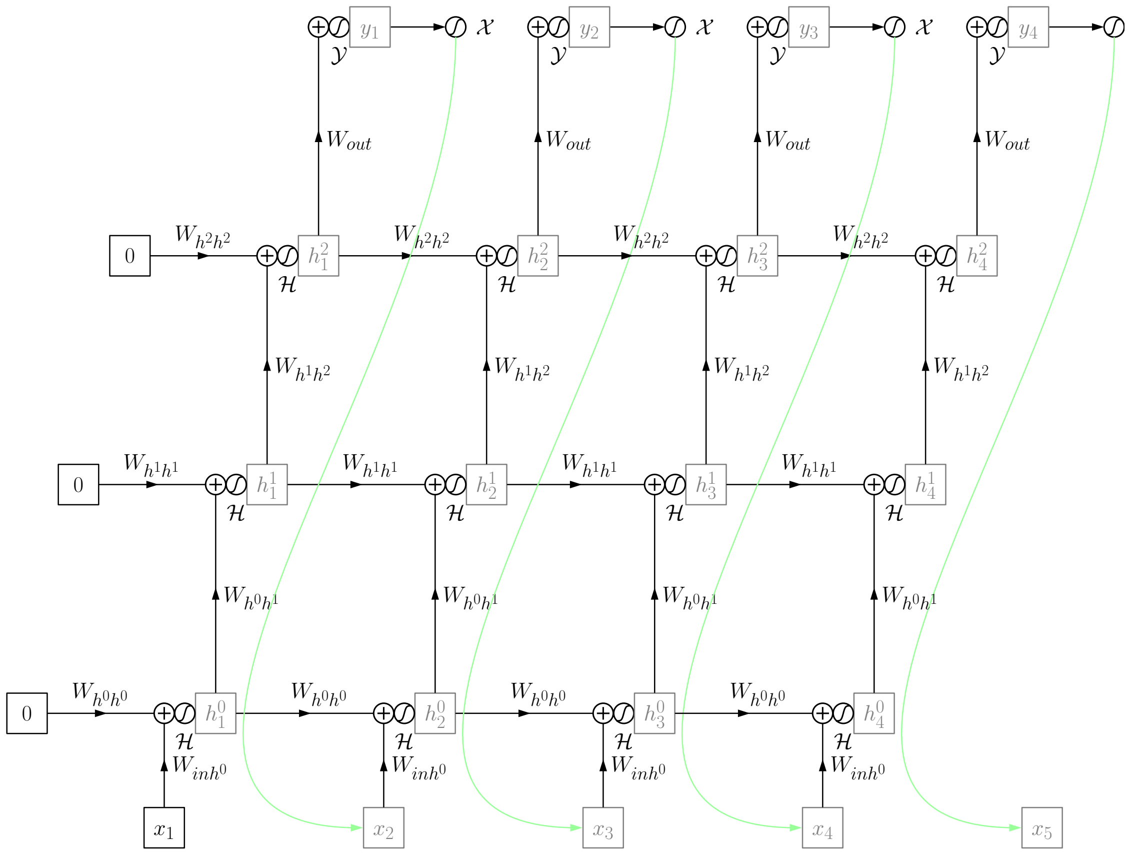


Figure 3.6: A model of a vanilla RNN where output feeds into input via the green connections. The $\mathcal{X}$ nonlinearity, is a prediction transformation like the ones mentioned in section 3.4.4, which is applied to make sure $\boldsymbol{x}_{t+1}$ is a proper input. Note that $\boldsymbol{x}_1$ is a static node, that seeds the sequence generation as opposed to the other inputs which are imaginary.

```
# train.py
import build_net, load_data
from nodecentricRNN import ConcatenateNode, EntropyCostNode
net = build_net.build(arch="VRNN", lay=2, hu=256, features=83)
Y_hat, input_nodes = ConcatenateNode, list()
for _ in load_data.seq_len:
    in_node, y_hat = net.add_t_step()
    Y_hat += y_hat
    input_nodes.append(in_node)

def train_epoch(stepsz=10^(-5)):
    for i in range(load_data.n_batches):
        X, Y = load_data.load(i)
        for in_node, x in zip(input_nodes, X):
            in_node.update_input(x)

        cost = EntropyCostNode(Y, Y_hat)
        cost.forward_prop()
        cost.start_backprop()
        for param in net.parameters:
             param -= param.gradient * stepsz
        cost.backward_reset()
```

Snippet 3.5: A very naïve implementation of RNN training against sudden changes in the gradients caused by very localized phenomena.

The running average is calculated as

$$r_t^i = \rho r_{t-1}^i + (1-\rho) \cdot g_t^i \tag{3.54}$$

where g_i is the ith gradient to be trained and ρ is the decay factor that determines how aggressively to smooth. The running average "window" is reciprocal to ρ such that

$$window \approx \frac{1}{\rho} \tag{3.55}$$

The step size becomes

$$\eta_t^i = \lambda \frac{\eta_{t-1}^i}{\sqrt{r_t + \epsilon}} \tag{3.56}$$

where λ is a hyper parameter called the *learning rate* and ϵ is a small value added for numerical stability.
With the step size the new parameter value is calculated as

$$w_{t+1}^i = w_t^i - g_t^i \cdot \eta_t^i \tag{3.57}$$

In this project the learning rate λ changes throughout the training process as prescribed in [KJL15], such that the step size becomes smaller and smaller for each epoch after some. The reason is that too high λ may cause the error to stagnate, as the model gradient descent overshoots repeatedly. By slowly decreasing it, at some point it will start converging, only to stagnate again if the learning rate is not decreased further. If λ is too small however, the training will not seem to converge because of the small step size. Therefore it should be kept just low enough that convergence doesn't stagnate (or reverse), but seemingly no lower

Specifically the model is allowed to train for 10 epochs before the learning rate is decreased. After that, it is decreased by 5% every epoch.

3.7 Long term gradient problem

The main advantage of RNN is the ability of an input at time step k to influence the error gradient at a later timestep t and therefore gain predition power over large time scales. In this section I will briefly discuss 2 problems that may arise over particularly long sequences (large t)

- Vanishing gradients, which cause the model to lose long term relationships
- Exploding gradients, which cause the model to lose local predictive power due to long term relationships

The basis of this analysis is [PMB12] who bases their argument on this simple representation of the RNN state at time t

$$\mathbf{x}_t = \mathbf{W}_{rec}\sigma(\mathbf{x}_{t-1}) + \mathbf{W}_{in}\mathbf{u}_t + \mathbf{b} \tag{3.58}$$

And the corresponding error gradient wrt. model parameters $\boldsymbol{\theta}$

$$\frac{\partial \varepsilon_t}{\partial \boldsymbol{\theta}} = \sum_{1\leq k\leq t}\left(\frac{\partial \varepsilon_t}{\partial \boldsymbol{x}_t}\frac{\partial \boldsymbol{x}_t}{\partial \boldsymbol{x}_k}\frac{\partial^+ \boldsymbol{x}_k}{\partial \boldsymbol{\theta}}\right) \tag{3.59}$$

The gradient of the error wrt. model parameters at time t is a sum of contributions from every timestep $k \in [1...t]$. Each of these contributions in (3.59) include the factor

$$\frac{\delta \mathbf{x}_t}{\delta \mathbf{x}_k} = \prod_{t\geq i>k}\left(\frac{\delta \mathbf{x}_i}{\delta \mathbf{x}_{i-1}}\right) \tag{3.60}$$

Initially we will only deal with the relatively simple model where the transformation σ is linear by setting σ to be the identity function and using (3.58)

$$\frac{\delta \mathbf{x}_t}{\delta \mathbf{x}_k} = \prod_{t\geq i>k}\left(\frac{\delta}{\delta \mathbf{x}_{i-1}}\mathbf{W}_{rec}\sigma(\mathbf{x}_{i-1})\right) \tag{3.61a}$$

$$= \prod_{t\geq i>k}\mathbf{W}_{rec} = (\mathbf{W}_{rec})^{t-k} \tag{3.61b}$$

Since we can ignore the second term which doesn't depend on $\boldsymbol{x}_{i-1}$
For brevity $l = t - k$ is introduced.

Now we consider another factor of the contribution $\frac{\partial \varepsilon_t}{\partial \boldsymbol{x}_t}$ expressed in a basis composed of the eigenvectors $\boldsymbol{Q}$ of $\boldsymbol{W}_{rec}$.

$$\frac{\partial \varepsilon_t}{\partial \boldsymbol{x}_t} = \sum_{i=1}^{N} c_i \mathbf{q}_i^T \tag{3.62}$$

In this new basis we write the product of the two factors (3.61) (3.62)

$$\frac{\delta \varepsilon_t}{\delta \mathbf{x}_t}\frac{\delta \mathbf{x}_t}{\delta \mathbf{x}_k} = \sum_{i=1}^{N} \left(c_i \mathbf{q}_i^T\right) \left(\mathbf{W}_{rec}\right)^{t-k} \tag{3.63a}$$

$$= \sum_{i=1}^{N} \left(c_i \mathbf{q}_i^T \left(\mathbf{W}_{rec}\right)^l\right) \tag{3.63b}$$

expand using $\mathbf{q}_i^T \left(\mathbf{W}_{rec}\right)^l = \lambda_i^l \mathbf{q}_i^T$

$$= \sum_{i=1}^{N} \left(c_i \lambda_i^l \mathbf{q}_i^T\right) \tag{3.63c}$$

$$= \lambda_j^l \sum_{i=1}^{N} \left(c_i \frac{\lambda_i^l}{\lambda_j^l} \mathbf{q}_i^T\right) \tag{3.63d}$$

if j has the property that $c_j \neq 0$ and $c_{j'} = 0 \quad \forall_{j'<j}$ then

$$= c_j \lambda_j^l \mathbf{q}_j^T + \lambda_j^l \sum_{i=1}^{j-1} \left(0 \frac{\lambda_i^l}{\lambda_j^l} \mathbf{q}_i^T\right) + \lambda_j^l \sum_{i=j+1}^{N} \left(c_i \frac{\lambda_i^l}{\lambda_j^l} \mathbf{q}_i^T\right) \tag{3.64a}$$

$$= c_j \lambda_j^l \mathbf{q}_j^T + \lambda_j^l \sum_{i=j+1}^{N} \left(c_i \frac{\lambda_i^l}{\lambda_j^l} \mathbf{q}_i^T\right) \approx c_j \lambda_j^l \mathbf{q}_j^T \tag{3.64b}$$

