University of Amsterdam

MSc Computational Science
Master of Science Thesis

Pondering in Artificial Neural Networks
Inducing systematic compositionality in RNNs

Anand Kumar Singh
(11392045)

Thursday 16th August, 2018

Informatics Institute · University of Amsterdam

Pondering in Artificial Neural Networks

Inducing systematic compositionality in RNNs

Master of Science Thesis

For obtaining the degree of Master of Science in Computational Science at University of Amsterdam

Anand Kumar Singh

Thursday 16th August, 2018

Student number:	11392045	
Thesis committee:	Dr. ir. Elia Bruni,	ILLC UvA, Supervisor
	Ir. Dieuwke Hupkes,	ILLC UvA, Supervisor
	Dr. ir. Rick Quax,	UvA, Examiner
	Prof. dr. Drona Kandhai,	UvA, Second Assessor
	Dr. ir. Willem Zuidema,	UvA, Third Assessor

An electronic version of this thesis is available at http://scriptiesonline.uba.uva.nl.

Informatics Institute · University of Amsterdam

UNIVERSITY OF AMSTERDAM

University of Amsterdam
Department Of
Computational Science

The undersigned hereby certify that they have read and recommend to the Department of Computational Science for acceptance a thesis titled **"Pondering in Artificial Neural Networks"** by **Anand Kumar Singh** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: Thursday 16th August, 2018

Supervisor:

Dr. ir. Elia Bruni

Supervisor:

Ir. Dieuwke Hupkes

Abstract

Abstract

Acknowledgements

Acknowledge

Amsterdam, The Netherlands
Thursday 16th August, 2018

Anand Kumar Singh

Contents

List of Figures

List of Tables

Nomenclature

Latin Symbols

μ_f	dynamic viscosity of fluid
ν_f	kinematic viscosity of fluid
ρ_f	density of fluid
ρ_f	density of particle
$C_{p,f}$	specific heat capacity of fluid
$C_{p,p}$	specific heat capacity of particle
k_f	thermal conductivity of fluid

Chapter 1

Introduction

"Can Start with some quote" How deep learning has taken the world by storm since Alexnet in 2012. Examples of some areas where it has reached human/super-human level performance.

However data hungry and thus poor at concept learning. Where has it failed?

The biggest criticism of deep learning is its overt dependence on voluminous data which has led many researchers to argue that deep networks are only good at finding pattern recognition in training distribution (give refs) and therefore conform to the basic tenet of statistical machine learning that train and test data should come from the same distribution(give refs). Deep neural networks therefore are still poor at zero shot generalization to test data which despite coming from the same 'rule space' don't follow the exact same distribution as training data (find a good example here). Human reasoning on the other hand is governed by a rule based systematicity (give refs) which leads us to learn complex concepts from small samples which leads to zero shot generalization (example).

1.1 Motivation

Humans exhibit algebraic compositionality in their thought and reasoning (ref). We discuss the concept of compositionality and breifly review how current deep networks deal with it. The review of these concepts is essential since they are my guiding principles for this work.

1.1.1 Systematic Compositionality

Compositionality is the principle of understanding a complex expression through the meaning and syntactic combination of its morphemes. This definition of compositionality does not directly imply dependence on context in which the expression appears or the intent of the speaker and therefore the compositional nature of natural language is an

active area of debate among linguists and philosophers [Szab, 2017]. One of the strongest arguments in the favour of compositionality in natural language is the presence of systematicity (cite Fodor). **Systematicity**. Provided [E1, E3] of same grammatical category and [E2,E4] of same grammatical category. If [E1,E2] can combine syntactically then so can [E1, E4], [E3, E2] and [E3,E4] and if one can understand ([E1,E2]) and ([E3, E4]) then they can also understand ([E1,E4]) and ([E3,E2]) provided that the is well formed.

