DEPARTMENT OF COMPUTER SCIENCE

# An LSTM for Esperanto

## Development of a Long Short Term Memory Network for Esperanto Grammar Detection

Ella Gryf-Lowczowska

10th January 2022

---

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science by advanced study in Computer Science in the Faculty of Engineering.

---

Word Count of Main Body: 15,020

# Author's Declaration

I declare that this dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. This dissertation has not been submitted for any other degree or diploma of any examining body. All material that is not my own work has been appropriately referenced. Except where clearly stated, this paper and any views expressed within are entirely the work of the Author.

Ella Gryf-Lowczowska, 10th January 2022



i

# Wording of Ethics

This project fits within the scope of ethics application 0028, as reviewed by my supervisor Dr Conor Houghton.

This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bristol.ac.uk/acrc/.

# Executive Summary

Natural Language Understanding (NLU) concerns machine comprehension of human language. NLU is considered an AI-hard problem, stemming from the **hierarchical structure of grammar** versus the **linear, sequential nature** of raw natural language data. Nonetheless, recent work implies that a deep learning algorithm called a Long Short-Term Memory network (LSTM) may possess grammatical encoding abilities [1, 2]. This project develops a LSTM and investigates its ability to model Esperanto grammar, assessed by an affix-prediction task and a human Esperantist control study. The main objective is to determine whether an LSTM for Esperanto is a good test case for using LSTMs to model grammar more broadly. This will hopefully contribute to NLU research and may also interest computational neuroscientists exploring the parallels between artificial intelligence and human brain functioning.

**Methods and Key Contributions**

- Methods for cleaning and tokenizing unstructured Esperanto data according to the system of affixes that define Esperanto grammar.
- Development of a LSTM, trained on multiple Esperanto datasets using various hyperparameter arrangements, and resultant models assessed for affix-prediction capabilities.
- An original set of tasks for assessing the LSTM, approved by Esperanto experts.
- A human Esperantist control study with over 260 respondents from across the world, making it the largest ever study of this nature that I could find evidence for.

**Results**. No LSTM model completed the affix-prediction tasks satisfactorily, with affix-prediction performance accuracy ranging from 28.57% - 14.29%. Human Esperantist performance was dramatically better than LSTM performance, with affix-prediction performance averaging 97.0% across the 263 respondents. LSTM hidden layer size and embedding dimensionality of 650 produced better performance than 350 or 128 size/dimension. Apart from this, smaller batch size and larger learning rate may have influenced slight improvements, but the performance gap was insignificant.

**Conclusions**. The poor results do not necessarily contradict the idea that LSTMs may possess a degree of grammatical ability, but they certainly imply that any such ability is very narrow. I recommend that future research try to delimit the extent of the LSTM's ability, and then focus on developing alternative architectures to be used in tandem with LSTMs as a collective, hybrid solution to NLU use-cases.

# Acknowledgements

I would like to sincerely thank my incredible supervisor Dr Conor Houghton, for his unrelenting support and commitment to my project for far longer than was originally planned. Conor has taught me so much, was always there to provide a helping hand, and made the entire process thoroughly enjoyable.

I must also thank the Universala Esperanto-Asocio (World Esperanto Association) for sourcing hundreds of Esperantists for a study I conducted as part of this thesis. In particular I thank the Association's Vice President Fernando Maia, for his enthusiasm and contribution to the aforementioned study. I thank Derek Roff and Ahmad Mamdoohi for multiple iterations of corrections to the study questions. I also thank Maria Majerczak from the Jagiellonian University of Kraków, and Angela Tellier from ESF Academic, for sourcing yet more Esperantists for the study.

Finally, I thank all the unit directors and my cohort for everything I have learnt from them. I thank my tutor Dr Daniel Page for the support he provided us over the course of the MSc program. I also thank Hannah Coe and Senior Tutor Chris Priest for their understanding and flexibility.

# COVID-19 Statement

The overall impact of COVID-19 on this dissertation was negligible. At no stage of my project did I ever intend to make use of a physical lab or meet with human test subjects in person. I was also fortunate to have a brilliant supervisor who seamlessly used Zoom to hold regular one-to-one meetings with me, and to host his PhD lab group, which I enjoyed attending. Furthermore, the taught units of this MSc program were all managed and run spectacularly well given the regrettable circumstances; the knowledge and skills I acquired during this period thus more than adequately prepared me to undertake this dissertation in the first place.

In my experience, the greatest loss from the pandemic's impact on the MSc program was being isolated from my cohort. The course evidently attracts a unique set of people from all manner of different disciplines, united by a common interest in computer science (and perhaps some shared stress around deadlines). Had we been studying side-by-side all year, I am sure we would have inspired and learnt more from our peers than was possible through online media. Nonetheless, we found other ways to communicate – such as the whole-cohort Discord group chat – and presumably we will all be more resilient engineers as a result of these circumstances.

# Abbreviations

**A.I.** Artificial Intelligence

**BPTT** Backpropagation Through Time

**DBN** Deep Belief Network

**GPUs** Graphics Processing Units

**LSTM** Long Short-Term Memory network

**M.L.** Machine Learning

**NLG** Natural Language Generation

**NLI** Natural Language Interpretation

**NLP** Natural Language Processing

**NLU** Natural Language Understanding

# List of Figures

# Contents

# Part I

# Background

# 1. Introduction

## 1.1 Motivation

Grammar is an essential feature of natural language necessary for mutual understanding. Grammar is a **hierarchical** structure: sentences contain clauses, clauses consist of phrases, phrases consist of words and so on. However, for purposes of external communication natural language is "flattened" into a sequential output, and the interlocutor must implicitly recover the hierarchical structure to comprehend what is meant. Much of the natural language data that we would like to process computationally thus comes in this linear format, such as medical reports or social media posts. Computationally it is *unstructured*, meaning the information is not organised according to any pre-determined data model or schema [7]. An algorithm for understanding natural language should therefore be capable of taking linear, sequential input and extracting its underlying syntactic information.

As of yet, no technology has attained an understanding of context-free grammar from unstructured data where that understanding is both *broad* (comprehension of a large amount of vocabulary and grammar) and *deep* (ability on a par with a fluent human speaker) [8]. The problem is, while popular Natural Language Processing (NLP) technologies like Recurrent Neural Networks (RNNs) are phenomenal sequence-processing devices, they do not explicitly encode syntactic information [9, 1]. Moreover, traditional RNNs become unable to track relevant context over increasingly long distances, which is a problem given that some grammatical rules govern long-term dependencies with intervening irrelevant information. For example, consider the sentence "The **dogs** standing next to the *lady* in the green and luscious park **are** barking" (rather than "**is** barking"). Here the singular subject, lady, gets in the way of the relevant plural subject, the dogs. Technologies like Amazon's Alexa appear to be scratching the surface of Natural Language Understanding (NLU), but at present their success is mostly attributable to other factors, not to deep and broad grammatical competence [10, 11].

Groundbreaking research conducted over the past five years indicates that Long Short-Term Memory networks (LSTMs) may be able to infer abstract grammatical rules from unstructured language data. Subject-verb number agreement has emerged as the standard method for assessing an LSTM's grammatical competence. The idea is that the ability to complete sentences like "The **girl** the *boys* like..." with "**is**" rather than "are" despite the latter being correct for the linearly closest subject, *boys*, is believed to require sensitivity to the hierarchical structure of language. Some researchers have tested LSTMs on subject-verb number agreement tasks involving constructed grammars ([12]), others have used natural language datasets in languages such as English, Italian, Russian and Hebrew ([13, 1, 2]).

The earlier studies (2016, 2018) concluded that LSTMs can learn some things of significance, like how to distinguish between relevant and irrelevant information, but ultimately fail to learn the relevant underlying context-free rules from unsupervised training on unstructured data [13, 12]. In contrast, certain studies conducted in the past two to three years imply LSTMs can acquire a non-trivial amount of grammatical competence [1, 2]. These studies do, however, mention the importance of very specific hyperparameter tuning to the LSTM's success [1, 2, p.1, p.5]. Critics also suggest the LSTM's apparent grammatical encoding abilities could alternatively be explained by opportunistic surface-pattern-based heuristics, not a genuine ability to discover structural properties of language during training [14, 15].

This state of affairs raises two key questions: First, can LSTMs in fact discover hierarchical syntactic information in unstructured training data, or is their apparent success still due to shallow sequence-processing? Second, if LSTMs do possess genuine grammatical competence, how closely tied are these abilities to specific hyperparameters?

These questions motivate the use of Esperanto in this thesis. Natural language datasets often contain nuisance variables that correlate with sequential statistical information and which could be confounded with genuine grammatical competence. Esperanto grammar is far more explicit and regular than natural language grammars - essentially it's rules are more obvious and easier to pick up (for humans anyway). Esperanto should therefore give us room to present the LSTM with tests that more explicitly depend upon on the structural features of the language as distinct from statistical contingencies in the data. If the LSTM performs well we have further support for the idea that LSTMs possess genuine grammatical encoding abilities, and an opportunity to further explore how closely linked such abilities might be to specific hyperparameter arrangements. Conversely, a poor performance would support the critics who question whether the LSTM's apparent success in previous studies came from a statistical approximation of the grammar rather than a genuine ability to model the hierarchical structure of the data.

The outcome might also be of interest to Computational Neuroscience - the perspective from which this thesis is conducted. As discussed in Chapter 2, LSTMs grew out of neural network designs originally based on the human brain and models of how humans learn, and now neural networks are in turn shedding light on the brain. Since Esperanto is widely considered much easier for humans to learn than natural languages, the results of this project may shed light on the parallels and/or differences between artificial language processing and language acquisition within the human brain.

## 1.2 Aims

- Investigate whether an LSTM can learn to model Esperanto grammar.
- If it can, how are its abilities influenced by hyperparameter selection?

## 1.3 Methodology

- **Data Preparation.** An Esperanto corpus is obtained by concatenating three separate Esperanto datasets. The corpus is separated into training and validation sets, cleaned and tokenized.
- **LSTM Development.** An LSTM is developed which can process the tokenized Esperanto corpus. Multiple copies are made, each with varying hyperparameters of embedding and hidden layer dimension, learning rate and batch size. *Hereafter the models will be collectively referred to as 'the LSTM' unless otherwise specified.*
- **LSTM Training.** The LSTM is trained on the corpora using Bristol's High-Performance Computing resources.
- **LSTM Testing.** Testing comprises sentence-completion tasks. Each sentence ends with a root word that is missing its affix (the word ending that contains the grammatical information in Esperanto). Testing assesses the LSTM's ability to assign a higher probability to the correct affix than to incorrect affixes.
- **Human Control.** An original study is conducted, in which over 260 human Esperanto speakers completed affix-prediction tasks highly similar to the test of the LSTM. LSTM results are compared with human performance.

## 1.4 Criteria of Success

- LSTM accuracy on affix-prediction tasks is comfortably above chance.
- Average LSTM performance on affix-prediction tasks is at least within 10-20 per cent of human accuracy.

## 1.5 Hypothesis

Given previous studies indicating grammatical competence in LSTMs and the simple, explicit nature of Esperanto grammar, **the LSTM is expected to fulfill the criteria of success above**. As for hyperparameters, previous studies imply that larger embedding/hidden layer dimensions will do better.

## 1.6 Scope of Project

The project does not provide a complete tokenization of the Esperanto language, rather, only a subset of its grammatical rules are formalised. This is because Esperanto is used here as a a test case to contribute to research regarding *whether* LSTMs have grammatical abilities. This project does not attempt to deliver a concrete application ready for use - if this were the goal a broader formalisation of Esperanto would be required.

The human Esperantist study is intended as a control against which to measure the performance of the LSTM, since the 'deep' requirement of natural language understanding is understanding on a par with a fluent human speaker. The human control also mirrors one of the previous studies whose results I hope to probe: Gulordava et al (2018) compared their LSTM with human (Italian) speakers. Additionally, the Esperantist study may interest those studying language processing in general.

## 1.7 Roadmap

I provide relevant background information about Computational Neuroscience in Chapter 2, NLP (and NLU) in Chapter 3, and Esperanto in Chapter 4. I then outline the implementation of: data preparation, LSTM development, training the LSTM, and testing the LSTM and human control study. Finally I analyse the LSTM's performance and contrast it with human performance, evaluate my project and its methodology, and recommend future avenues of research based on my findings.

# 2. Computational Neuroscience

## 2.1 Origins of the Field

The term 'Computational Neuroscience' was coined at a conference in 1985 by Eric L. Schwartz, to summarise the state of a growing field which was at that point known by various names such as Neural Modelling, Brain Theory and Neural Networks [16]. As leading researchers Churchland *et al* noted at the time, the term speaks to the potential for success stemming from cooperative research projects between neuroscientists and scientists from fields centred on computation [17]. On the one hand, the expression implies that neurobiologists and others primarily concerned with the human brain and nervous system can impart valuable insights to computer scientists, mathematicians, physicists, and others in related fields [17]. The expression also implies the converse; that the conceptual and technical resources of computational research can be utilised to investigate the brain and central nervous system, in particular, how these neural structures use electrical and chemical signals to process and interpret information [18].

## 2.2 Core Principles: Two Branches

From a high-level perspective, then, it can help to think of Computational Neuroscience as having two major branches; a Human-to-Computation branch, and a Computation-to-Human branch. The first incorporates what neuroscience tells us about the human brain and nervous system into computational research. For instance, using neuroscientific models of the networks of neurons that comprise the human brain to design early versions of Artificial Intelligence (A.I.). Conversely, the Computation-to-Human branch utilises the tools, designs and methodologies of traditionally computational disciplines to advance neuroscientific research. One use is data handling, regarding which Bouchard et al believe that utilising high performance computing to process and analyse the burgeoning volumes of experimental data will "revolutionise" the field [19, 20, p.628]. Similarly, great insights have been obtained through the use of AI to model and test theories about how the brain performs computations [21, 22]. According to Chethan Pandarinath, a biomedical engineer at Emory University and the Georgia Institute of Technology, "The technology is coming full circle and being applied back to understand the brain [21]."