As $|\lambda_j| > \left|\lambda_j'\right| \quad \forall_{j'>j}$ and therefore $\left|\frac{\lambda_j}{\lambda_{j'}}\right|^l \approx 0$ when $l \to \infty$
The condition on j simply translates to the largest eigenvalue of $\boldsymbol{W}_{rec}$ which correspond to a eigenvector which has a non-zero projection of $\frac{\partial \boldsymbol{\varepsilon}_t}{\partial \boldsymbol{x}_t}$

Recall that l is the temporal distance from current step t to a gradient contribution from step k. As such it is easily realized that as t increases the contributions with large l will begin to dominate the gradient. Depending on the eigenvalue λ_j this may cause one of the 2 problems prefacing this section

$$\boldsymbol{g}_k(l) \propto \lambda_j^l \begin{cases} 0 & \text{if } \lambda_l < 1 \\ \infty & \text{if } \lambda_l > 1 \end{cases} \tag{3.65}$$

It is from this that the following conclusion comes:

"It is *sufficient* for the largest eigenvalue λ_1 of the recurrent weight matrix to be smaller than 1 for long term components to vanish (as $t \to \infty$) and *necessary* for it to be larger than 1 for gradients to explode." [PMB12]

The implication of this is that small eigenvalues will guarantee vanishing gradient problems, whereas large eigenvalues risk making the gradients explode iff the first non-zero loading c_j correspond to a $\lambda_j > 1$.

Furthermore we can deduce that

- One of the two problems *will* occur given high enough l unless $\lambda_j = 1$ which is highly unlikely.
- The rate of explosion/vanishing is exponentially tied to how far from 1 the eigenvalues of $\boldsymbol{W}_{rec}$ are.

This deduction only holds for linear σ but the main insight has been generalized to non-linear σ by [PMB12] with the main difference that tipping point for λ_j is dependent on the σ used.

3.7.1 Gradient clipping

To curb the mentioned gradient problems, a very simple solution called gradient clipping is often applied. The concept is simply to introduce restrictions in the gradient back propagation that constrain the values of the gradient. Usually this is done by setting a cap on the norm of the gradient vector and truncating or scaling values such that the constraints are satisfied. Typical choice of norm is $L1$, $L2$ or $L\infty$, the exact choice not being crucial to the methods described below.

In [PMB12] the proposed implementation constrains the gradient to have a norm below some threshold. If this threshold is overstepped the gradient vector is scaled the factor

$$\frac{threshold}{||\hat{\boldsymbol{g}}||} \tag{3.66}$$

Which ensures that the norm of the gradient is exactly equal to the threshold.

This method is a variation of a truncation implementation proposed by [Mik12]. Instead of scaling the vector, the individual gradient values is truncated at every node to be in some range e.g $[-15; 15]$. This method is used by [KJL15] where it is reported that truncating to $[-5; 5]$ yields good results for the data sets in question.

3.8 Long Short Term Memory

One of the first effective expansions of the RNN architecture was Long Short Term Memory proposed by [Gra13]. This architecture introduces some new concepts that will be discussed in this section. An overview of the architecture can be seen in figure 3.7, which may be a helpful reference when trying to visualize the concepts explained.

3.8.1 Memory Cells

In LSTM there are 2 sets of hidden units in each layer. The hidden state is we know it from VRNN, that feeds into other layer and acts as an out to the "outside". The memory cells however is an internal hidden state that is not directly fed to other layers. The memory cell vector has same size as the hidden state and acts as a more long term storage from context. This can be done as the cells are not directly exposed to the output such that contextual information not relevant in the current time step can be kept back to be exposed at a later time. In the same way the input to these cells is not directly exposed, such that a cell can be directly copied over from a $t-1$ at one point in the sequence, and copied directly form the input to the layer at another point.

3.8.2 Gates

The most significant change over VRNN is the addition of so-called gates. These are effectively parameters computed on the fly, controlling the data flow between other nodes in the network. The main purpose of these gates in LSTM is to control data flow in and out of the memory cells and thus the degree of exposure.

A gate i from node A to node B with associated output and input vectors of length N, constructs a vector of the same size with values in the range $[0;1]$. This vector is then applied by *elementwise* multiplication on the output of node A and result is sent to the input of node B.

$$\boldsymbol{B}_{in} = \boldsymbol{A}_{out} \odot \boldsymbol{i} \tag{3.67}$$

Usually the construction of the gate vector is a linear combination of other nodes with a sigmoid transformation.

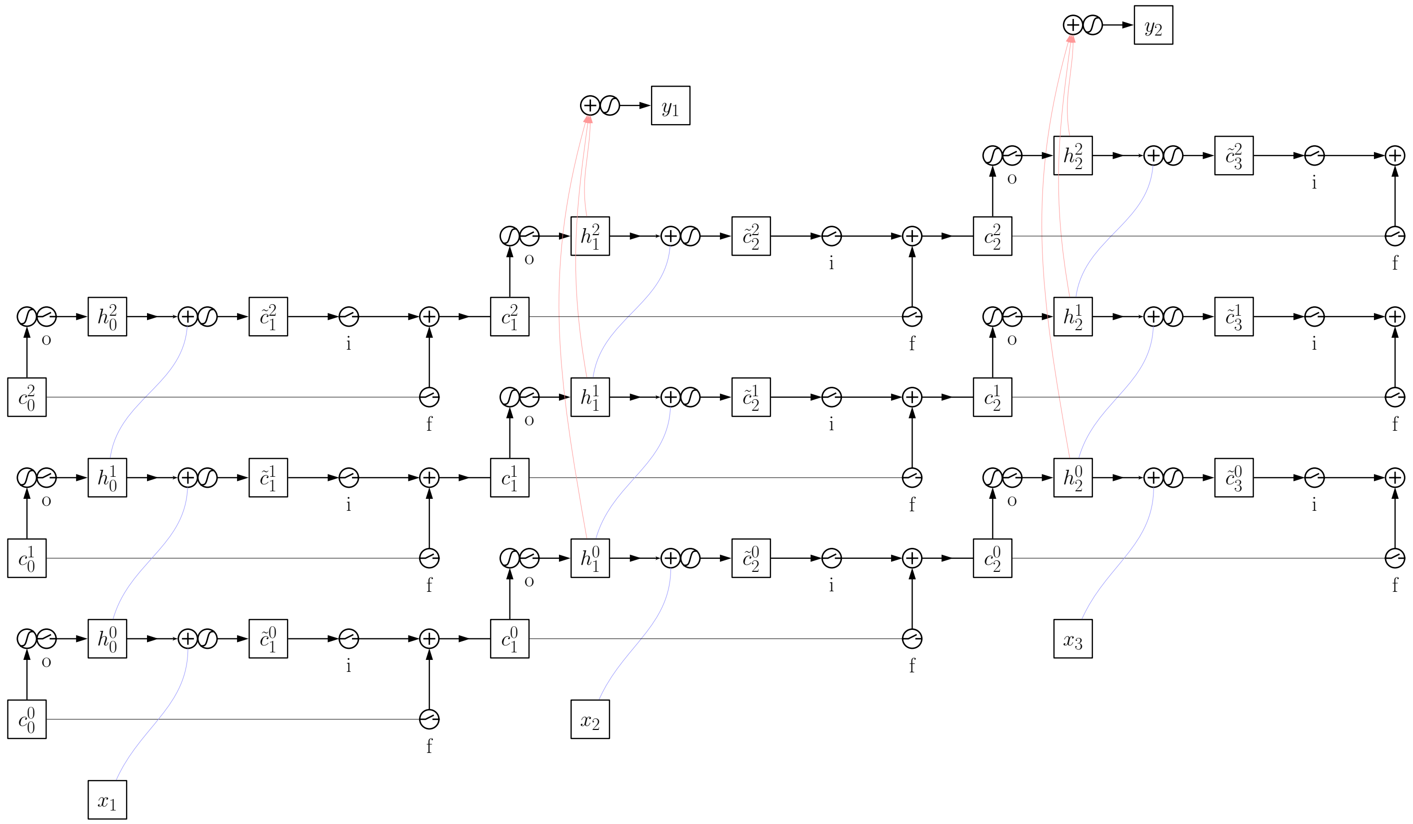


Figure 3.7: A model of a LSTM. All information flows from left to right, and bottom to top. Transformation nodes are labelled with the type of transform. Edge labels denote what linear transformation is performed on the data flowing along it. Blue lines are data from raw input, red are data flowing to output

In LSTM there are 3 gates:

Input gate $\boldsymbol{i}_t^l$ controls the amount of incoming data from the layer below and the previous hidden state to store for later use in the memory cells.

$$\boldsymbol{i}_t^l = \sigma\Big(\boldsymbol{W}_{h^{l-1}i^l}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}i^l}\boldsymbol{h}_{t-1}^l + \boldsymbol{W}_{co^l}\boldsymbol{c}_{t-1}^l + \boldsymbol{b}_l^i\Big) \tag{3.68}$$

It is applied to a candidate memory cell vector based on the layer below and the previous hidden state

$$\tilde{\boldsymbol{c}}_t^l = \tanh\Big(\boldsymbol{W}_{h^{l-1}c^l}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}c^l}\boldsymbol{h}_{t-1}^l + \boldsymbol{b}_l^c\Big) \tag{3.69}$$

Forget gate $\boldsymbol{f}_t^l$ controls the amount of data to retain in layer l from the previous memory cells to the current. Although it is called the "forget" gate it can be helpful to think of it as a *retention* gate, as high activation, results in high retention rather than high forgetfulness.

$$\boldsymbol{f}_t^l = \sigma\Big(\boldsymbol{W}_{h^{l-1}f^l}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}f^l}\boldsymbol{h}_{t-1}^l + \boldsymbol{W}_{cf^l}\boldsymbol{c}_{t-1}^l + \boldsymbol{b}_l^f\Big) \tag{3.70}$$

It is applied directly to the previous memory cells and together with the flow from the input gate it generates the current memory cells.

$$\boldsymbol{c}_t^l = \boldsymbol{f}_t^l \odot \boldsymbol{c}_{t-1}^l + \boldsymbol{i}_t^l \odot \tilde{\boldsymbol{c}}_t^l \tag{3.71}$$

From (3.71) we see that the are 3 modes for generating a memory cell

- with full retention and no input it is possible to "copy paste" a memory cell from previous step to next. Saving the value for later use.
- with no retention and full input a cell can be overwritten completely representing the "newest" data available
- any mix of these will result in a gradual replacement of memory cells that could for example compute running averages and other values with a kind of inertia.