Aritifical languages (such as the ones we'll encounter in chapter 6) on the other hand can be constructed to strictly follow the principles of systematic compositionality. Given human propensity for compositional solutions [Schulz et al., 2016] these datasets provide excellent test-beds for validating the existence of compositional skills in neural networks.

1.1.2 Compositionality in Seq2Seq Models

Lake and Baroni [2017] introduced the SCAN dataset which maps a string of commands to the corresponding string of actions and is therefore a natural setting for seq2seq models. The commands consist of *primitives* such as jump, walk, run etc.; *direction modifiers* such as left, right, opposite etc.; *counting modifiers* such as twice, thrice and *conjuctions* and, after. This grammar generates a finite set of unambiguous commands which can be decoded if the meaning of the morphemes are well understood. The experiments on this dataset by the authors indicates that seq2seq models fails to extract the *systematic* rules from the grammar which are required for generalization to test data which doesn't come from the same distribution as train data but follows the same underlying rules. Seq2seq models generalize well (and in a data efficient manner) on novel samples from the same distribution on which they have been trained, but fail to generalize to samples which exhibit the same systematic rules as the training data but come from a different statistical distribution viz. longer/unseen compositions of commands. These results indicate that seq2seq models lack the algebraic capacity to compose complex expressions from simple morphemes by operating in a 'rule space' and rather resort to pattern recognition mechanisms.

The inability if a vanilla seq2seq model to solve a compositional task was shown by Liška et al. [2018].

1.2 Attention in Humans

Not too sure about this section at the moment. Let it be for now. Attention from a cognitive neuroscience perspective. Attentional blink

1.3 Objectives

Objectives

1.4 Outline

Thesis outline

Chapter 2

Technical Background

This chapter provides a brief technical overview of the models that we will frequently encounter in the rest of the thesis. A fundamental understanding of these systems is essential for appreciating the problems associated with them and the potential solution for overcoming those problems, which will be discussed in the subsequent chapters.

The key concepts of **attention** and **pondering**, whose conceptualization lies in the study of human reasoning are presented in this chapter as well. I elaborate on how these concepts have been the crucial first steps towards making deep neural networks think like human beings, thereby making their decision making process more interpretable. These concepts also serve as the backbone for the first major contribution os this thesis, which is presented in chapter 3.

This chapter concludes with a overview of formal language theory which is a prerequisite for understanding the theory behind creation of a new language (the second major contribution os this thesis) which I present in section **??**.

2.1 RNN - A non linear dynamical system

We frequently encounter data that is temporal in nature. A few obvious examples would be, audio signals, videos signals, time series of a stock price and natural language. While traditional feed-forward neural networks such as a multi-layer perceptron (MLP) [Rosenblatt, 1962] are excellent at non linear curve fitting and classification tasks, it is unclear as to how they will approach the problem of predicting the value of a temporal signal T at time t given the states $T_0, T_1....T_{t-1}$ such that the states over time are not i.i.d. This is owing to the fact that a conventional feed-forward network is acyclic and thus doesn't have any feedback loop rendering it memoryless. Human beings arguably solve such problems by compressing and storing the previous states in a 'working ' memory,[Miller, 1956] 'chunking ' it [Neath and Surprenant, 2013] [Craik et al., 2000] and predicting the state T_t.

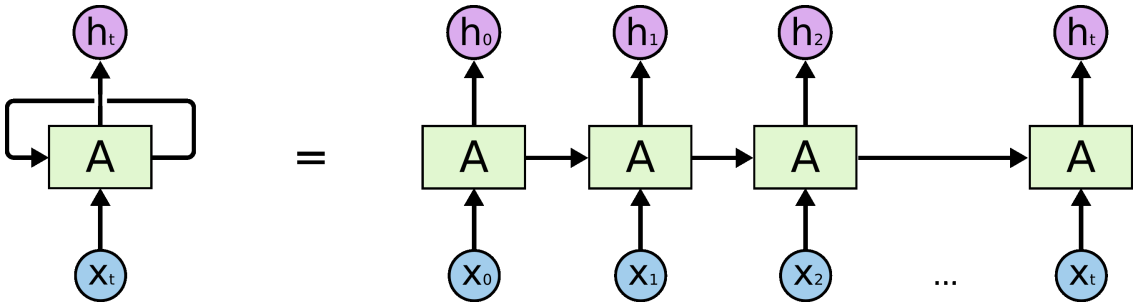


Figure 2.1: Schematic of a RNN Olah [2015]