Evidently, the two overarching branches of computational neuroscience complement and reinforce one another. Many forms of AI originate from designs that were informed by neuroscience.

As AI in turn deepens our understanding of how the brain processes information, this may lead to further breakthroughs in the design of machines with human-like intelligence, which may in turn be better-equipped to model the computations that go on inside the brain, and so on [21]. This cycle of mutual reinforcement defines and delimits a Computational Neuroscience-approach to natural and artificial language research - the central theme of this project.

## 2.3 Linguistics

Section *Assumptions* will outline two core premises that computational neuroscientists bring to language research. The subsequent section will explain the impact of each premise on the field of Natural Language Processing (NLP). 'NLP' refers to the application of computational techniques to the analysis and synthesis of natural language and speech [23].

### 2.3.1 Assumptions

#### 1. Biological learning is relevant to NLP (*Human-to-Computation*)

An agent, human or artificial, must to some extent *learn* a language in order to process it effectively (here 'learn' could me 'train' in an NLP context). Learning is thus a crucial component of language research across many different disciplines. Indeed, within neuroscience alone, the neurocognitive bases of speech, language and related cognitive functions are studied in neuroscience departments around the world [24, 25, 26]. How the *human* brain learns is referred to as **biological learning**. Computational neuroscientists generally believe that biological learning is relevant to artificial learning, so in light of the above, the assumption follows that biological learning is relevant to NLP.

#### 2. NLP can inform neuroscientific language research (*Computation-to-Human*)

As stated, neuroscientists use computational technologies to handle data and to simulate brain functioning, but also to inspire novel hypotheses about how the human brain processes language. As a recent example, in late 2021 the journal *Neurobiology of Language* published a call for papers centred on the question of *whether the level of performance on sentence comprehension tasks displayed by deep learning language models is achieved through mechanisms comparable to the ones employed by the human brain.* The editors specify that the focus should be on how the architecture, the training task, and the internal representations of deep learning language systems compare and relate to those of the biological language network [27].

### 2.3.2 Impact on NLP

**I. Models of biological learning informed the invention of Deep Learning techniques that now dominate the field of NLP.**

The earliest predecessor of modern Deep Learning was the field of Cybernetics (approx. 1940 – 1960). Initial impetus came from development of the McCulloch-Pitts Neuron, an attempt to mimic the biological neuron computationally [28, 29]. Following additional research by Frank Rosenblatt and others, this artificial neuron took the form of a linear model designed to take inputs, attach weights, and output a function of the input and the weights. Cybernetics was later overtaken by Connectionism, where scientists who again drew from the cognitive sciences developed the concept of an Artificial Neural Network (ANN) with a hidden layer between the input and output [29]. The concept saw artificial neurons connected to form a network, with weights applied to the connections between the neurons to control the strength of the effect one neuron has on another [29]. Crucially, the addition of one-or-two hidden layers made the networks two-or-three layers deep overall - an advancement from the previous period [29].

Unfortunately, investments later dried up and an "A.I. winter" ensued [30, p. 13]. Nonetheless, the tail end of this period saw *Hochreiter and Schmidhuber* introduce the first LSTM model, as well as the development of back-propagation models used to train deep neural nets [31]. Both remain important components of Deep Learning today - and both are used in this project. A third wave of A.I. then emerged around 2006 with the breakthrough development of Deep Belief Network (DBN) [32]. Of note is the increased number of hidden layers - increased *depth* - of models since 2006, which resulted in the term "Deep Learning" being adopted. A deep learning algorithm is just any neural network that consists of more than three layers inclusive of the input and output layer, as represented by the following diagram [4, 3].

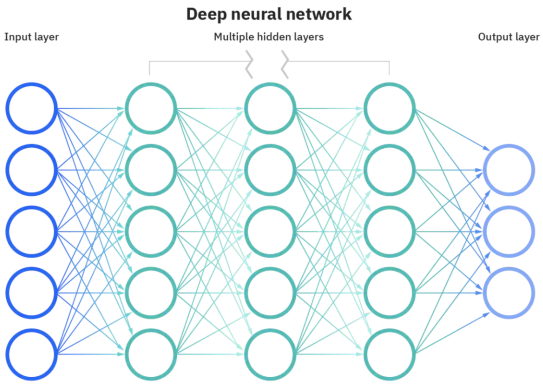



**Figure 2.1:** A deep neural network. *Image by IBM* [3].

Researchers have since built neural networks with upwards of fifty layers, but still many of the networks developed in this century "are directly inspired by biological...and cognitive...models," according to researchers from the Association for Computational Linguistics (ACL) [33]. Hence

we can speak of NLP as having grown and evolved from ideological premises that persist in Computational Neuroscience today. These advancements were largely thanks to phenomenal improvements in computer hardware, in particular, Graphics Processing Units (GPUs). GPUs pack thousands of processing instructions onto a single chip and were used to train the LSTM developed in this project.

## II. AI-driven results from NLP are shedding new light on how the brain processes language

In late 2021, neuroscientists at the Massachusetts Institute of Technology published a study aimed at providing a mechanistic account of how the brain extracts meaning from language [34, p.1-2]. The researchers used various ANNs, including a LSTM, to reverse-engineer the neural mechanisms that underpin the extraction of meaning from spoken, written, or signed words and sentences [34, p. 1].They investigated possible links between the language models' performance and human language processing, as reflected in neural and behavioral data obtained from human participants via brain scans and self-paced reading times [34, p. 1].

The researchers found that the internal workings of next-word prediction deep learning models resemble those of language-processing centres in the brain, in turn implying that the human brain optimises for predictive processing when trying to extract meaning from language [34]. The researchers therefore note that, while contemporary AI focuses on improving model performance over building models of brain processing, "this endeavor appears to be rapidly converging on architectures that might capture key aspects of language processing in the human mind and brain [34, p. 2]." As researcher Daniel Yamins commented, the MIT study indicates that "a kind of convergent evolution has occurred between AI and nature [35]."

# 3. Natural Language Processing

This chapter will define key terms, outline RNNs and their shortcomings, explain what distinguishes LSTMs, and outline the data structures involved. Previous work investigating grammatical ability in LSTMs is also reviewed, and the motivation for this project reiterated.

## 3.1 Definitions

*Key Terms: natural language; unstructured data; Natural Language Processing (NLP); Natural Language Understanding (NLU); Natural Language Generation (NLG).*

A **natural language** is any language that has evolved through use and repetition amongst humans, rather than conscious planning or construction [36, pp.46-72]. Having evolved across time and space, natural languages are rarely simple, uniform or strictly logical [37, 38].

**Unstructured data** is information that is not organised according to any pre-determined data model or schema [7]. The types of data that NLP tasks are designed to process are often unstructured in their raw format, such as emails, medical reports or social media posts.

**Natural Language Processing (NLP)** is the branch of computer science that applies computational techniques to the synthesis and analysis of natural languages by computers, and as explained, it evolved from computational linguistics [23]. The task at the core of NLP is to convert raw, unstructured natural language data into a structured data format(s) that computers can more easily work with to meet given tasks. How and with what success tasks are met can vary greatly, encompassing sub-fields like Natural Language Understanding and Natural Language Generation.

**Natural Language Understanding (NLU)**, sometimes called Natural Language Interpretation (NLI), is the sub-field of NLP concerned with machine *comprehension* of natural language. Three types of analysis are required for a machine to determine the underlying meaning of linguistic data. A **syntactic** analysis to extract grammatical structure of the data; a **semantic** analysis to infer its intended meaning; and an **ontological** analysis to formulate a data structure specifying the relationships between words and phrases [39].

**Natural Language Generation (NLG)** is another sub-field of NLP that focuses on machine-driven construction of natural language text or 'speech' [39].

## 3.2 Recurrent Neural Networks

An LSTM is a special kind of Recurrent Neural Network (RNN). This chapter will first discuss RNNs: their place in the field of artificial intelligence, how they learn, and the problem of vanishing or exploding gradients that make RNNs ineffective in the face of long-term dependencies in the data. Then follows a discussion of what distinguishes LSTM architecture from other RNNs and how this architecture addresses the problem of long-term dependencies.

### 3.2.1 Context

The term 'Recurrent Neural Network' refers to a group of machine learning architectures which are together a subset of a broader group of architectures known as Artificial Neural Networks (ANNs). As the previous chapter explained, 'deep learning' refers to a subset of ANNs that have more than three layers inclusive of the input and output layer. Most ANNs can therefore be shallow or deep, depending on the number of layers. This thesis develops and investigates an LSTM with two hidden layers between the input and output, meaning that inclusive of the input and output the LSTM has four layers in total, making it a deep LSTM. Back to the high-level classification, neural networks (ANNs) themselves are a form of Machine Learning (M.L.), which is in turn a form of artificial intelligence. IBM usefully represent the relationship in terms of Russian nesting dolls, where each technology is essentially a component of the prior term [4].

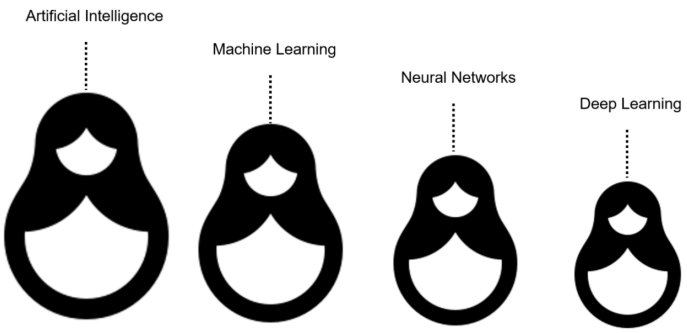



**Figure 3.1:** Each technology is a subset of the prior term. *Image by IBM* [4].

### 3.2.2 RNNs are Sequential Machines

Recurrent Neural Networks (RNNs) take sequential or time series data as input, making them appropriate for inputting language in its "flattened" (linear, sequential) output form. Moreover, whereas other types of ANN treat inputs and outputs fairly independently of each other, RNNs have a capacity for "memory" which enable it to use the context provided by prior inputs to influence the effect that new inputs have on the output [40]. This makes RNNs

a natural choice for grammatical tasks, where the correct form of a word usually depends on the form of an earlier word or clause in the sentence.

The capacity for memory comes from the loops in RNN architecture that allow information input at an earlier step in the sequence, $x_{t\text{-}1}$, to persist across the network as it takes in new input, $x_0$. Information passing along the loop, between the initial input and the final output, is known as the hidden state of the network [41]. In the image below, created by Christopher Olah in his renowned post '*Understanding LSTM Networks*' [5], the diagram to the left of the equals sign shows how input $x_t$ is passed to the hidden layer(s), $A$, which finally produce output $h_t$. The diagram to the right of the equals sign shows an "unrolled" RNN, which is simply a visualisation technique to demonstrate how information from previous inputs is carried through to later stages of the network that receive new input. At each time step, $t$, the hidden state is calculated from the current input and the previous time step's hidden state [42]. The chain-like structure created by the loop is also telling of the parallels between RNN architecture and sequences and lists [40, 5].

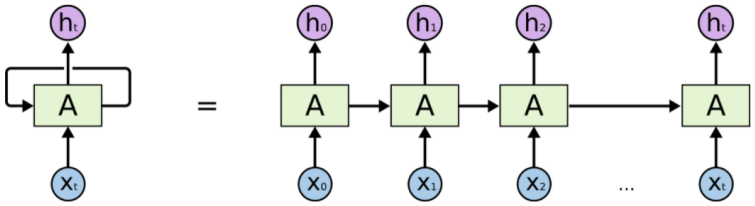



**Figure 3.2:** RNN Architecture *Image by C. Olah* [5].

The diagram above demonstrates how, just like the human brain has 'nodes' (regions) whose edges measure connectivity between regions, the loci of information within a RNN are connected to other such regions in the network. Information is shared within the network: from input to hidden layer, from one hidden layer to another, and from hidden layer to output. Each of these connections is parameterized by a **weight matrix**, and all three weight matrices are shared across time steps. The magnitude of the weight assigned to each connection represents the strength of that connection and ultimately determines the amount of impact that connection has on other nodes in the network [42]. Naturally, adjusting the weights adjusts the strength of the effect that information input to one node in the network (e.g. "dogs" ) has on information input to another node in the network (e.g. "barking"). Adjusting the weights is a core part of how RNNs learn to improve their predictions.

### 3.2.3 How RNNs Learn

Recall that ANNs originate from models of biological learning. According to a simplified account, when human brain neurons receive energy they do not immediately fire it on through the network, but instead act like a threshold logic unit, summing the received energies and then

sending their own quantity of energy on through the network only if the sum reaches a certain threshold [43]. The brain was understood to learn through the process of repeatedly adjusting the number and strength of these connections in a manner than minimises discrepancies between expectation and lived reality. Substitute people's lived reality for a machine-readable corpus of training data and this is a powerful model of how RNNs are trained.

Training a RNN consists of three main stages. (1) First, we instruct the network to do a forward pass and make a prediction. This means, given an unrolled RNN as pictured above, propagate the network forwards for $k$ time steps to calculate the hidden state and the output of the network at each time step $t$. For instance, propagate the network from input $x_{\mathrm{t=0}}$ to input $x_{\mathrm{t=k}}$, obtaining outputs $h_{\mathrm{t=0}}$, $h_{\mathrm{t=1}}$ ... $h_{\mathrm{t=k}}$. (2) Next, the network must compare its prediction, $h_{\mathrm{t+k}}$ ($h_{\mathrm{t=k}}$ in this example), to the target value, $p_{\mathrm{t+k}}$. The comparison is made using a loss function, $e = h_{\mathrm{t+k}} - p_{\mathrm{t+k}}$, where the error value, $e$, indicates how far off the network's prediction was from the truth. (3) Now that we have the error value, we can go back and adjust the weights in the network so as to bring the network's prediction closer to the target value. This final step uses an algorithm called Backpropagation Through Time (BPTT) to calculate the gradients used to update the weights assigned throughout the network [41, 44].