Output gate $\boldsymbol{o}_t^l$ controls the amount of data flowing from the layer l memory cell at time t to the corresponding hidden unit. As the hidden unit *is* the output of the layer this can be seen as the amount of exposure from the memory to the output.

$$\boldsymbol{o}_t^l = \sigma\Big(\boldsymbol{W}_{h^{l-1}o^l}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}o^l}\boldsymbol{h}_{t-1}^l + \boldsymbol{W}_{co^l}\boldsymbol{c}_{t-1}^l + \boldsymbol{b}_l^o\Big) \tag{3.72}$$

It is applied to a candidate hidden state vector $\tanh\big(\boldsymbol{c}_t^l\big)$ to create the hidden state

$$\boldsymbol{h}_t^l = \boldsymbol{o}_t^l \odot \tanh\big(\boldsymbol{c}_t^l\big) \tag{3.73}$$

Together with 3.71 we can see that the layer can at any point selectively chose to output new information, more sluggish inertial context or very long term, almost static, variables. At the same time data cen be held back such that it is only exposed when relevant.

The crucial part of the system is that the computation of a gate vector is dependent of all the information avaibleble to the system, thus making it possible to quickly open and close gates depending on the situation.

3.9 Gated Recurrent Unit

A recent addition to the RNN family is the Gated Recurrent Unit (GRU) proposed by [Cho+14]. It is very similar to LSTM, in that it uses a combination of gates to adjust exposure from input to the hidden states. It does however not use memory cell, instead opting to fully expose it state to the output, thereby also doing away the output gate. Switching between full retention, mixed and forget mode is implemented using a reset gate r and an update gate z.

Reset gate controls the exposure from the previous hidden state $\boldsymbol{h}_{t-1}^l$ to a candidate $\tilde{\boldsymbol{h}}_t^l$

$$\boldsymbol{r}_t^l = \sigma\Big(\boldsymbol{W}_{h^{l-1}r}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}r}\boldsymbol{h}_{t-1}^l\Big) \tag{3.74}$$

$$\tilde{\boldsymbol{h}}_t^l = \tanh\big(\boldsymbol{W}_{h^{l-1}h^l}\boldsymbol{h}_t^{l-1} + \boldsymbol{U}_l\big[\boldsymbol{r}_t^l \odot \boldsymbol{h}_{t-1}^l\big]\big) \tag{3.75}$$

Update gate Acts as a switch that interpolates between the candidate hidden state and the previous hidden state.

$$\boldsymbol{z}_t^l = \sigma\Big(\boldsymbol{W}_{h^{l-1}z}\boldsymbol{h}_t^{l-1} + \boldsymbol{W}_{h_{t-1}z}\boldsymbol{h}_{t-1}^l\Big) \tag{3.76}$$

$$\boldsymbol{h}_t^l = \big(1 - \boldsymbol{z}_t^l\big) \odot \boldsymbol{h}_{t-1}^l + \boldsymbol{z}_t^l \odot \tilde{\boldsymbol{h}}_t^l \tag{3.77}$$

Full retention is simply achieved by not activating $\boldsymbol{z}_t^l$, which simultaneously opens for the previous hidden state and closes the incoming candidate. Conversely forgetting is achieved by full activation if $\boldsymbol{z}_t^l$ *and* not activating the reset gate $\boldsymbol{r}_t^l$.

The model thus has most of the features of LSTM with fewer trainable parameters and shorter paths from output to the hidden state which may result in faster convergence. It does however lack the option to not expose it "memory" to the output, which may hurt long terms memory. Others work has indicated that GRU and LSTM may perform very similarly[Chu+14], but the fewer parameters, and smaller memory footprint of GRU models may be advantageous, given similar performance.

3.10 Theano

Theano[B+10] is a library that uses a node network like the ones mentioned in 3.1 to compute network output and gradients. It has two main features that makes it considerably faster than a simple, pure python implementation like nodecentricRNN described in 4.1.

- The computational graph is optimized. For example by recognizing nodes that hold the same expression and collapsing the graph such that these are merged into a single node, thereby avoid calculating the same values twice. Such a optimized graph trades computation time used on optimizing, for computation time when calculating output. For training of RNNs this is a very good trade as the same function will be called *many* times.
- The resulting computation graph is the compiled into C code, and if a cuda enable GPU is available into C-cuda. A function that interfaces with the temporary binary produced by the compilation is then returned as a python object to be used as any other function. Doing calculations is

lower level languages such as C is often **much** faster because of the lack of checks and garbage collection. The most Python interpreters, including the standard cPython are themselves implemented in C, so it is no surprise that pure C is strictly faster.

By doing these 2 things, Theano can provide performance comparable, or even better[Bas+12], than what can be achieved by framework in other languages that interfaces *natively* with C, such as Torch which is written in lua.

This does, however come with the cost of compilation time, as the compiled functions current has no robust way of being stored to disk. This is probably due to the complex interfaces with Python objects in the run-time environment.

3.11 Lasagne

Lasagne[Die+15] is a neural network framework that implement a layer abstraction atop the node perspective provided by Theano. It makes it possible to view the layers themselves as nodes, rather than the hidden vector, weight matrices and other innards that are contained in a neural network layer. Along with utility function such as parameter update functions, Lasagne also provides a number of pre-set layers such as:

- InputLayer, analogous to an input node.
- DenseLayer, useful for projecting a layer to another shape such as $h_{last} \rightarrow \hat{y}$.
- LSTMLayer, implements a layer in a LSTM network with gates, memory cells and everything else contained in the layer instance.
- GRULayer, like, the LSTMLayer, but implementing a GRU layer.

By using one layer as input to the next, focus can be directed towards choosing the order in which to apply layers to the network, rather than keeping track of the nitty gritty details and inner workings. The process layering the network is called making the *lasagne*.

Chapter 4

Experimental Setup

In this chapter I will describe the conditions under which experiments was carried out, how they were structured and what choices was made that could influence the results.

4.1 Environment

Hardware All model training was conducted on a single machine. The machine has multiple CPUs and GPUs, but the software framework cannot utilize more than one of each at any time. Multiple processes can however use the same devices simultaneously, so timings for how long training took might not be entirely reproducible. Especially since this project was not the only one using these computing resources.

CPU	12 core
GPU	4 x Nvidia Titan X, 12Gb memory, 3072 cores
Memory	32Gb

At no point was any the GPU thrashing or in other ways memory bound.

Software All software was written in python 3.5 which makes it incompatible with 2.7, but in some cases also 3.4. The LASAGNECATERER package for instance **will not** run on 3.4 or below due to the usage of `async` and `await`[1][2] syntax in the asynchronous batch controller. All critical python packages was using the newest available version, and in some cases development versions were used.

Any packages mentioned here that is not available through the Python Package Index (PyPI) can be acquired though the github repositories provided in the footnotes. These can be easily installed with the following command:

```
$ pip install git+https://github.com/[gitusername]/[repositoryname].git
```

Select library versions

- numpy (1.10.4)
- Theano (0.8.0.dev0) (A fork[3] of Theano implementing `@` infix operator was used, but should not be essential for making project code run. This fork is currently under pull request review)
- pycuda (2015.1.3)
- Lasagne (0.2.dev1)

Additionally the code for this project depends on some other, somewhat unrelated, software packages of mine.

- **elymetaclasses (1.9)**[4] A collection of metaclasses for use with python 3 changing the usual behaviour of python classes.
- **monutils**[5] A single file package created as a spin-off from this project. Contains an asyncronous producer-consumer messages model for system monitoring. This is used for awaiting system resources such as free gpu devices, and providing a web GUI for monitoring batch job progress and system load.

There are 3 seperate github repositories containing code central to this project.

[1]`https://www.python.org/dev/peps/pep-0492/`
[2]`https://docs.python.org/3/whatsnew/3.5.html`
[3]`https://github.com/emillynge/Theano.git`
[4]`https://github.com/emillynge/python-metaclasses.git`
[5]`https://github.com/emillynge/monutils.git`

- **lasagnecaterer (0.3)**[6] All the experience with training and using RNNs gained from this project has gone into creating this library. It seeks to create single file containers for the models that eases the workflow of training, testing and using a model in applications. A model consists of 5 entities:

  A *fridge*, which is the model container.

  A *recipe* that build atop the Lasagne library, to provide layer architectures that are customized through mixin classes. The recipe ensures unified training, testing and prediction interfaces.

  An *oven* that handles batch generation.

  A *cook* that is handed the other entities and use them to provide utility such as cross validation and automated model training.

- **nodecentricRNN**[7] The product of the exploration of nodes and computational networks as discussed in section 3.1. It is mostly useful for understanding the how recursive gradient computations work and for solving very simple problems. Large networks are infeasible as there are no graph optimizations or c-extensions like in the Theano library.

- **rnn-speciale**[8] Main repository for the project with all the miscellaneous bits and bobs that cannot justify repositories of their own such as scripts for data aggregation and plotting. This is mainly in internal work tool and *not* very user friendly.

4.2 Crossvalidation scheme

The data set is divided into 3 parts: Training, validation and test sets in 80/10/10 proportions. There is a temporal component so the data is not shuffled before partitioning. This means the first 80% of the characters in the data set is put into the training set, the next 10% is put into validation and the last 10% is put into the test set.

4.2.1 Parameter tuning

Ostensibly only the dropout level is a tuned hyper parameter. But it is important to note that early stopping during training is done by checking the

[6] https://github.com/emillynge/lasagne-caterer.git
[7] https://github.com/emillynge/nodecentricRNN.git
[8] https://github.com/emillynge/rnn-speciale.git

validation error after every training epoch. The number of epochs to use for training, therefore falls into a grey area where it is not a hyper parameter per se, but it does rely on the result of a validation error. This mirrors the implementation in [KJL15], but a case could be made for a practice, where early stopping relies on a hold out partition of the training data.