A Recurrent neural network (RNN) [Hopfield, 1982][Elman, 1990] overcomes this restriction by having feedback loops which allow information to be carried from the current time step to the next. While the notion of implementing memory via feedback loops might seem daunting at first the architecture is refreshingly very simple. In a process known as unrolling a RNN can be seen as a sequence of MLPs (at different time steps) stacked together. More specifically, RNN maintains a memory across time-steps by projecting the information at any given time step t onto a hidden(latent) state through parameters θ which are shared across different time-steps. Figure 2.1 shows a rolled and unrolled recurrent and the equations of an RNN are as follows: and network.

$$c_t = tanh(Ux_t + \theta c_{t-1}), \tag{2.1}$$

$$y_t = sotmax(Vc_t). \tag{2.2}$$

2.2 BPTT and Vanishing Gradients

The conventional method of training a feedforward neural network is a two step process. The first step called the **forward pass** when values fed at input layer pass through the hidden layers, are acted upon by (linear or non-linear) activations and come out at the output layer. In the second step called the **backward pass** the error computed at the output layer (from the target output) flow backward through the network i.e. by applying the chain rule of differentiation, the error gradient at output layer, is computed with respect to all possible paths right upto input layer and then aggregated. This method of optimization in neural networks is called **backpropagation** [Rumelhart et al., 1986].

The optimization step i.e. the backward pass over the network weights is not just with regards to the parameters at the final time step but over all the time steps across which the weights (parameters) are shared. This is known as Back Propagation Through Time (BPTT) [Werbos, 1990] and it gives rise to the problem of vanishing (or exploding) gradients in vanilla RNNs. This concept is better elaborated upon through equation in the following section.

2.2.1 BPTT for Vanilla RNN

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = -\sum_{t} (y_t \log \hat{y}_t) \tag{2.3}$$

The weight W_{oh} is shared across all time steps. $\therefore$ adding the derivatives across the sequence:

$$\frac{\partial \mathcal{L}}{\partial W_{oh}} = \sum_{t} \frac{\partial \mathcal{L}}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W_{oh}} \tag{2.4}$$

For time-step t $\rightarrow$ t+1:

$$\frac{\partial \mathcal{L}(t+1)}{\partial W_{hh}} = \frac{\partial \mathcal{L}(t+1)}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial W_{hh}} \tag{2.5}$$

W_{hh} is shared across time steps, we take the contribution from previous time steps as well, for calculating the gradient at time $t+1$. Summing over the sequence we get:

$$\frac{\partial \mathcal{L}(t+1)}{\partial W_{hh}} = \sum_{\tau=1}^{t+1} \frac{\partial \mathcal{L}(t+1)}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_{\tau}} \frac{\partial \mathbf{h}_{\tau}}{\partial W_{hh}} \tag{2.6}$$

Summing over the whole sequence we get:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t} \sum_{\tau=1}^{t+1} \frac{\partial \mathcal{L}(t+1)}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_{\tau}} \frac{\partial \mathbf{h}_{\tau}}{\partial W_{hh}} \tag{2.7}$$

From equation 2.7 it is clear than the gradient of a RNN can be expressed as a recursive product of $\frac{\partial h_t}{\partial h_{t-1}}$. **If this derivative is $\ll 1$ or $\gg 1$, the gradient would vanish or explode respectively** when the network is trained over longer time-steps. In the former case error back-propagated would be too low to change the weights (the training would freeze) while in the latter it would never converge.