General backpropagation is known as a 'workhorse algorithm' in machine learning [45]. Its core purpose is to determine how much the various nodes in the network contributed to the overall error and to adjust the weights accordingly so as to minimise the error value next time around. To do this, the algorithm travels backwards through the network, calculating the gradients of the weight matrices given the error value from the loss function (a gradient essentially measures how much a function's output changes after a change to the inputs). For each weight parameter, the gradient gives us a sense of how much that weight impacted the overall loss (error value). The objective is to use the gradients to adjust the weights up or down, depending on what adjustments will minimise the loss. BPTT is just a special kind of backpropagation suited to the sequential nature of RNNs. Given an "unrolled" network, the BPTT algorithm steps backwards from the prediction to the first time step, calculating the gradients at each step and adjusting the weights accordingly. Repeating these three stages of training until the loss stabilised is how RNNs learn. However, this causes issues when the algorithm calculates gradients that "vanish" or "explode" as it travels back through the network.

### 3.2.4 Problem of Long-Term Dependencies

The problem of vanishing or exploding gradients means traditional RNNs can only carry relevant information across a small gap between where that information was input and where it is needed. Gradients vanish or explode when the distance between the data cue and the target dependent on that cue is significantly large for the gradient to become *exponentially* smaller or

larger, respectively, as the RNN back-propagates through the network. A vanishing gradient updates the weight parameters by smaller and smaller amounts until they become insignificant (i.e. zero), at which point the network stops actively learning. Essentially, the amount by which the network adjusts itself in order to learn from its mistakes becomes insignificant and so it fails to learn from its mistakes. Conversely, an exploding gradient makes the weights exponentially large, causing significance changes in the loss rate between different updates, making the model as a whole unstable and thereby unable to learn [40]. In both cases, this inability to learn from prior inputs after a given distance between the elements in the dependency is what characterises traditional RNNs as having short-term memory, ultimately motivating development of the LSTM.

## 3.3 The LSTM Explained

### 3.3.1 Cell Structure: Three Gates

The LSTM is a special type of RNN introduced by Hochreiter and Schmidhuber (1997) to address the problem of long-term dependencies [31]. An LSTM has the same high-level chain-like form of repeating units of neural network common to all RNNs and pictured above in the "unnrolled" representation. However, the internal structure of each repeating unit (i.e. hidden layer) of the network, $A$, is far more complex in the case of the LSTM. Whereas each RNN unit is structured by a single tanh layer, the hidden layer of a LSTM is a **gated unit** with four, interacting layers.

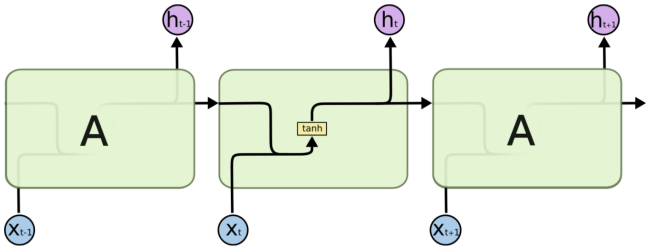



**Figure 3.3:** RNN unit. *Image by C. Olah* [5].

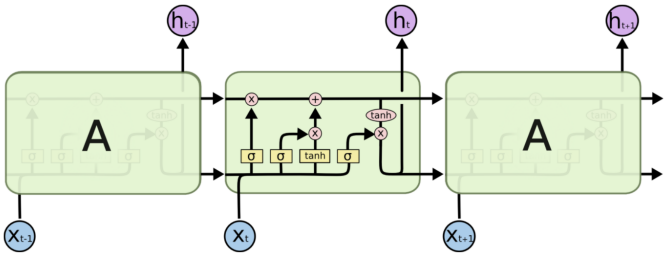



**Figure 3.4:** LSTM cell. *Image by C. Olah* [5].

Each LSTM cell has three inputs: the hidden state of the previous cell, the cell state of the

previous cell, and the current input. For given time $t$ these are $h_{\text{t-1}}$, $C_{\text{t-1}}$ and $x_{\text{t}}$, respectively. The hidden state is essentially a working (short-term) memory. The cell state, represented above by the horizontal line running through the top of the cell, is the long-term memory of the model. The cell state runs through every hidden layer and at each stage can remain unchanged or have information added or removed, determining what information gets passed on to the next unit. This flow of information to and from the cell state is regulated by the three gates in each unit [46].

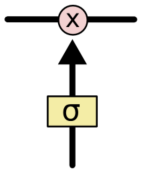



**Figure 3.5:** LSTM Gate. *Image by C. Olah* [5].

If we plot all the computations involved in training a LSTM on a graph, a *gate* is a point on the graph where weights are applied to incoming data in order to filter unimportant information from relevant information. Mathematically, this is achieved using matrix multiplication of data vectors and weight matrix vectors [47]. Gates consist of a sigmoid layer (the yellow box) and a pointwise multiplication operation (the pink circle). The sigmoid layer determines the amount of each component to pass through by outputting a number between zero and one, zero letting nothing through and one allowing all information to pass [5]. The three gates in the LSTM unit are the "forget gate", the "input gate" and the "output gate".

The **forget gate** decides what information to *discard*. It calculates a forget vector by passing the current input, $x_{\text{t}}$, and previous time step's output, $h_{\text{t-1}}$, through a sigmoid function, (F1), with weights specific to the forget gate.

$$\text{(F1)} \quad f_{\text{t}} = \sigma(W_{\text{f}} \cdot [h_{\text{t-1}}, x_{\text{t}}] + b_{\text{f}})$$

The **input gate** essentially filters unimportant information from the current input and previous time step's output, with weight matrices again specific to this gate. A sigmoid function (F2) transforms $x_{\text{t}}$ and $h_{\text{t-1}}$ to values between zero and one. Meanwhile the activation function (F3) takes $x_{\text{t}}$ and $h_{\text{t-1}}$ and creates a vector of new candidate values, $\tilde{C}_{\text{t}}$, to be added to the cell state. The activation function essentially regulates the network by keeping the values restricted within certain bounds, defending against vanishing or exploding gradients.

$$\text{(F2)} \quad i_{\text{t}} = \sigma(W_{\text{i}} \cdot [h_{\text{t-1}}, x_{\text{t}}] + b_{\text{i}})$$

$$\text{(F3)} \quad \tilde{C}_{\text{t}} = tanh(W_{\text{c}} \cdot [h_{\text{t-1}}, x_{\text{t}}] + b_{\text{c}})$$

The new cell state, $C_{\text{t}}$, can now be obtained by multiplying the old cell state, $C_{\text{t-1}}$, by the forget vector, $f_{\text{t}}$, and adding this to the input vector multiplied by the candidate values, as in (F4). This is effectively the new long-term memory of the model.

$$\text{(F4)}\ \ C_{\text{t}} = C_{\text{t-1}} * f_{\text{t}} + i_{\text{t}} * \tilde{C}_{\text{t}}$$

Finally, the **output gate** puts $x_{\text{t}}$ and $h_{\text{t-1}}$ through a sigmoid layer with output-specific weights, shown by (F5). The result is then multiplied by the activation function of the new cell state, as in (F6). This pushes the values to between -1 and 1, producing the new short-term memory for the network. The short and long term values produced by these gates are carried to the next LSTM unit where the process repeats.

$$\text{(F5)}\ \ o_{\text{t}} = \sigma(W_{\text{o}} \cdot [h_{\text{t-1}}, x_{\text{t}}] + b_{\text{o}})$$

$$\text{(F6)}\ \ h_{\text{t}} = o_{\text{t}} * tanh(C_{\text{t}})$$

### 3.3.2 Data Structures: Arrays, Matrices, Vectors

As mentioned, matrix multiplication is key to the functionality of the gates. This is because of the shapes of the inputs and outputs to and from LSTM layers. Input must be a three-dimensional array, representing: the **batch size**, a hyperparameter defining the number of samples to work through before updating the internal model parameters; the **time steps**, the number of values that exist in a sequence; and the number of **units** (sometimes called *features*) at one time step. The features refer to how many dimensions are used to represent data at one time step, for instance if processing pictures, the features at one time step would be the number of pixels in one image [6, 48]. Output from the LSTM is an array of either two or three dimensions, which necessarily includes batch size and units per sequence and optionally includes the number of time steps [6].

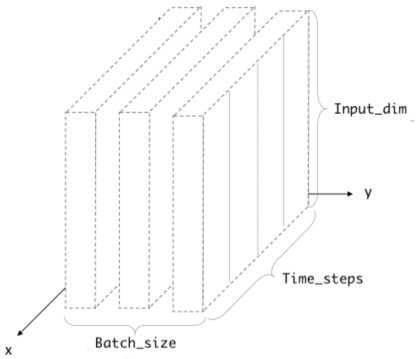



**Figure 3.6:** Three-dimensional Input Array. *Image by S. Verma* [6].

### 3.3.3 Role of GPUs

As an aside, since LSTMs perform matrix multiplication many times across huge bodies of data during training, the process of training an LSTM is very computationally expensive. This is why GPUs have been so integral to the advancement of deep learning. Traditionally,

the core component of most computing devices was the Central Processing Unit (CPU), an integrated electronic circuit that executes instructions from a computer program. A Graphics Processing Unit (GPU) is a specialized electronic circuit with superior capabilities for parallel processing compared with CPUs, allowing GPUs to break complex problems into thousands or even millions of separate tasks and compute each task simultaneously. GPUs were initially developed to accelerate graphics rendering and used by the gaming industry to improve real-time 3D graphics applications. However, the power of GPUs to handle thousands of calculations at once can dramatically speed up machine learning operations, and since the AI community realised, "GPUs have ignited a worldwide AI boom [49]."

GPUs also represent a continuation of the spirit of Moore's Law if not the precise content. Moore's Law states that the number of transistors per silicon chip will double roughly every two years, driving exponential increases in computing power. Reality has been in line with Moore's prediction since it was made in the 1960s, however, experts agree that the 2020s will see computers reach the physical limits of Moore's Law as it relates to transistors on chips [50]. Nonetheless, in terms of operations per second, GPUs arguably represent a continuation of the historical outcome of Moore's Law for computing capacity [51].

## 3.4 LSTMs and Grammar

### 3.4.1 Previous Work

#### Linzen, Dupoux and Goldberg (2016) [13]

As we have seen, LSTMs were initially introduced to extend the memory of RNNs to cover long-term dependencies in *sequential* data. Linzen et al were amongst the first to investigate whether LSTMs can learn to infer the grammatical rules that underpin unstructured language data. The researchers tested several LSTM models on their ability to predict **subject-verb number agreement** in English sentences. This task is assumed to require syntactic information, because English grammar dictates that the form of a third-person present tense verb depends on whether the head of the syntactic structure is plural or singular [13, p.2]. Sentences (a) and (d) below are therefore valid whereas (b) and (c) are grammatically invalid:

(a) The **key *is*** on the table.

(b) *The **key *are*** on the table

(c) *The **keys *is*** on the table

(d) The **keys *are*** on the table.

The subject-verb number agreement task requires the LSTM to assign a higher probability to the grammatically correct verb given the form of its subject. The test is designed to hone in on structural features of language as distinct from statistical features. While for the above sentences the model could employ a simple statistical heuristic, like "agree with the most recent noun," such heuristics become unavailable when the distance between the subject and the verb grows and contains within it intervening nouns with the opposite number from the subject (called *agreement attractors*) [13]. In sentence (e) below, 'building' is an agreement attractor between the subject *banners* and its verb *are*. The theory is that when hierarchical and sequential information conflict like this, a prediction that agrees with the hierarchical rule surely demonstrates an ability to model syntax as distinct from advanced statistical powers [13, p.3]. Number agreement has since become the standard method for assessing LSTMs for grammatical encoding abilities.

(e) The only championship **banners** that are currently displayed within the *building* **are** for national or NCAA Championships.

Linzen et al found that LSTMs can in theory learn to model grammatical structure, but only given explicit supervision on the task (i.e. not directly from unstructured data), and even then accuracy frequently decreased when sequential and structural information conflicted. The researchers concluded that without supervision, LSTM architecture is insufficient to capture a non-trivial amount of syntax-sensitive dependencies from unstructured data [13].

### Sennhauser and Berwick (2018) [12]

Two years after the Linzen result, Sennhauser and Berwick evaluated the ability of LSTMs to learn context-free grammar using a well-formed bracket prediction task using two types of brackets, governed by four simple grammatical rules. Positively, intermediate state analysis showed the model quickly and effectively forgot irrelevant characters, demonstrating it did more than merely memorise the training input. However, overall the results were negative. First, as the distance and embedded depth between relevant brackets grew, the memory requirements of the LSTM rose exponentially where a sub-linear memory should be enough. Given that irrelevant characters were quickly forgotten, the extra memory can't have been needed for storing irrelevant information for the original classification task. Second, the model did not generalise well. Now, as Sennhauser and Berwick point out, the strength of structural rules is that they generalize well; grammar enables humans to create previously unseen and unheard sentences that are immediately meaningful to us [12, p.7]. With this in mind, the authors followed a proof by contradiction: ($P1$) If the LSTM learns the correct rules, the model would generalize. ($P2$) The LSTM generalised poorly. ($C$) Therefore, the LSTM must not have learnt the right rules [12, p.8]. Even though the bracket grammar had only four simple rules, presumably the model learnt to determine each character's relevance without resorting to hierarchical rules.