The full cross validation suite took approximately 2 weeks per data set to complete. This fact has to be taken into account when considering the limitations and bound I imposed in the initial phase. Ideally there had been laxer bounds, but any added tuning complexity may cause cross validation time to become infeasible for this project and the resources available.

Dropout was determined by training using a predetermined set of dropout levels and choosing the one that gave the lowest validation error. In early training and validation, it was found that dropout from the last hidden layer to the projection layer led to very bad performance for all model sizes. The experiment was therefore adjusted such that dropout is only applied to the connections *between* 2 recurrent layers. Dropout on the recurrent connections is usually problematic and dropout from raw input to the first hidden layer is not used by [KJL15] so neither was attempted. On models with only 1 layer, this means there are no connections to perform dropout on and the only application of the validation error is as early stopping criteria.

None of the initial cross validation results indicated that dropout above 30% was beneficial and this was therefore chosen as the highest value for subsequent tuning.

It is possible that better results could be obtained by tuning the dropout level individually instead of using the same value for all dropout layers. But as mentioned above it was not feasible to double computation time by introducing such extra hyper parameter.

Number of training epochs was determined by computing validation performance after each training epoch and stopping training if no significant improvement has been found over the last 10 epochs. To continue training there must be an average validation cost decrease of 5‰between epochs such that

$$C_t \leq C_{t-N} \cdot (1-\tau)^N$$

Where τ is the tolerance of 5‱and N is the memory of 10 epochs. Once the training is stopped the model with the lowest validation error is saved for further use.

4.2.2 Learning rate

[KJL15] proposed that when using `RMSprop`, a starting learning rate in th range $2{\cdot}10^{-2.2}$ to $2{\cdot}10^{-3}$ produces stabile training results and report settling on $2{\cdot}10^{-3}$. Early results in this project found the learning rate to have a large impact not only on convergence time, but also the performance of the converged model. I therefore chose to do a full cross validation and testing suite for 3 different learning rates in this range: $2 \cdot 10^{-2.2}$, $2 \cdot 10^{-2.5}$ and $2 \cdot 10^{-3}$.

4.2.3 Testing

Usually one would use the tuned parameters from the cross validation to train a new model on the concatenated training and validation set. It is however still necessary to determine the number of training epochs using a separate data set. Using the test set for this would defeat its purpose as it would not be truly "unknown" to the trained model prior to testing. Therefore, I decided to reuse the model with the best validation performance to compute the cost on the test set without further training.
Including the validation set in a training set that is already quite large, would not be expected to do much for generalized prediction power. This choice however, does significantly change the temporal distance from what the model has been trained on, to the data it is tasked with predicting. This begs the question of what the testing thought to accomplish, my interpretation being that it should reflect a "real life" situation where the model is *not* continually trained on incoming data. With this choice the test set serves the purpose of showing 2 important things:

- Whether the validation error was low by chance. If that is the case we expect the test error not to be coincidentally low as well and the error should be slightly higher.
- Whether the validation error is dependent, on having the model trained on data that is temporally close to the data it's used on. If so the test error will be much higher as the test set is temporally much farther from the training set than the validation set.

4.2.4 Initialization of hidden state

It is generally recognized that memory cells and hidden state should be initialized to zero values. But [KJL15] stresses the importance of reusing the hidden state between minibatches. Therefore the hidden states are only reset at the beginning of an epoch rather than before every minibatch. In this context the training, validation and test set are seen as separate epochs such that the hidden state is reset before computing the validation and test costs.

Chapter 5

Results

This chapter is broadly divided into 2 parts. The first part of this chapter contains the results obtained from training ~ 350 different RNN models using different model sizes, number of layers, architectures, learning rates and dropout levels. Every combination was trained on both the linux kernel (LK) and war and peace (WP) data set, producing ~ 700 models. A large part of the project is the reproduction of results from [KJL15], so they serve as a standard will compare my own results to. This part divided into 3 sections: Overall test performance, the generalizeablity of the models and the effect of learning rate

The second part showcase text generation that the best of these models were able to produce and discusses the results.

5.1 Test performance

In figure 5.2 all test and validation errors can be seen for the 3 different learning rates. The most interesting of these tables, test error for the best performing learning rate, is shown below with the corresponding results from [KJL15] to the right.

The results for the best performing learning rate can be seen in table 5.1. From

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.492	1.459	1.484	1.448	**1.403**	1.429
	128	1.325	1.318	1.335	1.325	**1.302**	1.326
	256	1.303	1.238	1.254	1.281	**1.234**	1.236
	512	1.355	1.223	**1.217**	1.268	1.332	1.547
Linux	64	1.639	1.636	1.641	1.597	**1.573**	1.605
	128	1.483	1.435	1.500	1.482	**1.412**	1.447
	256	1.475	**1.355**	1.375	1.477	1.366	1.365
	512	1.525	1.331	**1.316**	1.515	1.385	1.323

(a) Test error with learning rate 2e-2.2

LSTM			GRU		
1	2	3	1	2	3
1.449	1.442	1.540	1.398	**1.373**	1.472
1.227	1.227	1.279	1.230	**1.226**	1.253
1.189	**1.137**	1.141	1.198	1.164	1.138
1.161	1.092	1.082	1.170	1.201	**1.107**
1.355	1.331	1.366	1.335	**1.298** [1]	1.357
1.149	1.128	1.177	1.154	**1.125**	1.150
1.026	**0.972**	0.998	1.039	0.991	1.026
0.952	0.840	0.846	0.943	0.861	**0.829**

(b) Corresponding results from [KJL15]

Table 5.1: The column labels 1,2,3 designate the number of layers in the model. The row labels 64, 128, 256, 512 designates the model size i.e the number og hidden units in a 1 layer LSTM model with equivalent number of parameters

this, it is apparent that the performance obtained in this project falls short of the standard set in [KJL15], especially for the linux kernel data set, which will be explored further in section 5.2.
But there are important similarities in the pattern of optimal model configuration. The best combination of architecture and number of hidden layers for a given model size, seems to be invariant to the data set in both tables. Both tables also show a tendency of GRU being *almost* strictly better than LSTM for any given complexity. This tendency is even more strong in this project, where GRU outperform LSTM for almost all combinations model size and number of layers.

5.2 Validation error generalization

An interesting result is how well the validation error generalizes to test error, that is whether there are significant differences between validation and test errors. In this regard it is important to note that it was decided that the validation set would *not* be used for training the model that computes test error. For a justification see section 4.2.3. Therefore it is the exact same model that has been used for computing both validation and test error.

Validation and test error for learning rate $2 \cdot 10^{-2.2}$ can be seen in table 5.2a and 5.2b. For WP there is a very strong correspondence between the two, to such

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.492	1.459	1.484	1.448	**1.403**	1.429
	128	1.325	1.318	1.335	1.325	**1.302**	1.326
	256	1.303	1.238	1.254	1.281	**1.234**	1.236
	512	1.355	1.223	**1.217**	1.268	1.332	1.547
Linux	64	1.639	1.636	1.641	1.597	**1.573**	1.605
	128	1.483	1.435	1.500	1.482	**1.412**	1.447
	256	1.475	**1.355**	1.375	1.477	1.366	1.365
	512	1.525	1.331	**1.316**	1.515	1.385	1.323

(a) Test error with learning rate 2e-2.2

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.502	1.468	1.490	1.458	**1.417**	1.439
	128	1.332	1.326	1.343	1.332	**1.310**	1.329
	256	1.304	1.236	1.260	1.283	**1.235**	1.239
	512	1.354	1.224	**1.216**	1.265	1.340	1.546
Linux	64	1.483	1.465	1.462	1.433	**1.396**	1.422
	128	1.313	1.281	1.337	1.301	**1.245**	1.285
	256	1.277	**1.184**	1.190	1.291	1.185	1.189
	512	1.340	1.138	1.134	1.328	1.175	**1.131**

(b) Validation error with learning rate 2e-2.2

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.531	1.514	1.544	1.474	**1.424**	1.443
	128	1.352	1.356	1.375	1.337	**1.311**	1.333
	256	1.285	1.255	1.297	1.289	**1.227**	1.268
	512	1.310	1.205	1.207	1.266	1.219	**1.197**
Linux	64	1.660	1.642	1.657	1.624	1.583	**1.582**
	128	1.498	1.476	1.528	1.487	**1.431**	1.472
	256	1.468	1.375	1.388	1.484	**1.359**	1.369
	512	1.477	1.338	1.336	1.493	1.330	**1.327**

(c) Test error with learning rate 2e-2.5

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.531	1.521	1.550	1.481	**1.434**	1.454
	128	1.353	1.365	1.382	1.347	**1.320**	1.347
	256	1.286	1.257	1.306	1.293	**1.230**	1.270
	512	1.307	1.203	1.210	1.268	1.217	**1.201**
Linux	64	1.516	1.483	1.478	1.460	**1.410**	1.414
	128	1.336	1.304	1.344	1.310	**1.269**	1.315
	256	1.286	1.196	1.211	1.298	**1.169**	1.198
	512	1.290	1.144	1.150	1.312	1.148	**1.137**

(d) Validation error with learning rate 2e-2.5

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.809	1.779	3.064	1.608	**1.567**	1.694
	128	1.544	1.573	1.876	1.434	**1.409**	1.502
	256	1.397	1.412	1.532	1.316	**1.313**	1.314
	512	1.327	1.303	1.348	1.274	**1.225**	1.234
Linux	64	1.829	1.767	1.861	1.750	**1.706**	1.773
	128	1.586	1.540	1.609	1.548	**1.523**	1.573
	256	1.477	1.438	1.479	1.467	**1.405**	1.455
	512	1.494	1.364	1.371	1.538	1.381	**1.362**

(e) Test error with learning rate 2e-3.0

		LSTM			GRU		
		1	2	3	1	2	3
WarPeace	64	1.531	1.521	1.550	1.481	**1.434**	1.454
	128	1.353	1.365	1.382	1.347	**1.320**	1.347
	256	1.286	1.257	1.306	1.293	**1.230**	1.270
	512	1.307	1.203	1.210	1.268	1.217	**1.201**
Linux	64	1.516	1.483	1.478	1.460	**1.410**	1.414
	128	1.336	1.304	1.344	1.310	**1.269**	1.315
	256	1.286	1.196	1.211	1.298	**1.169**	1.198
	512	1.290	1.144	1.150	1.312	1.148	**1.137**

(f) Validation error with learning rate 2e-3.0

Table 5.2: Test and validation error for 3 different learning rates. The outermost index in the rows is the data set: Linux Kernel and War and Peace. The inner index is the model size which is the number of hidden units in a LSTM with 1 hidden layer and equivalent number of parameters in the model. The outermost column is the RNN architecture and the innermost is the number of hidden layers in the model. The model with lowest error in the row is written in bold and represents the best model of a particular complexity.

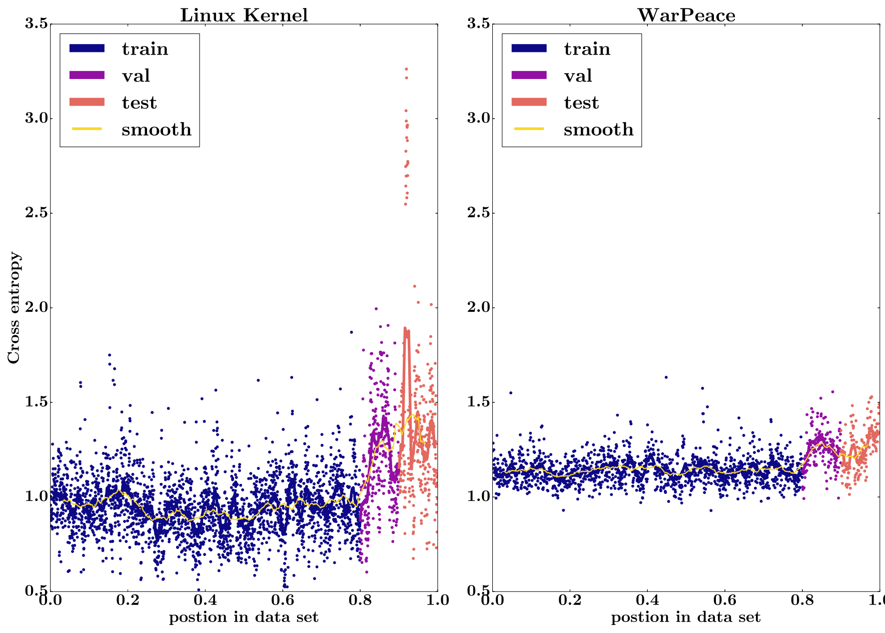


Figure 5.1: Cost for all mini batches in the data set using a 2 layer LSTM with model size 256 and learning rate of $2 \cdot 10^{-2.5}$. Note the how LK har much more varied performance in the training set.
The LK data set is approximately twice as large as WP, so the x axis are scaled differently, affecting the slope. Hidden states are **not** reset between the partitions

an extent that several test errors are actually lower than the validation error. This indicates that the last chapters in the novel somehow follows a structure, that is slightly more similar to the first part of the novel than the validation set is, or that their structure is just "simpler" in some way. But for LK there is a very large discrepancy between the performance on the validation and test set. A possible reason is that the LK has some long term, temporal dependency not present in the WP. This could for instance be function or variable names that is not present in the training partition, but is part the test partition. It is in this regard interesting that the C programming language used in LK does not permit the usage of variables or functions that has not been explicitly declared in the code and that every new function may introduce previously unseen variables. In contrast, one would expect that a novel does not introduce a lot of new places or characters in the last chapters, and thusly the model may be able to rely on the names learned in the training partition.

To examine whether such a mechanism might be at work, the cost of every minibatch in the two data sets are plotted in figure 5.1. The first observation is that LK has a much more varied performance, also in the training partition. It also looks like there is a very sharp performance decrease in LK which is not as profound in WP. One must, however, remember that the two data sets are of different size, and the same rate of change *per character* would give different slopes in figure 5.1. For at better, and more comparable look and what happens at the transitions between partitions, see figure 5.2

What we expect to see in these transitions is that the error rate increases sharply when transitioning from the training partition to the "unknown" data in the validation partition. A major factor in this could be the presence specific words that the model has learned starts to decrease. One would expect that there are used temporally localized words such as a variable name specific to a certain function or a character which is only present in a certain chapter. These may be trained explicitly by the model, and as it transitions into the validation partition, a majority of these words will be "unknown" to the model. We do also expect to see the error stabilize because the model have learned some underlying structures that still has predictive power, even in the test partition. This could be what character combinations are likely in *any* word, or specific words that keeps getting used, such as the names of main characters or keywords specific to the programming language.

From figure 5.2 it does not seem that the rate of performance loss is significantly different between WP and LK. But it is very clear that this climb continues for

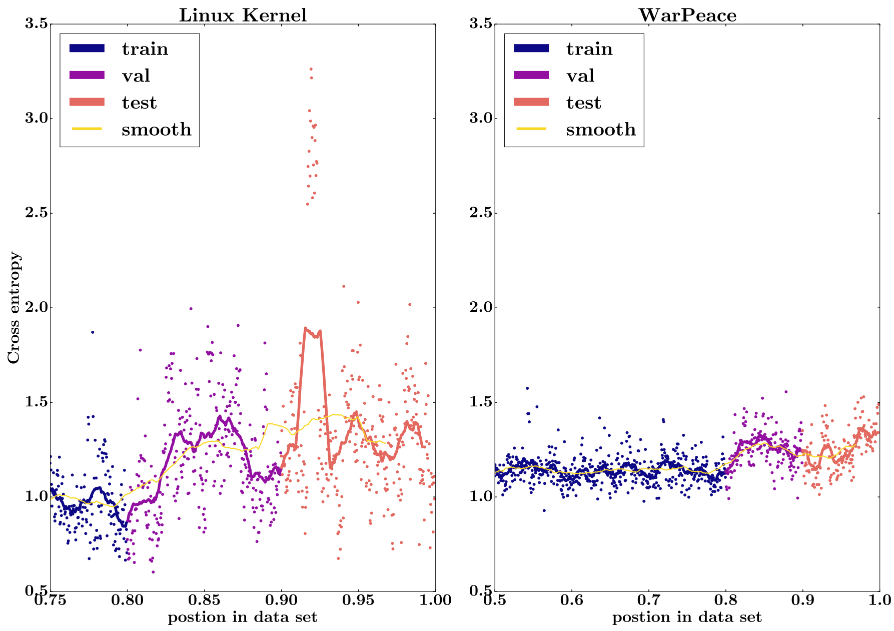


Figure 5.2: Cost for all mini batches in the data set using a 2 layer LSTM with model size 256 and learning rate of $2 \cdot 10^{-2.5}$. Note how the cost increases after the training set stops and then stabilizes in WP. The LK data set is approximately twice as large as WP, so the x axis limits are different to make slopes comparable. Hidden states are **not** reset between the partitions

```
/* struct.c */
struct rb_irq_work {
        struct irq_work             work;
        wait_queue_head_t           waiters;
        wait_queue_head_t           full_waiters;
        bool                        waiters_pending;
        bool                        full_waiters_pending;
        bool                        wakeup_full;
};
```

Snippet 5.1: An example of a struct definition from LK. The name of the struct is `rb_irq_work` and the contents is pairs of data types and field names such as `bool wakeup_full`

much longer in LK whereas WP quickly stabilizes. An interpretation could be that the LK model learns specific words to a higher degree than WP, but that they have a larger temporal span of useability.

Additionally, there is a clear spike at .92 in LK that affects the averaged error greatly. In figure 5.3 it can be seen that at this point there is a sudden spike in the rate of previously unseen words encountered. By inspection of the data set, it is found that at this point in the data set, there is a section dominated by struct definitions. These structs are named data collections with a lot of named fields. An example of a struct definition from this part of the LK data set can be seen in snippet 5.1. This somewhat explains the spike, as this section is almost nothing, but new words being introduced for the very first time in the data set. It also further strengthens the suspicion that the LK model is very reliant on leaning specific words. Note that the figure also shows that there isn't an appreciable difference in the rate of new words encountered overall.

5.2.1 Relation to Karpathys results

Although the above might explain why my results shows such a difference between validation and test error in LK, it does not address that my results show better performance in WP than LK - which is opposite what is found by [KJL15]. There are two probable scenarios that could explain this.

It is possible that my implementation does not accurately mirror the one in [KJL15], and that their model is much better at learning more general structures rather than specific words. Such a model would be much more robust against the temporal dependencies my models seem to struggle with. Their generally better results would support such a theory.

It is also possible that they have chosen to retrain the model after determining optimal dropout level in cross validation. If such retraining included both the training and validation partition, the temporal dependencies I have found to exist in LK may not have had such large effect on their results.

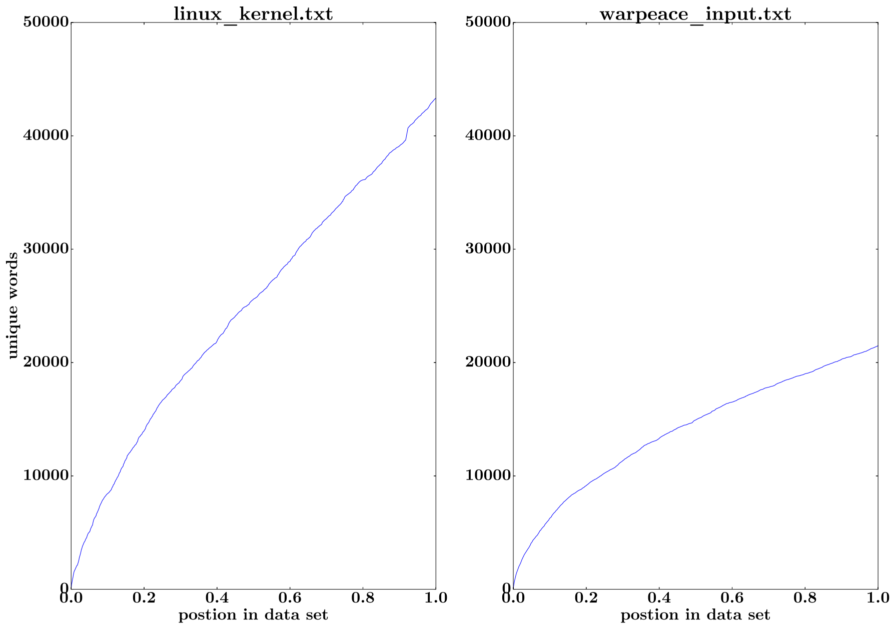
This however would pose the question of what is used as stopping criteria for the retraining. In the paper they explain that:
“We use early stopping based on validation performance and cross-validate the amount of dropout for each model individually” [KJL15]

An approach that would not work if the validation partition was used for training. Therefore, if the model was retrained, it was either trained for the full 50 training epochs without using early stopping or they would have, rather questionably, used the test error as stopping criteria.