The authors note that, since LSTMs perform exceptionally well on statistically-oriented sequential tasks, the presence of nuisance variables that crop up in natural language datasets (like sequential statistical correlations between words) might go undetected and be confounded with true hierarchical information. Taken together, the results thereby imply that rather than an explicit rule-based solution, what the LSTM in fact acquired was a sequential statistical approximation to this solution [12, p.8]. Human in contrast can understand the bracket completion task within just a few training sentences, inviting the authors' conclusion that "it will be very challenging for LSTMs to understand natural language as humans do [12, p.9]."

### Gulordava, Bojanowski, Grave, Linzen and Baroni (2018) [1]

In the same year, Gulordava and others (including Tal Linzen from the 2016 study) revisited the issue by testing whether an LSTM and other RNNs trained *without* explicit supervision in four languages (English, Italian, Russian and Hebrew) could predict long-distance number agreement in various grammatical constructions. The researchers used three evaluation paradigms: the Linzen (2016) tasks; new number-agreement tasks using nonsensical sentences; and a human Italian speaker control group. The nonsensical sentences provide a stronger assessment than the Linzen tasks, as they reduce the possibility of the LSTM relying on semantic or frequency-based information rather than purely syntactic structure. After all, the training dataset is unlikely to feature "*The colourless green ideas I ate with the chair sleep furiously.*"

The study was a triumph and contradicts the Linzen (2016) and Sennhauser (2018) findings. Here, an LSTM trained on natural language corpora without explicit supervision on the given task performed exceptionally well on both the Linzen (2016) tasks and the nonsensical sentences, implying that LSTMs can acquire a non-trivial amount of grammatical competence [1].

### Lakretz, Kruszewski, Desbordes, Hupkes, Dehaene and Baroni (2019) [2]

The Gulordava results are complemented by Lakretz et al (2019), who used an approach inspired by neuroscience to study the inner dynamics of an LSTM as it performed number-agreement tasks. Their LSTM, trained without supervision, successfully learned to perform structure-dependent linguistic operations, supporting the hypothesis that LSTM architecture is sufficient for inducing non-trivial grammatical rules from unstructured data [2]. Interestingly, both the Gulordava and Lakretz studies trained their LSTM language models with a 650-dimensional embedded layer and two 650-dimensional hidden layers, and both mention the importance of careful hyperparameter tuning to the LSTM's success [1, 2, p.1, p.5].

### 3.4.2 Unanswered Problems: Motivation for Current Study

While the most recent results have been positive their reliability is debated. Both the Gulordava and Lakretz studies used natural language datasets, but as Sennhauser and Berwick noted, natural language datasets can contain nuisance variables of a statistical nature that could be confounded with hierarchical information. Perhaps the Gulordava and Lakretz results are ultimately grounded in advanced statistical capabilities masquerading as genuine grammatical competence.

Given that Esperanto has an explicit grammar that it follows without a single exception or irregularity, if the LSTM has genuine grammatical competence, it should find it easier to model Esperanto from unstructured input than it would English. Using Esperanto as a test case thus provides scope to present the LSTM with a broader set of grammatical tasks than just number agreement, and to explore the impact of various hyperparameter combinations on its performance.

### 3.4.3 Anticipated Contributions

Mainly, I hope to contribute to NLU-research by investigating whether LSTMs could model grammar more broadly, or whether the NLU-community would be best to research fundamentally different architectures and methods in pursuit of NLU technologies. The sub-field Natural Language Generation (NLG) was also defined above. NLG and NLU are distinct enterprises, however, given that humans apply grammatical rules in service of authoring text or speaking, this project may potentially contribute to NLG research. Finally, potential contributions to computational neuroscientists. We know humans can induce structural rules then generalise to previously unseen examples, and we also know Esperanto is easier for humans to learn than most natural languages. Perhaps a positive result in this project would justify delving more deeply into the inner workings of biological language learning in order to develop better LSTMs for NLU. A poor result might alternatively suggest a fundamental disjoint between LSTM processing and biological learning.

# 4. Esperanto

## 4.1 History

Esperanto is an international auxiliary language with two million speakers world-wide, spanning 120 countries and every continent [52]. It was constructed in the nineteenth century by Polish-Jewish ophthalmologist Dr Ludwik Łazarz Zamenhof (1859 – 1917) [53, 54]. Zamenhof believed that "the diversity of languages is the...basis for the separation of the human family into groups of enemies [53]," leading him to identify the need for a "neutral tongue" that would promote peace by enabling communication between people without a common first language [55]. Zamenhof therefore constructed Esperanto from a variety of widely spoken languages: root words predominantly come from Italian, French, German and English, while others have their origin in Latin, Greek, Lithuanian Russian and Polish [56]. Zamenhof also designed Esperanto with accessibility in mind, claiming its grammar could be learned within an hour [56]. Indeed, when Google Translate added Esperanto, company research scientist Thorsten Brants acknowledged "the high quality of machine translation for Esperanto... Esperanto was constructed such that it is easy to learn for humans, and this seems to help automatic translation as well [56, p. xvii]." Esperanto's accessibility is key to this project.

## 4.2 Grammar

Esperanto grammar is uniquely simple, regular and explicit. It has one definitive article and no indefinite articles, nouns have no gender, and verbs are all regular with only one form per tense [57, 52]. Its vocabulary also has a very small number of root words which can be combined with one or more **affixes** (comparable to case-endings) to create new words as and when needed, removing the need to learn various thematic words without visible ties to each other. For instance, consider the English: *to eat*, *meal*, *food*, *cutlery*, *trough*, *to guzzle*, *to nibble*, *to feed*, or *dining room*. In Esperanto, once one knows the word *manĝi* (to eat) and Esperanto's affix system, they can predictably create all those words: *manĝi*, *manĝo*, *manĝaĵo*, *manĝilaro*, *manĝujo*, *manĝaĉi*, *manĝeti*, *manĝigi*, *manĝejo*. Importantly, **the system of affixes means the last letter(s) of a word always indicate its grammatical role**.

## 4.3 Relevance

Crucially, Esperanto's grammar provides scope to probe the breadth and depth of LSTM grammatical abilities with more challenging (less controlled) tasks than the number agreement assessments used in previous studies. Moreover, the grammar is highly accessible to humans, and *presumably* this carries over to machines; if not, this implies a fundamental disconnect between biological and artificial language acquisition that may be interesting to computational neuroscience.

Three more factors make Esperanto a relevant choice. First, there are practical benefits for a student project where storage capacity on personal devices is limited; since the uniformity of Esperanto's grammar means we can expect interesting linguistic results even on small datasets, my relatively large dataset (large in the given context) should be plenty to enable a good comparison to the previous studies [58]. The simplicty and regularity of its grammar also simplifies grammatical tokenisation, enhancing the reliability of the data preparation stage of development and ultimately the quality of the data the LSTM is trained on. Finally, as some researchers have noted, the goal at the very core of Esperanto to overcome of language barriers is also the goal of many in the NLP community [58].

# 5. Summary of Background

## Context and Problem

NLU is AI-hard: grammar is hierarchical yet external communication requires "flattened", linear sequential output, meaning language data tends to be unstructured [12]. NLU thus requires a technology that can take linear, sequential input and implicitly recover its underlying syntactic structure, including when the grammar governs long-term dependencies.

Traditional RNNs are great sequence-processors but cannot capture long-term dependencies due to vanishing or exploding gradients, where a meaningfully-valued gradient cannot be conserved through the lengthy backpropagation required to learn from long-term dependencies. LSTMs are state-of-the-art RNNs that can handle long-term dependencies because its input, forget and output gates together maintain a sensible gradient throughout backpropogation. Some previous studies say LSTMs cannot induce grammar from unsupervised training on unstructured data. Others report genuine grammatical competence, though critics suggest this may stem from opportune, surface-level heuristics rather than grammatical competence.

## This Project

Two outstanding questions: (1) Are LSTMs able to discover true hierarchical information implicit in linear sequential data, or are they just shallow pattern-extractors capable of convincing statistical approximations to the effects of grammar? (2) How dependent is LSTM performance on specific hyperparameter arrangements? Esperanto is used as a test case to probe these questions, because its grammar provides scope to test the LSTM with a broader/less controlled set of grammatical tasks than number agreement when other variables are controlled.

Given that Gulordava (2018) and Lakretz (2019) developed LSTMs that demonstrated syntactic processing abilities, and in light of the explicit nature of Esperanto, it is hypothesized that the LSTM developed here will perform well on the affix-prediction tasks. A positive finding will support the previous studies and provide room to explore the LSTM's dependence on specific hyperparameter arrangements. Conversely, a poor performance might justify tentatively suggesting that previous studies were warped by some of the "many side effects" of natural language that Sennhauser and Berwick mention can influence LSTMs [12, p.116]. Or, perhaps the previous studies did discover genuine grammatical ability, but this is as far as LSTMs can go and for broader grammatical ability the NLU community should look to other architectures.

# Part II

# Implementation

# 6. Project Overview

## 6.1 Technologies used in Development

### 6.1.1 Hardware and Operating Systems

*MacOS, High-Performance Computing cluster, Blue Crystal Phase 4, Blue Pebble*

Most work was carried out on my personal laptop which uses MacOS Catalina. Training the LSTM involved submitting jobs to two of Bristol's High Performance Computing (HPC) platforms, Blue Crystal Phase 4 and Blue Pebble, used for their GPUs and high memory capacity. HPC platforms network many compute servers together into a cluster to deliver much higher performance than would be possible using a single computer.

### 6.1.2 Languages and Environments

*Python, PyCharm, Atom IDE, Slurm and PBS job scheduler*

Code was mostly written in Python (version 3.9.5), using the PyCharm integrated development environment by JetBrains and Atom IDE. I learnt Python using the Udemy course *2022 Complete Python Bootcamp From Zero to Hero in Python*. Only the shell scripts used to submit training jobs to Bristol's HPC cluster were not written in Python. Blue Crystal 4 uses the Slurm job scheduler to find compute nodes available to run the given code and then execute it. Blue Pebble at first used the PBS job scheduler for this purpose, but later switched to using Slurm. I learnt both scripting languages through documentation provided by the Advanced Computing Research Centre at University of Bristol, and with help from my supervisor [59, 60].

### 6.1.3 Libraries

*PyTorch, NumPy, Pickle, MatPlotLib*

#### PyTorch

Open-source libraries are sets of useful functions written by other developers and made pub-

licly available to eliminate the need for writing all code from scratch. Of most importance to this project is PyTorch, a machine learning library developed by Facebook's AI Research lab (FAIR) [61]. Key functionalities include the `torch` package for tensor computation, which can be used on CPU or GPU, and the `torch.nn` package which provides all the building blocks necessary for creating and training neural networks. Importantly, whereas many other libraries have a static view of the world, meaning one cannot alter a network's behaviour without re-coding it, PyTorch uses reverse-mode auto-differentiation to change network behaviour dynamically, without overhead for the coder [62]. The LSTM developed here is essentially built on top of PyTorch. Using a deep learning framework like PyTorch is common practice as it saves developers time writing complicated functions that are already publicly available within the framework. For example, Tesla Autopilot and Uber's probabilistic programming language "Pyro" are built on top of PyTorch [63, 64].

PyTorch was chosen for this project in favour of alternative, Google-supported deep learning framework Keras. Keras is a high-level API that wraps commonly used deep learning layers and operations into concise, simplified abstractions of the real complexities going on under the hood. PyTorch is comparatively verbose because it operates on a lower level with less abstraction, giving its user much fuller access to the bare bones of the functions involved in deep learning. Being thus confronted with the full complexity of things can initially be quite daunting for developers new to deep learning - I was certainly in this camp. However, it does give developers more freedom to write custom neural net layers and to more closely analyse and tweak numerical optimization tasks, essentially making PyTorch more flexible than Keras at a nuts-and-bolts level. Furthermore, although the conceptual hurdle is perhaps greater to begin with, ultimately the relative lack of abstractions does arguably encourage a deeper understanding of deep learning concepts.

### NumPy

NumPy ('Numerical Python') is a Python library that makes work involving large, multi-dimensional arrays and matrices such as those used in LSTM both faster and easier. This is because the NumPy array is more compact and consumes roughly four times less memory than a standard Python list [65]. While it is often helpful that a single Python list can contain different data types, input to the LSTM is represented numerically anyway, so the requirement that NumPy array elements be homogeneous does not cause any difficulties and makes mathematical operations performed on NumPy arrays much more efficient.

### Matplotlib

Matplotlib is the natural Python library of choice when it comes to dimensional plotting. It

used it during training, to plot loss against epochs, and when analysing the LSTM's results.

#### Pickle

Pickle is a module within the standard Python library that supports serialization and deserialization. Serialization is a process for converting a data structure into a linear form that can be stored or transported over a network - deserialization is the inverse. Training my LSTM model involves the creation of two dictionaries - `index_to_word` and `word_to_index` - which store unique words in the corpus from the most commonly occurring to either the least common or to the point at which the specified maximum vocabulary size truncates the rest. In my code serialization is used to transform these dictionaries into streams of bytes which are then written to file for storage. The dictionaries are deserialized at a later point when the data is needed.

### 6.1.4 Version Control

*GitHub, Overleaf*

I used GitHub for version control of my code and Overleaf for version control of this paper. Both platforms ensured my work was always securely backed-up, and made it easy for my supervisor and I to share code and comments in real-time.