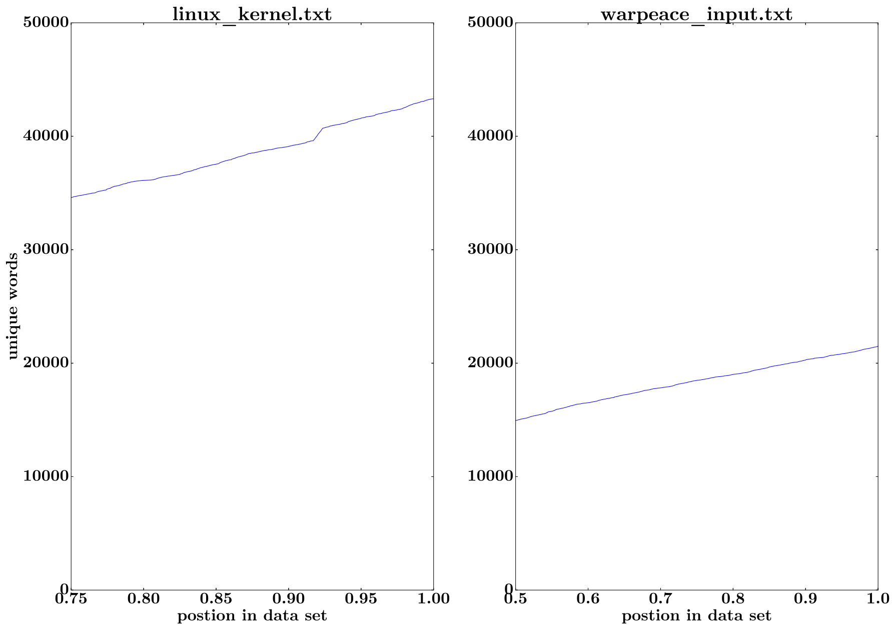
5.3 Learning rate and dropout

Dropout is applied to decrease overfitting issues, and should a cross validation result in no dropout being the optimal setting it indicates that the model has may in fact be underfitting. Conversely a high level of dropout indicates a very complex model that would otherwise tend to overfit. The figure 5.5 shows the relationship between model size learning rate and number of hidden layers. As expected there is a clear relationship between model size and optimal dropout level. This relationship is especially strong in LK

The most significant observation is that models trained with the lowest of the learning rates has optimal dropout that is very low or none at all. This would indicate that they are never allowed to become sufficiently flexible to stop under fitting. This effect is clearly seen in figure 5.4, where the less complex model has little benefit of dropout when training with the low learning rate. But on the other hand the more complex model converges too quickly with the higher learning rate and is unable to utilize the dropout to counter over fitting.The 0.2 and 0.3 lines converges around the same time with the lower dropout providing better performance indicating that overfitting isn’t the obstacle, but rather that the training reached a dead end. The lower learning rate here seems able to more effectively use dropout and keeps favouring higher and higher dropout as the model fits better and better over the epochs.



(a)



(b) Comparable x axis

Figure 5.3: The total number of uniqe words in the dataset at any given point.The steeper the slope, the more higher the rate of previously unseen words encountered. The Linux Kernel data set is approximately twice as large as WarPeace, so the x axis limits in (b) are different to make slopes comparable.

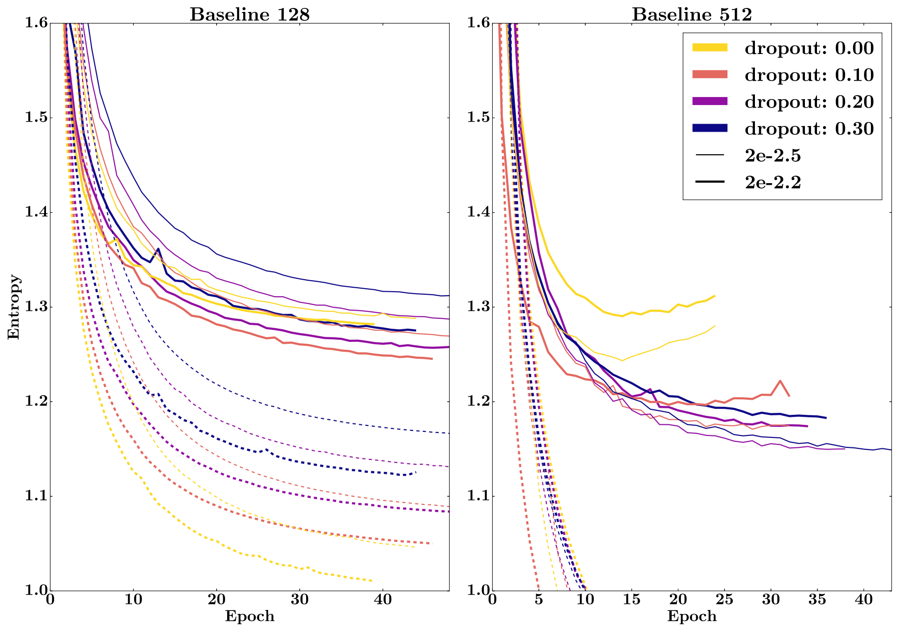


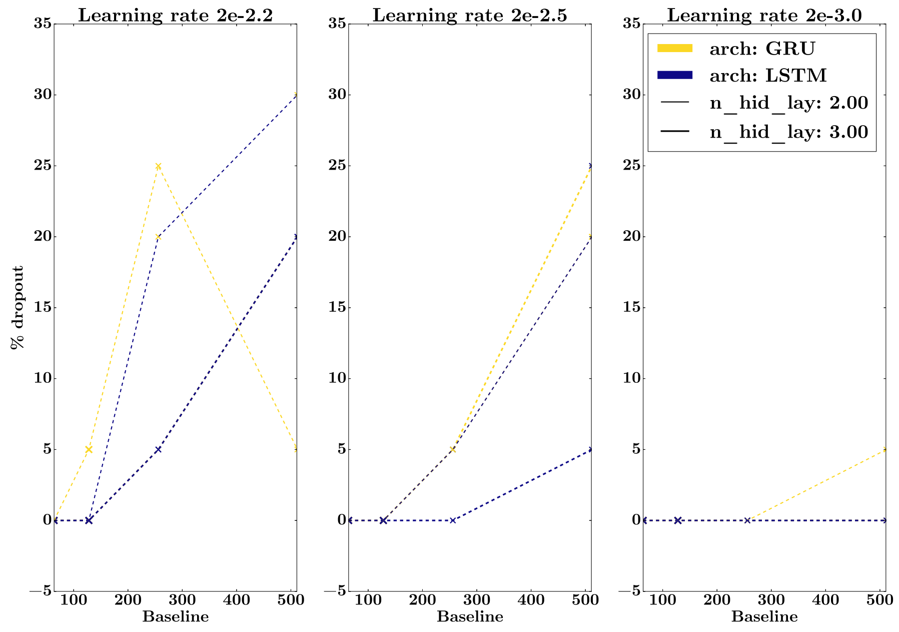
Figure 5.4: validation error vs training epochs for 2 layer GRU model of model size 128 and 512 on the LK data set. Dotted line is training error.

Another insight from figure 5.4 is why the number of training epochs could be seen as a parameter that needs tuning. Overly complex models clearly overfit when trained for too many epochs causing the validation error to rise whilst the training error keeps dropping. This however does not seem to be a problem for models with adequate dropout, and it is not entirely certain that early stopping is required, provided that enough regularization, such as dropout, is applied.

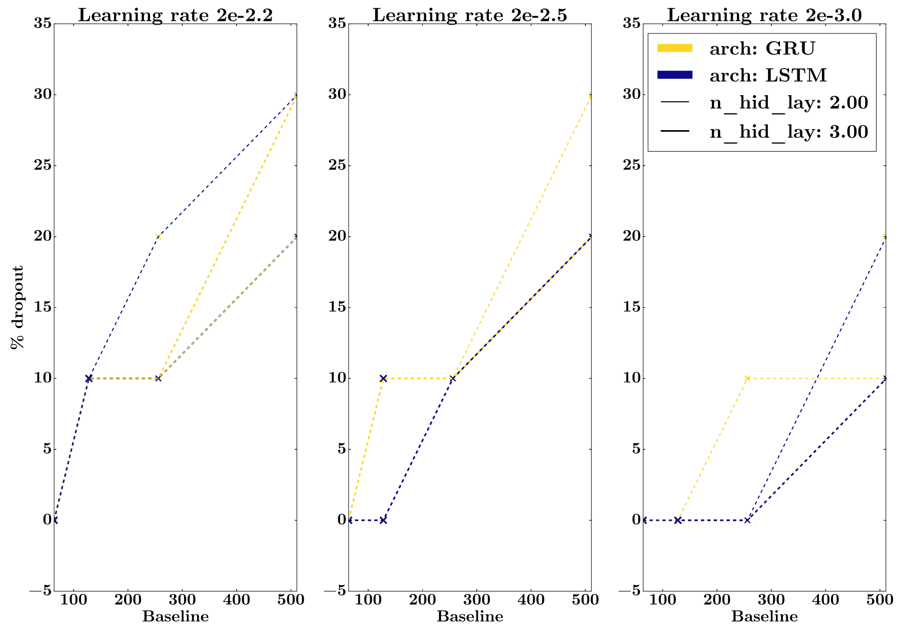
Another, slightly counter intuitive, observation is that more hidden layers causes a lower optimal dropout. Remember however, that more hidden layers are compensated by, by having fewer hidden units such that the models share same baseline. Two models with same model size but with different number of hidden layers therefore has the same flexibility. But dropout is applied between recurrent layers such that a model with 3 layers will have the dropout applied twice whereas a model with 2 layers only has dropout applied once. To have the same overall effect on equally complex models a model with fewer hidden layer must therefore use a higher rate of dropout. What is seen in figure 5.5 is therefore to be expected.

5.4 Prediction

As mentioned in 3.4.4 the output from the softmax projection layer can be treated as a probability vector. When doing character prediction, these values can be used as measure of how confident the model is about the prediction. Below is an excerpt from WP that has been fed into a 3 layered GRU of model size 512 trained at learning rate $2 \cdot 10^{-2.5}$. The color of each character indicates the probability the model assigned to it - Red is low probability and green is high. If the model guessed incorrectly, the prediction is placed below the text

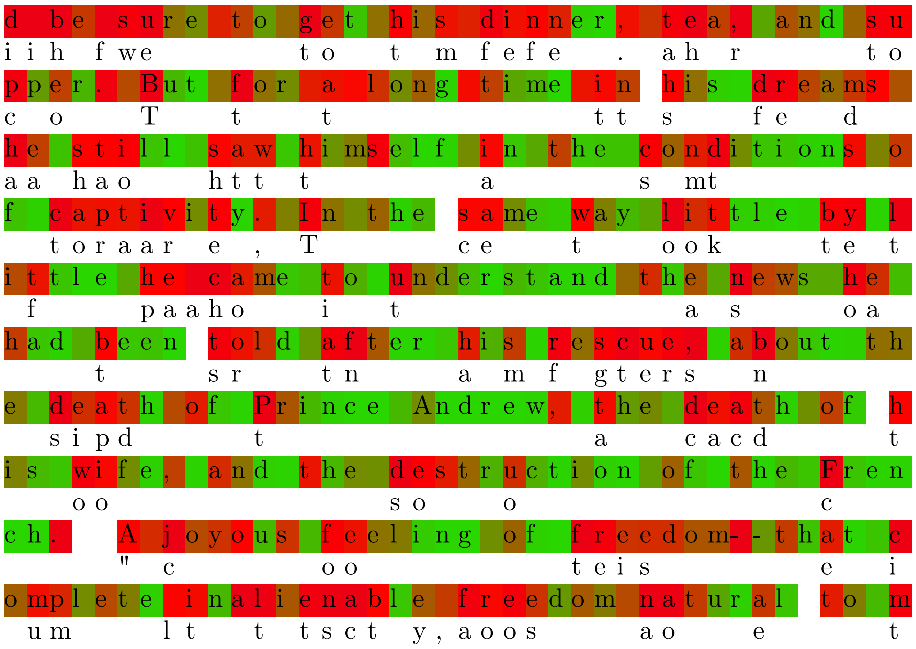


(a) War and Peace data set

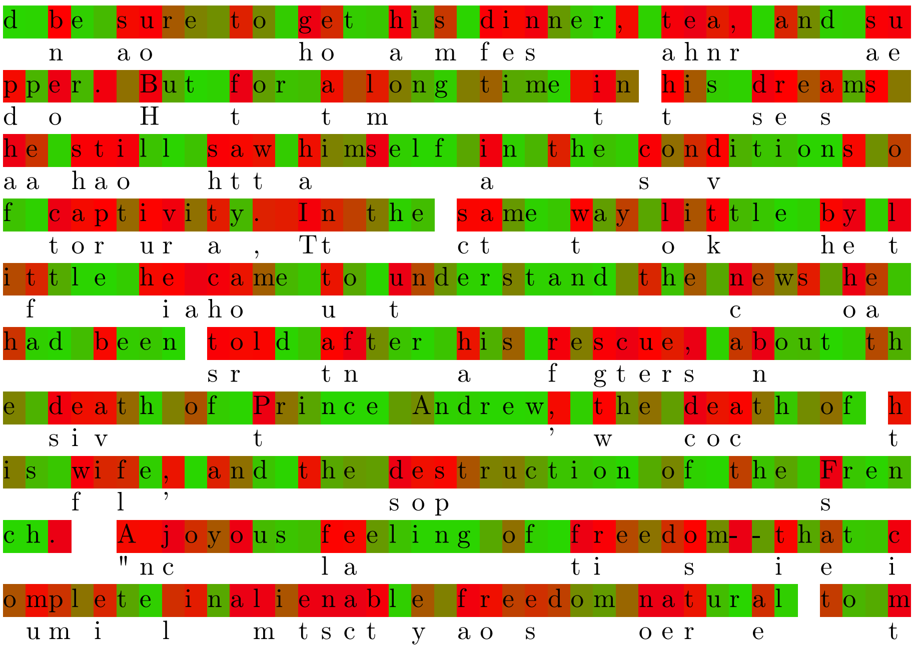


(b) Linux Kernel data set

Figure 5.5: Optimal dropout plotted against model size (baseline) for models trained at different learning rates. Models with 1 hidden layer is not shown as they have not been cross validated for dropout.


d be sure to get his dinner, tea, and su
i i h f we to t m fefe . ah r to
pper. But for a long time in his dreams
c o T t t tt s fe d
he still saw himself in the conditions o
aa hao htt t a s mt
f captivity. In the same way little by l
toraar e , T ce t ook te t
ittle he came to understand the news he
f paaho i t a s oa
had been told after his rescue, about th
t sr tn a m f gters n
e death of Prince Andrew, the death of h
sipd t a cacd t
is wife, and the destruction of the Fren
oo so o c
ch. A joyous feeling of freedom--that c
" c oo teis e i
omplete inalienable freedom natural to m
um lt t tsct y, aoos ao e t

There are some interesting patterns. Firstly the model seems to struggle with the beginning characters of a word, getting better and more confident as it moves away from the start. An interesting exception is name like word combinations such as "Prince Andrew", where "Andrew" is immediately guessed.
Only rarely does incorrect prediction occur in a word where the first few characters have been guessed. One such occasion is "that" in the second to last line where the model incorrectly guesses an "e" in place of the "a". This must be an effect of a *very* common word "the" dominating.
It also struggles at the very beginning of the text, but seeing as this is a cold start it is expected. The same text but warmed up is printed below



Here the first 2 lines are a bit better, but not impressively so, and the performance seems equally good after 2-3 lines.
An interesting piece of the text is the first bit of line 2 "...supper. But...". Note that the model *does* recognize that the sentence has ended by correctly predicting a period. Additionally it recognizes that after a period comes a space, followed by a capitalized letter. The same happens in line 4 where the period is mispredicted, but over since the predicted character is a comma it does still recognize a sentence has ended. And again the space and capitalization follows the period. In the second to last line this capitalization is not predicted, but this could be because the model expected a quotation instead.

5.4.1 Text generation

Section 3.5 describes how to use RNNs for text generation. To show how the model learns, a 3 layered GRU of model size 512 trained at learning rate $2 \cdot 10^{-2.5}$ was trained. Between every training epoch 200 characters was auto generated from a seed of a period. As the model learned, it developed features more reminiscent of certain language components.

```
,,,KKKKKKK444444440444444444444444)4440044444444444444444444444444))40000K044**4444444444444KKK444444444444**KK444444444444400044444::4444KK4K44444444444*K4K4
```

Initialized Almost completely random text, except for the tendency to produce the same characters over and over. This is the first hint that a non-stochastic character generation tends to cycle.

```
ldn hh     hhh     hhh     hhh     hhhh     hhh     hhhh     hhh     hhh     hhhh     hhh     hhhh     hhh     hhhh    hhhh     hhh     hhhh
hhh     hhh     hhhh     hhh     hhhh    hhhh     hhh     hhhh     hhh     hhhh
```

1 training epoch Already the model seems to have grasped the concept of character clusters with spaces in between. But the same characters repeats.

```
TI the and tor to to the tor to to to to to to to to to the tor and to to to the tor to to to to to to the tor to to to the tor to to the tor to to the tor to
```

3 training epochs The words are now pronounceable. Dominated by small common words like "the" and "and".

```
""" sI the countered to the countered to the countered to the countere to the countere to the countere to the countere to the countered to the countere to the
```

6 training epochs First instance of a long word. Some misspelling.

```
""

"If they was the could the counted the maye and the could the counter.

"What they they theye was the could the could the could the could the could the counters, and the could the could the counte
```

9 training epochs

Whole sentences with newlines and periods, but the sentences are gibberish.

```
""

"I have to the countess of the more and the more of the more and the countess of the sound of the countess of the more of the soldiers of the more of the more
```

14 training epochs Long, convoluted and difficult to draw meaning from. But ultimately more or less a valid sentence, and no spelling mistakes.

```
.." To say the prince of the soldiers and the countess of the
same to the prince of the countess and the soldier who had been and
the soldiers were all the countess of the regiment and the street of t
```

27 training epochs The width of the lines now starts to match that of the novel.

```
.." "I should be a strange to the men with
the same to the countess of the countess and the same to see the same to the
country and the soldiers were all the same to the same to the countess.

"I have
```

49 training epochs The last model. The same words still cycles through, maybe even more than earlier.