## 6.2 Software

### 6.2.1 Development Cycle

Research was conducted from the start and throughout most of the project. The key stages of development were then as shown in the flow chart below. Project Implementation will be discussed in terms of these key stages. *Software development practices will be discussed later during project evaluation.*

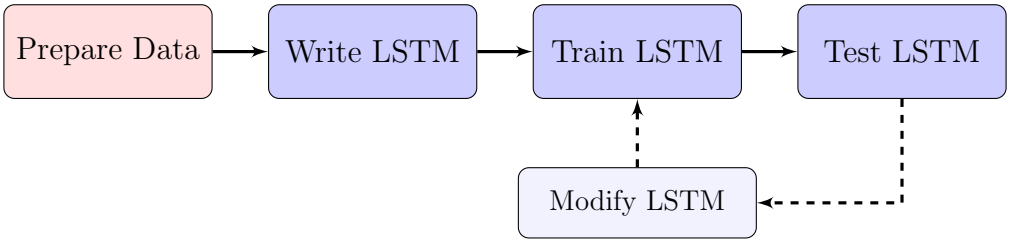



**Figure 6.1:** High-level development cycle.

### 6.2.2 Flow of Program Execution

The pink boxes in the flow chart below belong to the data preparation stage of development. For this, Esperanto data was obtained and separated into training and validation sets only once in the course of the project, however, the data must then be tokenized at least once for each distinct model. Each model must then be trained on its tokenized data and then tested on previously unseen data, in that order.

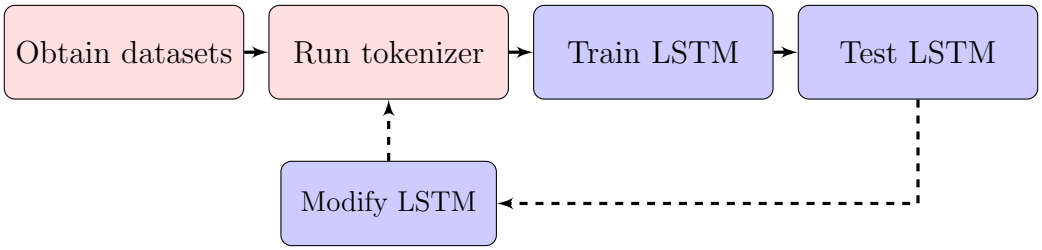
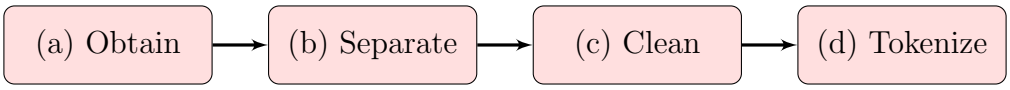



**Figure 6.2:** Ordered run of execution.

# 7. Data Preparation

Data preparation should produce a clean, tokenized corpus of Esperanto data for training the LSTM and a second such corpus for testing the LSTM during training, known as the validation test set. Data preparation for this project involved the stages listed in the flow chart below: stages (a) and (b) are executed once during the project, stages (c) and (d) are repeated for each new language model.




## 7.1 Obtain Data

*Consolidation* denotes the process of combining disparate data into a single repository. For this project a very large Esperanto corpus was obtained by concatenating the Esperanto versions of the open-source OSCAR corpus (Open Super-large Crawled Aggregated coRpus) and the literature and Wikipedia files from the Leipzig Corpora Collection [66, 67, 68]. After concatenation the plain text document file in which the data was stored contained 313.7 megabytes.

## 7.2 Separate Training and Validation Data

The point of a validation set is to have a new set of conditions, that were not part of the training corpus, with which to assess the network's prediction capability. The validation set is usually obtained by separating the last 20% of the data from the beginning 80% [69]. As my overall corpus was obtained by concatenating different corpora, I decided that simply taking a 20% chunk off the end of the original corpus might produce a situation where the validation set is heavily dominated by the Leipzig literature crawl (which contained lots of archaic language not commonly used today), while the training corpus might be dominated by more modern, everyday language use. While the validation set should be different from the training set, such a thematic slant is unreflective of the varied communication styles used in real life and might unnecessarily influence the loss value and prolong training.

The validation set is obtained by writing every hundredth line of the full corpus to the validation set file. The training set is then obtained by parsing every *remaining* line in the full corpus, checking whether any line is also present in the validation set, and every line that is not is written to the training set file. Afterwards both datasets are passed to a Boolean function

to verify that neither file contains a line that is also present in the other file. Both resultant datasets have to be cleaned and tokenized.

## 7.3 Clean Corpus

The purpose of cleaning (or 'cleansing') the corpus is to remove invalid, corrupted or irrelevant data. In this project, the tokenizer - file `make_indexed.py` - calls function `clean_file()`, which reads in the raw corpus and, for each line, does the following. First, all characters are normalised to lowercase and all non-alphabetic characters are removed, reducing the vocabulary size and generally simplifying the upcoming tokenization process without any real cost to the grammatical information relevant for this project. All Esperanto special characters (accented characters that don't exist in other languages) are then x-encoded, achieved by obtaining the Unicode code value of each character using the in-built method `ord()` and replacing each instance matching the code value of an Esperanto special character with their x-encoded equivalents.

## 7.4 Tokenize Data

Splitting the outermost affix (the grammatical cue) from each word was a personal choice that makes sense given the nature of my project. Creating word-index dictionaries and an index file is standard data pre-processing when working with LSTMs, since all input to an LSTM must be represented numerically.

### 7.4.1 Split Affixes

As mentioned in Chapter 4, in Esperanto the affix(es) contain the grammatical information relevant to this project. I therefore split affixes from their root words at this stage, so that the grammatical structure of a sentence is more immediately obvious in the input to the LSTM. Even though Esperanto has an agglomerative approach to word formation, meaning one root word can be suffixed by multiple affixes, for simplicity I only split the final affix at the end of the word - meaning at most one affix is split from each original word.

Affixes are split in conjunction with the aforementioned process of cleaning the data in order to save reading and writing so much data twice over. The file `constants.py` defines seventeen affixes of variable lengths and five roots from Esperanto's table of correlatives. The seventeen affixes are: four nouns, six adjectives, six verbs and one adverb. The tokenizer cleans the corpus line by line, as above, then passes each (now clean) word in the corpus to function

`split_affixes_word(word)`. This function first checks if the word ends in any of the affixes defined in `constants.py`. Since the affixes are of variable length and the closing characters of a longer affix might match the entirety of a shorter affix, the function groups the affixes by length and searches the input word for longer affixes first, only looking for shorter ones if no affix was already found. The return type is a string array containing either two elements or one. Either it returns the root word (what is left of the original word after being split from it's closing affix) and said closing affix, or it just returns the original word if no affixes were found. *Function* ***`split_affixes_word()`*** *and the list of defined and tokenized affixes can be found in the Appendix.*

### 7.4.2 Create Dictionaries

Now the corpus has been cleaned and all outermost affixes have been split from their root words (hereafter 'affix-split'), we can define the network's vocabulary. For this two dictionaries are created: `word_to_index` stores words as number indexes and `index_to_word` the reverse. Each word has the same index in both dictionaries, e.g. {`'o': 0, 'la': 1, ...`} and {`0: 'o', 1: 'la', ...`}. Note also that post-affix-split a 'word' could be an original or root word or an individual affix. The dictionaries are created from the training set and saved to file with the steps outlined below.

A short version of the corpus consisting of every hundredth line of the (cleaned and affix-split) original is read in. A 1D array `all_words[]` is created, the short corpus is traversed line-by-line, with each line split by whitespace and every resulting element appended to `all_words[]`. Using Python methods `Counter()` and `sorted()` on this array, a 1D list containing all *unique* words in the corpus is obtained and sorted from most to least commonly occurring word. The array is then truncated at the index specified by the pre-defined `vocab_size`, which for this LSTM is 5000. With the help of Python method `enumerate()` applied to the truncated array the two dictionaries are created. For each dictionary the word "unk" is placed at the vocab size index (here the 5000th index), essentially tagging the point at which the dictionary was truncated, so that when the LSTM encounters the tag it knows that datum is not important. Finally, the dictionaries are saved to file via serialization with Pickle.

### 7.4.3 Create Index File

The index file is a numerical representation of the (cleaned and affix-split) original corpus, where each word from the original corpus is represented by its index in the dictionary (or "unk" if it is not in the dictionary). In this project, Pickle is used to deserialize the `word_to_index` dictionary and the full-size cleaned and affix-split corpus is read in. For each word in each line, that word is used as a key to obtain its index from the dictionary, or the

index of `unk` if the word is absent from the dictionary, and the returned index value is written to the index file in string format. This is done using Python's Dictionary getter as such: `out_file.write(str(word_to_index.get(word.strip(), unk_n)) + " ")`.

## 7.5 Output Files

- Index-to-word dictionary
- Word-to-index dictionary
- Training set index file
- Validation set index file

## 7.6 Limitations

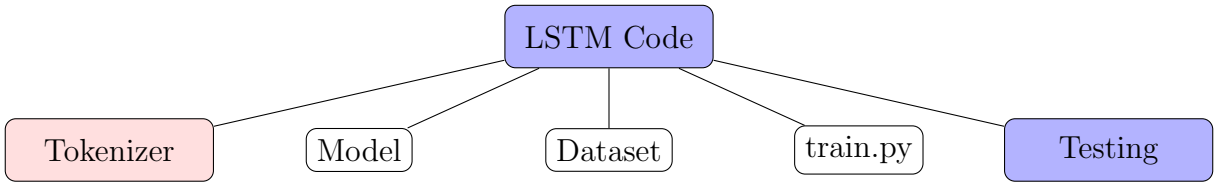
**Incompleteness.** Only a sub-set of Esperanto's affixes are tokenized, and only the outermost affixes are split from words in the corpus, ignoring the grammatical information more deeply embedded in agglomerate words. That said, while such incompleteness would be problematic if the LSTM was intended for release, that is far beyond the scope of this project. For investigating whether the LSTM is able to induce rules defining Esperanto's affix system, it is arguably best to balance completeness with simplicity in these early stages of research. The affixes tokenized here are amongst the most commonly used Esperanto affixes and cover a good range of grammatical tenses, numbers, moods and cases, and the outermost affix provides enough grammatical information to grasp the basic structure of a given sentence. *Tokenized affixes are listed in Chapter 10 and in the Appendix.*

**Time lost due to simple coding errors.** Various coding errors were made and only noticed and corrected after some time had passed. For instance, an additional '`\n`' in the code used to write the separate training and validation datasets to file caused errors down the line which resulted in the corpora being invalid. This was only picked up after attempts to train LSTMs on these corpora returned suspicious results, unnecessarily prolonging development.

# 8. LSTM Code

*The present LSTM and also my personal understanding were heavily informed by the online, publicly available tutorial 'PyTorch LSTM: Text Generation Tutorial' created by Domas Bitvinskas. Citation information can be found in the bibliography at number: [70].*

This chapter focuses on the Model and Dataset classes in my LSTM code and the training script that references them. *Code for both classes and the training script can be found in Appendix B.*




## 8.1 Model

The purpose of the Model class is to define the LSTM and initialise the neural network layers. Model inherits PyTorch class `nn.Module`, a base class for neural networks, then initialises the Embedding Layer, the core LSTM, and the linear layer. The **Embedding layer** converts word indexes to word vectors, functioning as an interface between the input layer and the LSTM cell. The benefit of word embedding is the geometric relationships between words in the embedding (in vector space) can represent semantic relationships between the words while using less space than methods like One-Hot encoding would use in this scenrio. The number of embeddings is set one greater than the vocabulary size (here 5001). Embedding dimension varies across language models - *see Hyperparameters section.* When the model does a forward pass during training it will look up words from the embedding layer, then during Backpropagation the network will adjust the embeddings to minimise the loss.

The **core LSTM** inherits from PyTorch class `nn.LSTM`. It initialises the input size and hidden size to 128, 350 or 650 [*see Hyperparameters section*] in line with the embedding dimension, representing the number of expected features in one input or hidden state (explained in Chapter 3 section on Data Structures), and the number of network layers is set to 2. A dropout of 0.2 is also set, which will randomly drop certain nodes during training in order to prevent over-fitting the training data. Finally, the **linear layer** uses PyTorch class `nn.Linear` to apply a linear transformation that maps from the hidden state space to the vocabulary size.

Once initialised the Model class has three functions. Function `to()` explicitly sets the model

to run on GPU or CPU. My training script uses PyTorch method `torch.cuda.is_available()` to determine whether the program is running on a GPU or a CPU (achieved by seeing if the system supports CUDA tensor types, which run on GPUs). The return value is passed to `torch.device()` - a zero sets the device to GPU and a one to CPU. The device is passed to the Model with `model.to(device)`. Function `forward()` takes current input and the previous state of the model (previous short-term and long-term memory) and does a forward pass, the return value of which is used by the training script to calculate the loss and then perform backpropagation. Finally, function `init_state()` is called by the training script at the start of every epoch in order to initialise the right shape of the hidden state and the cell state, both of which are PyTorch tensors.

## 8.2 Dataset

The purpose of the Dataset class is to load the corpus data into PyTorch using code that is decoupled from the model training code. Here the Dataset class inherits from PyTorch data primitive `torch.utils.data.Dataset` and stores data samples and their corresponding labels as follows. First, when initialising pass as arguments: the index file of the corpus (created in the tokenization stage), the sequence length and the vocabulary size. The index file is deserialised and each word index is appended to a 1D array, which function `to()` then converts into a PyTorch tensor. Method `__len__()` returns the number of samples in the dataset. Method `__getitem__()` loads and returns a sample from the dataset at the given index passed as an argument [71].