5.4.2 Randomized

To avoid this cycling through words and phrases a random generation approach can be used as described in section 3.4.4.

```
"
"I shall not be a thing and the same time I have not yet going to the
countess. I shall be a state of a conversation in the same time, and
the same time to the same time. I am so glad to say that I shall remember
that the count had been sent to the countess and the countess and the
same time that he was a man who has been so much as a man who was a
consider of the same time. I am so much that the commander-in-chief
to meet the countess and the countess and the same time that is the same
time to do so. I love you and the same time I am not to do it all the
same time. I can't be a such a consciousness of the same time it was a
strange and more in the count of the countess and a conversation with
men who have been sent to the country of the same time and the same to
the countess and the countess to the countess and the same time that
the countess was a man who had been sent for the same time to the
same time. The count was still more many of the same and the same time that
she had not y
```

$\tau = 10^{-2}$ As expected the low temperature makes the same words cycle.

```
"

"You'll see that the count of the country, and that is it not to take our
Majorse Helene."

"Yes, my dear, and the same young man, but it is no interests, and a
can one another that you would not at all these that the same time I
can that a country. I am so distraction to you."

"All right, and life, that's a state for infantry, and that the same
time in the same service.... What is the count had been an attack in the
```

```
same position and the same mash think of the commander-in-chief has been
for the regiment. And I think that I don't never that matter and that
money devil the millions of the Sovion in the Emperor of Moscow and the
third man in the same things is not what I am fellow. Then I understand the
purpose of the Emperor's head anything to anyone in the way. He is a word
to find the militiamen and the same time, the room with a stern and supers--
and the strange even the count is to do so as much as a battle of
account of him. But this is the soul of the concerning the presence
```

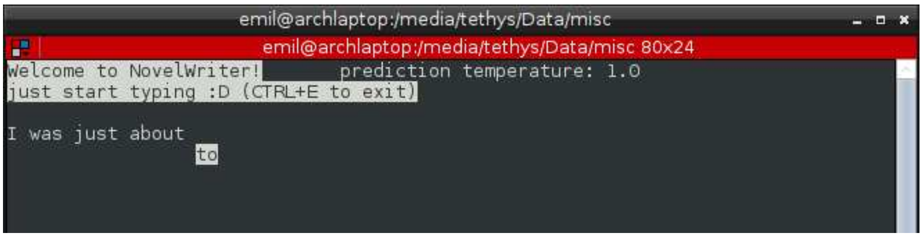
$\tau = 10^{-0.3}$ A lot more variability, but "the same time" still repeats a bit.

```
K'te
paid badly, yet just that day the hidted plump like that.

"Now them, understand, why have nurse because Nicholas had been
atquired to see that this," she said to her,
also, and relations spart galloping about a gray behind. If neither to
make the two companion, mercury to open in the comrades.

"Oly he are so much for which anything would be defendions..."

"Ah, letters voice," said some continue; slave trees which the vainchin
showed the commander the first passage. It seemed to the next corners and
with outline banding at Bolkonski.

Juzh deceased by the regiment all, on, natural is to have for his reception and
different former duty. Why of a moments were brighten understand tho
were losing but for their days, now eat going.

"Oh, you are you like it." (At the near morning altoss as they
pleased various look ill was sitting over Prince Andrew's voice.
"And what you are fortifation? Than God, my dear friend, that issue must
be called, holding a silver." Natasha excased dark, dr
```

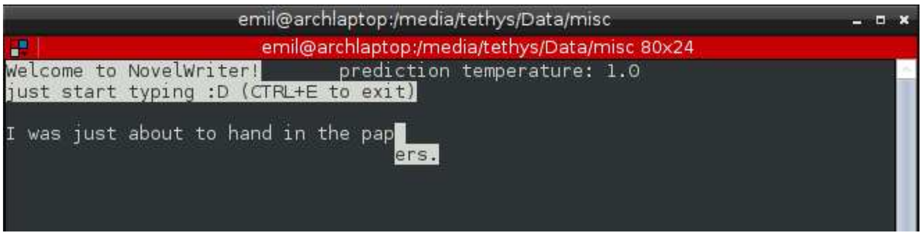
$\tau = 10^{-0}$ No repeats, but the randomization has introduced quite a lot of spelling mistakes.

5.4.3 NovelWriter

To see what kind of application there might be for this kind of text generation a small program is made. It's called NovelWriter and is supposed to "help" an unskilled author write a novel in the language of a true master. It is basically just an autocompletion engine trained on WP. Screendumps from the program can be seen in figure 5.6. The program has the following features:



```
emil@archlaptop:/media/tethys/Data/misc
emil@archlaptop:/media/tethys/Data/misc 80x24
Welcome to NovelWriter!       prediction temperature: 1.0
just start typing :D (CTRL+E to exit)

I was just about
                 to
```



```
emil@archlaptop:/media/tethys/Data/misc
emil@archlaptop:/media/tethys/Data/misc 80x24
Welcome to NovelWriter!       prediction temperature: 1.0
just start typing :D (CTRL+E to exit)

I was just about to hand in the pap
                                   ers.
```

Figure 5.6: NovelWriter program. Helps you write novels like Tolstoy by autocompletion

- Always suggest what to write until next whitespace character.
- Press tab to accept the suggestion
- Adjust softmax temperature by using up and down arrows
- Press rightarrow to get a new suggestion
- Limited erase capability by pressing backspace. You can only undo til last accepted suggestion.

The program is available as a single file program and is started in a terminal window by calling:

```
user$ python3 novelwriter.lfr
```

Chapter 6

Conclusion

The training, computation and development of RNNs relies heavily on flexible and fast automatic differentiation schemes. The principles of these schemes are very simple and can easily be implemented, in a easy to understand albeit inefficient way. As a learning tool such a simple implementation has its benefits, but RNNs are not feasible applications for it.

Unfortunately the need to compile at run time when using the more efficient Theano library, has been a greater challenge than initially anticipated. Much consideration, and time, has gone into ensuring that Theano functions are defined and cached on demand rather than at model instantiation. This is necessary to avoid spending resources on compiling functions that may, or may not be used in that session or recompiling an already compiled function. If such challenges are not addressed it may hinder widespread usage of the Lasagne library for actual applications.

On-demand functionality has been a key part of rationale behind lasagnecaterer library written for this project, as well as complex bookkeeping to invalidate cached functions whenever underlying assumptions change. lasagnecaterer models can therefore liberally define numerous Theano functions, without having to consider the compile time of seldom, used functionality. Together with automated batch generation, containers for packaging the model and other utility, the framework hopefully reduces the barrier to entry, that may hinder the practical use of RNNs.

From my tests I cannot conclude that the Python implementation, provided through the Lasagne library leads to the exact same performances or training recommendations as [KJL15].

Learning rate has been found to benefit from a higher value than the one recommended in [KJL15], although not outside the *range* recommended. Additionally there seems to be some dynamic between the learning rate and model complexity which could suggest that a global learning rate may not be the best solution. How to tailor the learning rate to complexity though, has not been in the scope of this project.

Performance of my implementation has been found to fall somewhat short of the standard set, despite putting considerable effort into making a Lasagne implementation that is equivalent to the torch implementation by [KJL15]. I did not find GRU and LSTM to have equal performance. In most cases GRU strictly outperformed LSTM, but not to a degree that would justify completely disregarding LSTM as the memory cells could conceivably provide interpretability that outweighs the slight performance gap.

Data set nature has had a very large influence on model performance in my implementation. Although code may have some strict rules that the model can learn, the nature of the functions and variable names does not seem to lend itself well to prediction. The natural language of the novel ,however yielded some interesting results where the models apparently learned not only words, but also grammatical rules and some punctuation.

Appendix A

Lookup

A.1 Node legend

A.2 Glossary

A.2.1 Graphs

edge	A connection between 2 nodes
flow	Moving "something" from a node along the edges in the graph to some other node is a **flow**.
(un)directed	an edge with an arrow is *directed*. The direction of the arrow defines which way flow can occur. If no such arrow exists, an edge is *undirected* and data can flow in both directions.
path	A sequence of edges from node A to B through which flow can occur is a *path*.
traverse	The edges that make up the sequence of a path are said to be *traversed* by that path.

$$\boxed{A} \xrightarrow{X} \boxed{y} \qquad AX = y$$

Ⓣ	transposition
⊖	subtraction
⊕	addition
○	dot product
$\emptyset$	nonlinear transform
$\boxed{\hat{y}}$	imaginary
$\boxed{X}$	static

connected	If there exist a path from A to B they are said to be *connected.*
nected component	a collection of nodes where any two nodes would be *connected* if all edges were *undirected*
cycle	A path that connects a node with itself without traversing the same edge twice is a cycle.
Acyclic graph	a connected component wherein no cycles exist is *acyclic.* Often shortened to AG
parent	Node A is a *parent* of B if there is a *directed* edge going **from** A to B.
child	Node A is a *child* of B if there is a *directed* edge going to A from B.
ancestor	Node A is an *ancestor* of B if there is a *directed* path going **from** A to B.
grandchild	Node A is an *ancestor* of B if there is a *directed* path going **from** B to A.
root	A node that is ancestor to all other nodes in a graph.
leaf	A node that has no children is a *leaf* node
text	An acyclic graph wherein there exists a *root*

Bibliography

[B+10] J Bergstra, O Breuleux, F Bastien, et al. "Theano: a CPU and GPU math expression compiler". In: *Proceedings of the* (2010).

[Bas+12] Frédéric Bastien et al. "Theano: new features and speed improvements". In: (23 11 2012). arXiv: `1211.5590 [cs.SC]`.

[Cho+14] Kyunghyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: (Mar. 2014). arXiv: `1409.1259 [cs.CL]`.

[Chu+14] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: (Nov. 2014). arXiv: `1412.3555 [cs.NE]`.

[Die+15] Sander Dieleman et al. *Lasagne: First release.* Aug. 2015. DOI: `10.5281/zenodo.27878`. URL: `http://dx.doi.org/10.5281/zenodo.27878`.

[Elm90] Jeffrey L Elman. "Finding structure in time". In: *Cogn. Sci.* 14.2 (Jan. 1990), pp. 179–211.

[Gra13] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: (Apr. 2013). arXiv: `1308.0850 [cs.NE]`.

[HS97] S Hochreiter and J Schmidhuber. "Long short-term memory". In: *Neural Comput.* 9.8 (15 11 1997), pp. 1735–1780.

[Jor86] Michael I Jordan. "Serial Order: A Parallel Distributed Processing Approach". In: *ERIC* (1986).

[Kar] Karparthy. URL: `https://cs.stanford.edu/people/karpathy/char-rnn/`.

[KJL15] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. "Visualizing and Understanding Recurrent Networks". In: (May 2015). arXiv: `1506.02078 [cs.LG]`.

[Mik12] Thomas Mikolov. “Statistical Language Models based on Neural Networks”. PhD thesis. Brno University of Technology., 2012.

[PMB12] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training Recurrent Neural Networks”. In: (21 11 2012). arXiv: `1211.5063 [cs.LG]`.

[TH12] Tijmen Tieleman and Geoffrey Hinton. *rmsprop: Divide the gradient by a running average of its recent magnitude.* 2012.