## 8.3 Training Code

The training script initialises the Model, a Dataset for the training corpus and another Dataset for the validation set corpus. These are passed to `train()` which uses PyTorch primitive `DataLoader` to iterate through the training and validation Datasets, returning features and labels from the dataset in batches defined by the batch size (explained in Chapter 3: Data Structures). *Batch sizes 256 and 32 were used.*

After loading and enumerating the data, a for-loop iterates through each **epoch** - a hyperparameter that controls the number of complete passes through the dataset during training. For each epoch the hidden and cell states are initialised, then training and validation are carried out through two nested for-loops. For the training dataset: for each batch number, feature and label the model makes a forward pass, calculates the loss, then steps through the process of backpropagation described earlier. Cross Entropy Loss was chosen for the loss function, which as described in Chapter 3 is used to calculate the error value for the current state of

the model so that the weights can be updated to reduce the error next time. The Adam optimization algorithm is used in place of classical stochastic gradient descent, because whereas the latter maintains a single learning rate (L.R.) for all weight updates and throughout training, Adam maintains individual adaptive learning rates for each parameter (network weight) and separately adjusts them all as learning unfolds. This makes for more controlled and unbiased weights throughout gradient descent [72, 73]. *Some models had learning rate 0.1, others 0.001.*

For the validation set the model does a forward pass and calculates the loss. The point of validation is to fine-tune model hyperparameters by providing an unbiased evaluation of the model's fit to the training dataset - hence the importance of data in one set not being contained in the other. The model's performance on the validation set provides a reference point that helps prevent the model over-fitting the training data (when the model is too closely aligned with the particular dataset and not useful on other datasets). Key to the model's lack of bias to the validation set is that the model does not directly learn from this set; hence backpropagation is never performed on the validation set.

## 8.4 Hyperparameters

The secondary goal of this project is to investigate how LSTM performance varies as the model's parameters are changed. The table below shows the parameter combinations for six LSTM models trained here. The **LSTM Size** is the number of features (units) in one input or hidden state. The **Embedding Dimension** is dimension of the vector space to which word indexes are mapped by the embedding layer. The **Batch Size** defines the number of samples to work through before updating the model's internal parameters. Finally, the **Learning Rate (L.R.)** defines the proportion that weights are updated during training, where larger values cause faster learning than smaller values. In this project the L.R. determines the initial rate before the Adam optimizer updates it during training [73].

| Model | LSTM size/Embedding dim | Batch Size | L.R. |
|---|---|---|---|
| 1 | 650 | 256 | 0.001 |
| 2 | 350 | 256 | 0.001 |
| 3 | 128 | 256 | 0.001 |
| 4 | 650 | 32 | 0.1 |
| 5 | 350 | 32 | 0.1 |
| 6 | 128 | 32 | 0.1 |

# 9. Training the LSTM

Training is how LSTMs learn and improve. Since the technicalities of training were explained in Chapter 3, this chapter will only discuss the key steps taken in training the language models developed here.

## 9.1 Execution

1. Prior to training, run tokenizer to obtain word-to-index and index-to-word dictionary files, indexed training set file and indexed validation set file. Upload files to Blue Pebble.
2. Upload LSTM Model and Dataset classes and training code to Blue Pebble. Also upload Slurm script, with directions to execute training code on a GPU node, and submit job.
3. Once training has completed, read the loss functions over time (printed to the slurm.out file) and plot losses against epochs to assess whether model has been sufficiently trained.
4. If satisfied with step (3), copy the final model from Blue Pebble to local computer for testing.

### 9.1.1 Step (3) Explained

As mentioned, the number of **epochs** controls the number of complete passes through the training dataset: one epoch equals one pass. The ideal number of epochs for which a model is trained will vary depending on factors like the model's hyperparameters and the size of the dataset. For instance, the graphs below show how the error value on the training and validation sets converges faster for the larger training dataset. Once the error value stabilises the model has usually been trained for enough epochs. With my dataset being so large, five epochs were enough for the loss to stabilise.


**Figure 9.1:** Large dataset.

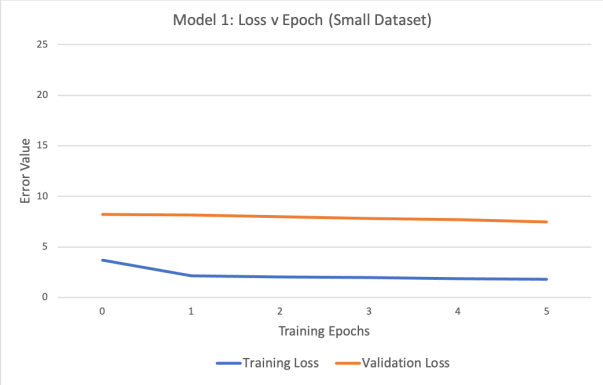



**Figure 9.2:** Small dataset.

## 9.2 Limitations

The training stage of this project was primarily limited by an almost one-month delay while I was getting to grips with how to schedule training jobs on the HPC clusters. I would wait days for jobs to finish only for them to eventually time-out. Thankfully my supervisor noticed an error in my submission scripts that meant I had been submitting the jobs to CPU nodes rather than GPU nodes as intended. I switched from Blue Crystal to Blue Pebble as I found the PBS scripts for Blue Pebble easier to write, and from then on training was much faster.

# 10. Testing the LSTM

## 10.1 Methodology

My methodology is inspired by Gulordava et al, who tested their models on number agreement tasks using nonsensical (yet still grammatical) sentences like "The colorless green **ideas** I ate with the *chair* **sleep** furiously." The point of the nonsensical sentences is to reduce the likelihood that the models could be relying on semantic or lexical cues in the data. The researchers also compared their model performance on the Italian dataset to human intuitions on a similar set of tasks. I too created nonsensical test sentences and a human control study for the same reasons as Gulordava et al. However, I tested the LSTM with a series of affix-prediction tasks and evaluated Affix Performance as the percentage times the LSTM assigned *highest* probability to the correct affix out of the fourteen possible affixes listed in the table below.

| Affix | Meaning |
|---|---|
| **o** | singular noun |
| **oj** | plural noun |
| **on** | singular accusative noun |
| **ojn** | plural accusative noun |
| **a** | singular adjective |
| **aj** | plural adjective |
| **ajn** | plural accusative adjective |

| Affix | Meaning |
|---|---|
| **as** | present time verb |
| **is** | past time verb |
| **os** | future time verb |
| **us** | conditional verb |
| **u** | commanding verb |
| **i** | infinitive verb |
| **e** | adverb |

**Figure 10.1:** The 14 affixes tokenized and assessed in this project.

My affix-prediction tasks are a broader test of grammatical ability than the Gulordava number agreement tests because of the evaluation procedure. Gulordava et al evaluation procedure was that for each test sentence, the researchers computed the probabilities assigned to words identical to the agreement target in all morphological features except number (e.g. "finds" instead of "find", but only looking at the probabilities assigned to verbs) [1, p.4]. They then say the model identified the correct target if it assigned a higher probability to the form with the correct number (singular vs plural). In other words, when evaluating number agreement they controlled the other variables of (i) category (noun, adjective, verb, adverb) (ii) tense (iii) accusative/non-accusative. In contrast, my tests do not restrict to a single variable like this, but rather, require the LSTM to predict the affix that agrees in number, tense, category and case all at once. This means that my tests can, for instance, pick up cases where the LSTM

assigns singular/plural correctly for the correct category (e.g. verb), but had as well assigned even higher probability to an inappropriate category (e.g. noun). Controlling category and other morphological features would result in the LSTM getting that question right, but by also testing tense, category and case, my tests could expose the limits of the LSTM's abilities.

### 10.1.1 Sourcing Esperanto Speakers (Esperantists)

I published a call for volunteers on online Esperanto forums and contacted organisers of Esperanto events and seminars requesting them to forward my call for volunteers. I was then fortunate to receive tremendous help from the individuals mentioned in the Acknowledgements chapter in sourcing more volunteers and constructing the test questions.

### 10.1.2 Obtaining Nonsense Sentences

#### Pilot

An initial set of nonsense Esperanto sentences were obtained in the following manner. First, for every tokenized grammatical category (noun, adjective, verb, adverb), a list of root words appropriate for the given category was created. A Python program was then written that first pulled a set of cleaned and tokenized Esperanto sentences from the testing and validation datasets. The program then parsed each sentence, and for each affix it detected, it used the list of root words defined for the given affix's category to replace the root word preceding that affix in the given sentence with another word from the list. This was intended to create a set of nonsensical sentences with entirely different meaning to the original sentences, but the same grammatical structure. Afterwards both datasets were parsed to verify that the nonsensical sentences did not appear in either.

Before proceeding with testing the LSTM I conducted a pilot human Esperantist study with the initial set of nonsense sentences. I was fortunate to receive feedback from fluent Esperanto speakers that the sentences were not ideal. Firstly, the methodology for creating the sentences had clearly produced words that don't exist in the Esperanto language. This seemed to result from two issues with the program. First, poor defining of constants when constructing the lists of root words to be substituted for the originals in the corpus, e.g. not all root words appropriate for a singular adjective affix would also be appropriate for a plural accusative adjective. Moreover, a bug in the code seemed to have substituted root words for different words, then encountered the same affix again and repeated substitution, resulting in agglomerate words that don't make sense in Esperanto.

**Final Question Set**

Rather than debug the program I adopted an entirely new approach, since I do not speak Esperanto myself and did not want to waste my volunteers' time proof-reading questions with similar problems. Instead, a group of fluent Esperanto speakers helped me to manually create a set of nonsensical yet grammatical sentences. This set of sentences were used to test the LSTM and for a full-sized human control study. Comments at the end of the full study confirmed that "the test sentences are perfectly meaningless" and "generally absurd," with one volunteer helpfully commenting, "I've never seen such a bunch of stupid, nonsensical phrases."

## 10.2 Affix Prediction Task

Each LSTM model was given fourteen questions. Each question consisted of a (cleaned and tokenized) Esperanto nonsense sentence with the last word missing its final affix, which as mentioned contains the grammatical information in Esperanto. The LSTM was tasked with predicting the next element in the sequence, i.e. the missing affix. For each affix of interest, the probability the LSTM assigns to it is compared to the probability assigned to the other thirteen affixes to establish its rank relative to the others. The full set of test sentences can be found in the Appendix.

## 10.3 Human Esperantist Study

The full size control study was conducted using Google Forms for the following reasons: the software is commonly used so many people are familiar with the format; it has the option to collect responses anonymously, which I availed of to protect volunteer privacy; it automatically collates responses in highly readable graphs and charts, saving me the time of transcribing and plotting results for over 260 individuals. For each question, volunteers were given a range of possible answers and asked to select the one they thought most likely correct. I also added an optional feedback window at the end of the quiz, which gave me great insights into people's thought processes for certain questions. As stated above, the questions were all vetted by fluent Esperanto speakers. The study included two additional questions which were not included in the LSTM test set, as discussed in the Evaluation section.

# Part III

# Results, Evaluation, Discussion

# 11. Results

## 11.1 The Results

The table headings below represent the following evaluation procedures. **Affix Performance (A.P.)** reflects the model's ability to predict the correct affix *most* likely out all assessed affixes (i.e. give correct affix rank 1 out of 14). **Category Performance (C.P.)** reflects the above but for the grammatical categories the affixes belong to: noun, adjective, verb, adverb. The probability of each category is the sum of the probability assigned to its members (its affixes) divided by the number of members. **Average Affix Rank (A.R.)** is the average rank assigned to an affix when that affix was the correct answer. This is a less all-or-nothing evaluation than Performance, since it reflects the LSTM's degree of closeness to the correct answer. **Average Category Rank (C.R.)** is the same but for the four categories to which the affixes belong. The final columns in the table show Affix Performance for *-o* and *-a*, and average rank for affixes *-o*, *-a* and *-e*, which were the three most frequently occurring affixes in the training dataset.

| Model | Hyperparameters | A.P. | Av. A.R. | C.P. | Av. C.R. | A.P. -o,-a | A.R. -o,-a,-e |
|---|---|---|---|---|---|---|---|
| 1 | 650,256,0.001 | 28.57% | 7.21 | 42.86% | 2.14 | 100% | 1 |
| 4 | 650,32,0.1 | 14.29% | 7.21 | 35.71% | 2.43 | 100% | 2 |
| 5 | 350,32,0.1 | 14.29% | 7.00 | 28.57% | 2.57 | 100% | 1.4 |
| 6 | 128,32,0.1 | 14.29% | 7.07 | 28.57% | 2.36 | 100% | 1.4 |
| 2 | 350,256,0.001 | 14.29% | 7.29 | 28.57% | 2.43 | 100% | 1.4 |
| 3 | 128,256,0.001 | 14.29% | 7.93 | 28.57% | 2.36 | 100% | 1.4 |

**Figure 11.1:** Results from affix-prediction tasks per **LSTM model**.

| Respondents | A.P. | C.P. |
|---|---|---|
| 263 | **97.0%** | **98.5%** |

**Figure 11.2: Human Esperantist** performance for all 263 respondents.

**Aim (1): Investigate grammatical abilities**

The overall LSTM performance across all affix prediction tasks was very disappointing, failing

to come close to the criteria of success that it should be comfortably above chance and also failing to come close to human Esperantist performance.

### Aim (2): Investigate effects of hyperparameter tuning

Performance variation across different architecture sizes/hyperparameters matches what Gulordava et al and Lakretz et al both found with their LSTMs. Namely, that careful tuning is "crucial" to success and specifically that larger hidden layer size/dimension does better. Though none of my models were successful by my criteria, the fact that A.P. for the best model is double that of the next-best when training data and tests were the same, implies that further parameter adjustments might bring improvements.

### Additional Observations

As shown in the last two columns of figure 11.1, all models predicted affixes *-o* and *-a* with 100% accuracy and performance for *-e* was not far behind. It was these three affixes on which the models did best overall. These three affixes also happen to be the top most frequently occurring affixing in the training dataset.

The graphs below represent data averaged across all LSTM models. The x-axes plot the average rank assigned to the correct affix for the given question (i.e. ideally all plots would be at rank 1), against the frequency of occurrence of each affix in the training dataset. Since affix *-o* occurred so much more frequently than any other affix, Figure 11.4 simply represents the same but excluding the average rank and frequency of affix *-o*. The outcome demonstrates a trend: as accuracy on the affix-prediction task decreases (i.e. lower rank), frequency of affix occurrence in the training dataset reduces in line with poorer ranking.

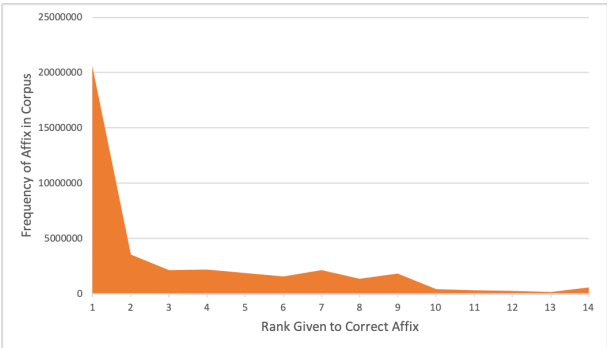



**Figure 11.3:** Rank v Frequency

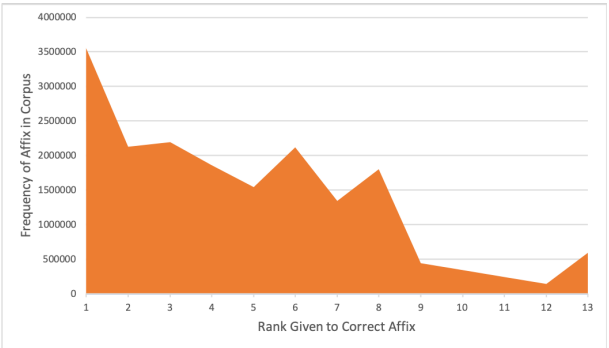



**Figure 11.4:** Rank v Frequency exc. -o

## 11.2 Analysis of Results

From the data I can see three possible explanations: (1) LSTMs are not grammatical and previous studies have misidentified subtle, advanced statistical processing as grammatical ability. (2) Previous studies did uncover genuine grammatical ability in LSTMs, however, this ability is limited to fairly controlled scenarios that are not reflective of real life grammar requirements - *in my opinion the most likely explanation.* (3) The models all overfit the training data and training should be repeated after alterations to the models.

### 11.2.1 Not Grammatical

As mentioned, previous studies that found evidence for LSTMs having grammatical encoding abilities - Gulordava et al (2018) and Lakretz et al (2019) - trained and tested their models on natural language datasets. Sennhauser and Berwick point out that natural language datasets typically contain "nuisance variables" of a statistical nature that can correlate with the correct grammatical targets for given cues, causing the models to acquire a statistical approximation to the syntactic rules which does not equate to genuine grammatical ability. Since the Gulordava results, findings by McCoy et al (2020) show that statistical neural network architectures are prone to adopting shallow heuristics, based on superficial properties of the data, instead of learning the underlying generalizations they are meant to capture [74]. This enables them to score highly on test sets drawn from the same distribution as the training set, but the models break down in more challenging cases since their exploitation of invalid heuristics does not equate to deeper understanding of language. The authors conclude that "targeted, challenging datasets... are important for determining whether models are learning what they are intended to learn" [74, p.9]. Perhaps my test case using Esperanto has exposed a flaw in the nature of natural language datasets which could have allowed the models to do well on the number agreement tasks due to something other than true hierarchical competence.

In favour of this suggestion is the aforementioned correlation between performance and affix frequency. The fact all models can predict the most commonly occurring affixes (*-o*, *-a* and *-e*) exceptionally well, but then all do shockingly badly overall because performance is dragged down by an inability to get almost any of the less common affixes correct, does imply that the models may still be very closely bound up with statistics. Nonetheless, I doubt that the Gulordava and Lakretz results are solely due to mere statistical approximations. Their LSTMs performed comfortably above chance and roughly on a par with human performance even on nonsensical sentences with agreement attractors, and their results complement each other, demonstrating reproducibility. In my judgement, the combination of these factors together rule out the possibility the such good results stem from a misidentification of statistical variables. Instead, I think it more plausible that the LSTMs are grammatical but only in limited contexts.

### 11.2.2 Grammatical but Limited

I reason that the correlation between good performance and affix frequency uncovered here might indicate that the LSTM may resort to shallow, statistical heuristics when the testing scenario is relatively uncontrolled - as mine is. There is no reason to assume that this tendency is mutually exclusive with grammatical competence in a more controlled setting (i.e. when checking for number agreement, control all other morphological features). My reasons not to doubt the Gulordava and Lakretz studies were given above. Further support for this explanation of my results could be sought by repeating my experiments with testing questions that control all variables but one, such as number.

### 11.2.3 Model Over-fitting Training Data

Part of the reason I investigated various parameter combinations is because the architecture and hyperparameters with which a model is training can greatly influence its performance. Perhaps all my models overfit the training data - meaning they learnt the training data too carefully including any statistical noise, resulting in poor performance when tested on new data [75].

While it is entirely possible that undetected training errors may have contributed to the disappointing results, I do not think the models specifically overfit the training set. A key indicator of overfitting is when the training loss decreases as the validation loss rises across training epochs. As the graphs in Figures 9.1 and 9.2 show, this is not the case for my LSTM. My large and varied training dataset and the inclusion of a Dropout arguably helped protect against such a scenario.

# 12. Project Evaluation

## 12.1 Project Methodology

I have evaluated my project and its limitations at various steps throughout this paper and I will not repeat the same here. Further to already discussed limitations, I believe the project would have been better had testing procedures been more thoroughly planned from the outset. The human Esperantist study had sixteen questions, two of which had to be omitted from the LSTM test set as they required the agent to predict the affix for a word occurring in the middle of the sentence. This was not suitable for the LSTM which does not use future input to make a prediction for a missing time step in the middle of an input sequence. Had LSTM and human test sets both been finalised in advance this could have been avoided.

In line with the above, I also believe my evaluation procedure for assessing the LSTM may have left too many variables uncontrolled at once. In hindsight it might have been a more reliable assessment had I only introduced one additional variable besides number agreement in the affix-prediction tasks.

## 12.2 Personal Working Practices

In terms of software development practices, my initial plan was to informally use the Agile framework. I made this decision because Agile worked well for me and my team during the term-time group project and I thought it's workflow (ongoing sprints of project planning and execution) would keep me accountable and on-schedule. As it transpired, I did not stick to my Agile time plan at all after the initial few weeks of the project.

I believe it was a mistake to attempt to use Agile in the first place, as its tenets are in fact inappropriate for the nature of this project. The Agile manifesto emphasises individuals and interactions over processes and tools. This doesn't make sense for a one-person (though supervised) project without external collaboration. The manifesto also emphasises working software over comprehensive documentation, which is not a pressing issue here given that the usefulness of my project comes from contributing to on-going research. Finally, Agile also emphasises responding to change over following a plan. This is somewhat relevant, but on the whole the brunt of the work involved in this project was predictable, and the absence of any commercial stakeholders meant initial project requirements were highly unlikely to change unexpectedly.

# 13. Conclusions

The main aim of this project was to investigate whether an LSTM could learn to model Esperanto grammar, and by the criteria of success specified here, it failed. This is a disappointing result as previous studies have reliably demonstrated grammatical ability in LSTMs when the testing scenario isolates individual morphological variables. However, I don't believe that my results invalidate these previous findings. From my results it would appear that LSTMs simply cannot model grammatical structure in less controlled settings, suggesting LSTMs' grammatical abilities are existent but narrow, since they don't perform well without some variables being controlled that wouldn't naturally be controlled in a real-life scenario. Statistical correlations between affix rank and affix frequency make me think the LSTM might just resort to statistical heuristics when variables are insufficiently controlled.

Going forward, I see two possible routes. On the one hand, improvements to the current project are needed to support my conclusions, and might be used to determine the exact point at which LSTMs reach their grammatical limit. The other route is to explore fundamentally different architectures for grammatical tasks. In my opinion the two routes should be explored collaboratively. Once we know the limits of LSTMs (if such limits exist), we would be better placed to know at what point to introduce different architectures to form an overall hybrid application for NLU use-cases.

## 13.1 Improvements to Current Project

- **Affix-prediction evaluation procedure**. Start by controlling all but two variables (e.g. number and tense), then step-wise remove controls and test in an attempt to delimit where LSTM grammatical abilities fail.
- **More hyperparameter tuning**. Since Model 1 achieved A.P. double that of the next-best model, I suggest similarly tuning hyperparamaters in step-wise fashion in the direction from model 1 to see if additional improvements can be attained.

## 13.2 Future Work in the Field

If my conclusions are correct and LSTMs have grammatical abilities but only within limited, controlled contexts, it might be wise to use LSTMs but in tandem with other architectures. When Hu et al (2020) conducted a systematic assessment of syntactic generalisation in neural

language models, they found that sequential models underperformed other architectures in general, and moreover their LSTM was the poorest performer of all. Furthermore, the researchers found that different architectures have different relative advantages across types of syntactic tests, suggesting that combining the capacities of a few architectures will make broader grammatical ability more likely [76]. One possible avenue of future research could be to explore Transformer-based solutions to grammatical problems and how such solutions could be combined with LSTMs. Another possible route is to explore possible applications of Attention mechanisms for solving grammatical problems.

# Bibliography


[1] K. Gulordava, P. Bojanowski, É. Grave, T. Linzen, and M. Baroni, "Colorless green recurrent networks dream hierarchically," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1195–1205, 2018.

[2] Y. Lakretz, G. Kruszewski, T. Desbordes, D. Hupkes, S. Dehaene, and M. Baroni, "The emergence of number and syntax units in lstm language models," *arXiv preprint arXiv:1903.07435*, 2019.

[3] I. C. Education, "Neural networks," 17-08-2020. [Accessed: 01-12-2021].

[4] E. Kavlakoglu, "AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?," 27-05-2020. [Accessed: 01-12-2021].

[5] C. Olah, "Understanding LSTM Networks," 15-08-2015. [Accessed: 12-12-2021].

[6] S. Verma, "Understanding input and output shapes in lstm," 14-01-2019. [Accessed: 27-12-2021].

[7] MongoDB, "Unstructured data," 2021. [Accessed: 01-12-2021].

[8] R. V. Yampolskiy, "Turing test as a defining feature of ai-completeness," in *Artificial intelligence, evolutionary computing and metaheuristics*, pp. 3–17, Springer, 2013.

[9] J. L. Elman, "Distributed representations, simple recurrent networks, and grammatical structure," *Machine learning*, vol. 7, no. 2, pp. 195–225, 1991.

[10] A. Gandhe, A. Rastrow, and B. Hoffmeister, "Scalable language model adaptation for spoken dialogue systems," in *2018 IEEE Spoken Language Technology Workshop (SLT)*, pp. 907–912, IEEE, 2018.

[11] A. Packer, "Natural language understanding in alexa," October 2019.

[12] L. Sennhauser and R. C. Berwick, "Evaluating the ability of lstms to learn context-free grammars," *arXiv preprint arXiv:1811.02611*, 2018.

[13] T. Linzen, E. Dupoux, and Y. Goldberg, "Assessing the ability of lstms to learn syntax-sensitive dependencies," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 521–535, 2016.

[14] A. Kuncoro, C. Dyer, J. Hale, and P. Blunsom, "The perils of natural behaviour tests for unnatural models: the case of number agreement," *Poster presented at Learning Language in Humans and in Machines, Paris, Fr., July*, vol. 5, no. 6, 2018.

[15] T. Linzen and B. Leonard, "Distinct patterns of syntactic agreement errors in recurrent networks and humans," *arXiv preprint arXiv:1807.06882*, 2018.

[16] E. L. Schwartz, *Computational Neuroscience*. MIT Press, 1993.

[17] T. J. Sejnowski, C. Koch, and P. S. Churchland, "Computational neuroscience," *Science*, vol. 241, no. 4871, pp. 1299–1306, 1988.

[18] S. Alla, "A brief introduction to Computational Neuroscience part 1," *Towards Data Science*, 06-01-2019. [Accessed: 22-11-2021].

[19] K. E. Bouchard, J. B. Aimone, M. Chun, T. Dean, M. Denker, M. Diesmann, D. D. Donofrio, L. M. Frank, N. Kasthuri, C. Koch, *et al.*, "High-performance computing in neuroscience for data-driven discovery, integration, and dissemination," *Neuron*, vol. 92, no. 3, pp. 628–631, 2016.

[20] M. Shardlow, M. Ju, M. Li, C. O'Reilly, E. Iavarone, J. McNaught, and S. Ananiadou, "A text mining pipeline using active and deep learning aimed at curating information in computational neuroscience," *Neuroinformatics*, vol. 17, no. 3, pp. 391–406, 2019.

[21] N. Savage, "How ai and neuroscience drive each other forwards," *Nature*, 24-07-2019. [Accessed: 22-11-2021].

[22] X. Fan and H. Markram, "A brief history of simulation neuroscience," *Frontiers in neuroinformatics*, vol. 13, p. 32, 2019.

[23] I. C. Education, "Natural Language Processing (NLP)," 02-07-2020. [Accessed: 22-11-2021].

[24] U. C. London, "Cognitive neuroscience of language." [Accessed: 23-11-2021].

[25] M. P. I. for Psycholinguistics, "Neurobiology of language department." [Accessed: 22-11-2021].

[26] N. N. Lab, "Pylkkänen Group." [Accessed: 24-11-2021].

[27] N. of Language, "Call for papers - special issue: Cognitive computational neuroscience of language," 2021. [Accessed: 27-11-2021].

[28] A. L. Chandra, "Mcculloch-pitts neuron — mankind's first mathematical model of a biological neuron," *Towards Data Science*, 24-07-2018. [Accessed: 22-11-21].

[29] V. Bansal, "The evolution of deep learning," *Towards Data Science*, 5-04-2020. [Accessed: 22-11-21].

[30] F. Chollet, *Deep learning with Python*. Manning Publications Co., 2018.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] L. Hardesty, "Explained: Neural networks. ballyhooed artificial-intelligence technique known as "deep learning" revives 70-year-old idea.," 14-04-2017. [Accessed: 22-11-2021].

[33] A. for Computational Linguistics, "Call for papers," 13-03-2019. [Accessed: 27-11-2021].

[34] M. Schrimpf, I. A. Blank, G. Tuckute, C. Kauf, E. A. Hosseini, N. Kanwisher, J. B. Tenenbaum, and E. Fedorenko, "The neural architecture of language: Integrative modeling converges on predictive processing," *Proceedings of the National Academy of Sciences*, vol. 118, no. 45, 2021.

[35] A. Trafton, "Artificial intelligence sheds light on how the brain processes language," 25-10-2021. [Accessed: 27-11-21].

[36] J. Lyons, *Natural Language and Universal Grammar: Essays in Linguistic Theory*. Cambridge University Press, 1991.

[37] Unknown, "What makes nlp difficult?." [Accessed: 01-11-2021].

[38] M. J. Garbade, "A simple introduction to natural language processing," 15-10-2015. [Accessed: 01-11-2021].

[39] E. Kavlakoglu, "NLP vs. NLU vs. NLG: the differences between three natural language processing concepts," 12-11-2020. [Accessed: 01-12-2021].

[40] I. C. Education, "Recurrent neural networks," 14-09-2020. [Accessed: 12-12-2021].

[41] M. Phi, "Illustrated Guide to Recurrent Neural Networks," 20-09-2018. [Accessed: 20-12-2021].

[42] J. Nabi, "Recurrent neural networks (rnns)," 11-09-2019. [Accessed: 20-12-2021].

[43] D. M. Andrew Blais, "An introduction to neural networks," *IBM Developer*, 01-07-2001. [Accessed: 20-12-2021].

[44] J. Brownlee, "A gentle introduction to backpropagation through time," 23-06-2017. [Accessed: 20-12-2021].

[45] N. Donges, "A Guide to RNN: Understanding Recurrent Neural Networks and LSTM Networks," 29-07-2021. [Accessed: 20-12-2021].

[46] A. U. aditianu1998, "Understanding of lstm networks," 25-06-2021. [Accessed: 27-12-2021].

[47] J. Johnson, "Lstms," *The Shape of Data*, 04-06-2016. [Accessed: 27-12-2021].

[48] J. Brownlee, "Difference between a batch and an epoch in a neural network," 26-10-2019. [Accessed: 27-12-2021].

[49] B. Caulfield, "What's the difference between a cpu and a gpu?," 16-12-2009. [Accessed: 27-12-2021].

[50] D. Rotman, "We're not prepared for the end of moore's law," *MIT Technology Review*, 2020.

[51] D. Vellante, "A new era of innovation: Moore's law is not dead and ai is ready to explode," *Silicon Angle*, 10-04-2021.

[52] J. Misachi, "10 facts about esperanto, the world's international language," *World Atlas*, 08-12-2020. [Accessed: 19-11-2021].

[53] R. P, "The wonderful horrible history of esperanto, the universal language," *Owlcation*, 11-12-2016. [Accessed: 20-11-2021].

[54] O. B. Waxman, "The serious history behind esperanto," *TIME*, 26-07-2016. [Accessed: 19-11-2021].

[55] *[Author Unknown]*, "The League: Esperanto Spurned," *TIME, Foreign News*, vol. 24, 13-09-1923. [Accessed: 19-11-2021].

[56] T. Owen and J. Meyer, *Complete Esperanto*. Hodder and Stoughton, 2018.

[57] "Esperanto grammar." [Accessed: 19-11-2021].

[58] H. Face, "How to train a new language model from scratch using Transformers and Tokenizers," 14-02-2020. [Accessed: 19-11-2021].

[59] A. C. R. Centre, "Bluecrystal phase 4 user documentation," 2018. [Accessed: 27-12-2021].

[60] A. C. R. Centre, "Acrc hpc documentation and user guides," 2019. [Accessed: 27-12-2021].

[61] M. Patel, "When two trends fuse: Pytorch and recommender systems," 07-12-2017. [Accessed: 27-12-2021].

[62] P. Tutorials, "Build the neural network," 2021. [Accessed: 27-12-2021].

[63] A. Karpathy, "Pytorch at tesla," in *PyTorch Devcon Conference*, vol. 19, 2019.

[64] N. Goodman, "Uber AI labs open sources Pyro, a deep probabilistic programming language," 2017.

[65] A. Babenhauserheide, "Memory requirement of python datastructures: numpy array, list of floats and inner array," 12-16-2014. [Accessed: 27-12-2021].

[66] P. J. Ortiz Su'arez, B. Sagot, and L. Romary, "Asynchronous pipelines for processing huge corpora on medium to low resource infrastructures," in *Proceedings of the Workshop on Challenges in the Management of Large Corpora (CMLC-7) 2019. Cardiff, 22nd July 2019* (P. Bański, A. Barbaresi, H. Biber, E. Breiteneder, S. Clematide, M. Kupietz, H. L"ungen, and C. Iliadi, eds.), (Mannheim), pp. 9 – 16, Leibniz-Institut f"ur Deutsche Sprache, 2019.

[67] P. J. Ortiz Su'arez, L. Romary, and B. Sagot, "A monolingual approach to contextualized word embeddings for mid-resource languages," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 1703–1714, Association for Computational Linguistics, 07-2020.

[68] D. Goldhahn, T. Eckart, U. Quasthoff, *et al.*, "Building large monolingual dictionaries at the leipzig corpora collection: From 100 to 200 languages.," in *LREC*, vol. 29, pp. 31–43, 2012.

[69] D. Optimization, "Lstm networks," 24-02-2021. [Accessed: 27-12-2021].

[70] D. Bitvinskas, "Pytorch lstm: Text generation tutorial," July 2020. =https://www.kdnuggets.com/2020/07/pytorch-lstm-text-generation-tutorial.html, [Accessed: 27-12-2021].

[71] P. Tutorials, "Datasets and dataloaders," 2021. [Accessed: 27-12-2021].

[72] Prakhar, "Intuition of adam optimizer," 24-10-2020. [Accessed: 27-12-2021].

[73] J. Brownlee, "Gentle introduction to the adam optimization algorithm for deep learning," 03-07-2017. [Accessed: 27-12-2021].

[74] R. T. McCoy, E. Pavlick, and T. Linzen, "Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference," *arXiv preprint arXiv:1902.01007*, 2019.

[75] J. Brownlee, "A gentle introduction to dropout for regularizing deep neural networks," 03-12-2018. [Accessed: 27-12-2021].

[76] J. Hu, J. Gauthier, P. Qian, E. Wilcox, and R. P. Levy, "A systematic assessment of syntactic generalization in neural language models," *arXiv preprint arXiv:2005.03692*, 2020.

# A. Appendix A

## A.1 Affixes defined in `constants.py`

Singular noun: -o. Plural noun: -oj. Singular accusative noun: -on. Plural accusative noun: -ojn. Singular adjective: -a. Plural adjective: -aj. Plural accusative adjective: -ajn. Comparative adjective: pli. Superlative adjective: plej. Comparative conjunction adjective: ol. Present time verb: -as. Past time verb: -is. Future time verb: -os. Conditional verb: -us. Command: -u. Infinitive verb: -i. Adverb: -e.

## A.2 Affix Prediction Task: Sentences & Affixes Tested

| #Q | Sentence | Correct Affix |
|---|---|---|
| 1 | La aligatoro rapide manĝis la panon, do ne restis pano pro la aligatoro. | **o** |
| 2 | Ĉiuj miaj fingroj havas inkajn makulojn. Estas makuloj sur ĉiuj miaj fingroj. | **oj** |
| 3 | Mi havas ferocan katidon. | **on** |
| 4 | Mi havas du tre inteligentajn trankaĵojn da pano, kiuj neniam ĉesas esti stultaj. Mi havas du tre inteligentajn trankaĵojn. | **ojn** |
| 5 | La bela, betona vojo ŝatas pentri kiam la suno brilas, kaj la vojo estas vere bela. | **a** |
| 6 | La biblioteko estis malplena kvankam multaj ĉokoladoj estis konstante uzitaj por fosi, vere tre multaj. | **aj** |
| 7 | Ni ĝojus, se iu povus trovi bonajn. | **ajn** |
| 8 | Kuri maratonojn kaj kuiri samtempe estas io, kion mi malamas. | **as** |
| 9 | Por eviti mian bopatrinon, en la lavejp mi restis. | **is** |
| 10 | Al la luno kaj reen, tien mi morgaŭ naĝos. | **os** |
| 11 | Se vi pafus la buteron, la pano ĝojus. | **us** |
| 12 | Nun aldonu. | **u** |
| 13 | Paŝi al la dolĉa envolvaĵo por elŝuti. | **i** |
| 14 | Post kiam mi finis krii ĉe la muro, mi estis petis de la heliko daŭrigi trankvile. | **e** |

# B. Appendix B: Code

## B.1 Split Affixes

```
def split_affixes_word(word):

    word_affixes = []
    word_l = len(word)
    if contains_affix(word):
        word, word_affixes = split_aff(word, word_affixes, 4)
        for affix_l in [3, 2, 1]:
            if len(word) == word_l:
                word, word_affixes = split_aff(word, word_affixes, affix_l)

    if len(word_affixes) != 0:
        return [word] + word_affixes
    return [word]
```

Function `split_affixes_word(word)`

## B.2 Blue Pebble `Slurm` and `PBS` scripts

```
#!/bin/sh
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=90:00:00
#SBATCH --mem=10gb
#SBATCH --gres=gpu:2
#SBATCH --partition gpu
#SBATCH --job-name=train

module load lang/cuda
module load lang/python/anaconda/pytorch

python train.py --corpus training.txt \
                --test_corp validation.txt \
                --max-epochs 5
```

Slurm train script

```
#!/bin/sh
#PBS -l select=1:ncpus=1:ngpus=1:mem=4gb
#PBS -l walltime=50:00:00

module load lang/cuda
module load lang/python/anaconda/pytorch

cd $HOME/EsperantoLSTM/
python train.py --corpus training.txt \
                --test_corp validation.txt \
                --max-epochs 10
```

PBS train script

## B.3 LSTM Class `Model`

```
class Model(nn.Module):
    def __init__(self, n_vocab):
        super(Model, self).__init__()
        self.lstm_size = 350
        self.embedding_dim = 350
        self.num_layers = 2

        # Embedding - converts word indexes to word vectors
        self.n_vocab = n_vocab

        self.embedding = nn.Embedding(
            num_embeddings=n_vocab,
            embedding_dim=self.embedding_dim)

        # LSTM - the main learnable part of the network
        self.lstm = nn.LSTM(
            input_size=self.lstm_size,
            hidden_size=self.lstm_size,
            num_layers=self.num_layers,
            dropout=0.2)

        self.fc = nn.Linear(self.lstm_size, n_vocab)

    def to(self, device=None):
        self = super().to(device)
        self.embedding = self.embedding.to(device)
        self.lstm = self.lstm.to(device)
        self.fc = self.fc.to(device)
        return self

    def forward(self, x, prev_state):
        embed = self.embedding(x)
        output, state = self.lstm(embed, prev_state)
        logits = self.fc(output)
        return logits, state

    def init_state(self, sequence_length,device):
        return torch.zeros(self.num_layers, sequence_length, self.lstm_size, \
                device=device),
               torch.zeros(self.num_layers, sequence_length, self.lstm_size, \
                device=device)
```

## B.4 LSTM Class Dataset

```
class Dataset(torch.utils.data.Dataset):

    def __init__(self, corpus,sequence_length,vocab_size):
        self.corpus = corpus
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size

        index_name=corpus[:-4]+"."+str(vocab_size)+".index.txt"

        index_file=open(index_name,'r')

        lines = index_file.readlines()

        self.words_indexes=[]

        for line in lines:
            data=line.split()
            for word in data:
                self.words_indexes.append(int(word))

    def to(self,device):
        self.words_indexes=torch.tensor(self.words_indexes,device=device)

    def __len__(self):
        return len(self.words_indexes) - self.sequence_length

    def __getitem__(self, index):
        return (self.words_indexes[index:index + self.sequence_length],
                self.words_indexes[index + 1:index + self.sequence_length + 1])

```

## B.5 LSTM Training Script

```
def train(dataset, test_dataset, model, args,device):

    model.train()

    dataloader = DataLoader(dataset, batch_size=args.batch_size)
    dataloaderTEST = DataLoader(test_dataset, batch_size=args.batch_size)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    out_file = open(args.corpus[:-4]+"."+str(args.vocab_size)+".log", "a")

    for epoch in range(args.max_epochs):

        state_h, state_c = model.init_state(args.sequence_length,device)
        print(state_h.is_cuda)

        # Training set
        for batch, (x, y) in enumerate(dataloader):
            optimizer.zero_grad()
            y_pred, (state_h, state_c) = model(x, (state_h, state_c))
            loss = criterion(y_pred.transpose(1, 2), y)
            state_h = state_h.detach()
            state_c = state_c.detach()
            loss.backward()
            optimizer.step()

            # Code hidden for Appendix brevity: prints training loss for given
            # batches to out file
            if batch == 1:…
            if batch % 1000 == 0:…

        # Testing/Validation Set
        model.eval()
        total_loss = 0.0
        counter = 0
        for batch, (x, y) in enumerate(dataloaderTEST):
            counter += 1
            y_pred, (state_h, state_c) = model(x, (state_h, state_c))
            loss = criterion(y_pred.transpose(1, 2), y)
            state_h = state_h.detach()
            state_c = state_c.detach()

            # Code hidden for Appendix brevity: prints validation loss for
            # given batches to out file
            if batch == 1:…

            total_loss += loss.item()

        total_loss = (total_loss/counter)
        print('VALIDATION SET LOSS', total_loss)

        model.train()
        filename = args.corpus[:-4]+"."+str(args.vocab_size)+".model."+str(epoch)+".pt"
        torch.save(model.state_dict(), filename)
```