Additionally revisiting equation 2.1 and rewriting it as follows:

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}), \tag{2.8}$$

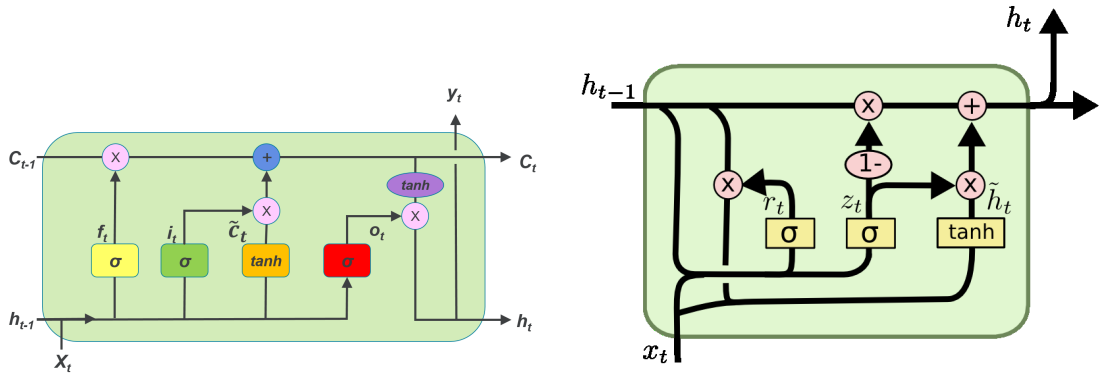
it is not difficult to see that in the absence of an external input $x^{(t)}$ an RNN induces a a dynamical system. The RNN therefore can be viewed as a dynamical system with the input as an external force (map) that drives it. A dynamical system can posses a set of points which are invariant under any map. These points are called the **attractor states** of a dynamical system. These set of points can contain a single point (fixed attractor), a finite set of points (periodic attractor) or an infinite set of points (strange attractor). The type of attractor in a RNN unit depends on the initialization of the weight matrix for the hidden state [Bengio et al., 1993]. Now under the application of map (input) if $\frac{\partial h_t}{\partial h_k}$ (for a large $t-k$ i.e. long term dependency), go to zero one can argue that the state h_t is in the basin of one of the attractor states. This implies that in order to avoid the

vanishing gradient problem a RNN cell must stay close to the 'boundaries between basins of attraction ' [Pascanu et al., 2012].

owing to this problem of vanishing (or exploding) gradients a vanilla RNN can't keep track of long term dependencies which is arguably critical for tasks such as speech synthesis, music composition or neural machine translation. The architectural modifications which solved the vanishing gradient problem and are the current de-facto RNN cell(s) are presented in the next section.

2.3 Gated RNNs

The problem of vanishing (or exploding) gradients makes a vanilla RNN unsuitable for long term dependency modeling. However if for instance the RNN was to compute an identity function then the gradient computation wouldn't vanish or explode since the Jacobian is simply an identity matrix. Now while an identity initialization of recurrent weights by itself isn't very interesting it brings us to the underlying principle behind gated architectures i.e. the mapping from memory state at one time step to the next is close to identity function.



(a) LSTM[Zheng et al., 2017] **(b)** GRU [Olah, 2015]

Figure 2.2: Gated RNNs

2.3.1 LSTM

Long Short Term Memory (LSTM) introduced by Hochreiter and Schmidhuber [1997] is one of the two most widely used gated RNN architectures in use today. The fact that it has survived all the path-breaking innovations in the field of deep learning for over twenty years to still be the state of the art in sequence modeling speaks volumes about the architecture's ingenuity and strong fundamentals.

The fundamental principle behind the working of a LSTM is, alter the memory vector only selectively between time steps such that the memory state is preserved over long

distances. The architecture is explained as follows:

$$i = \sigma(x_t U^{(i)}, m_{t-1} W^{(i)}) \tag{2.9}$$

$$f = \sigma(x_t U^{(f)}, m_{t-1} W^{(f)}) \tag{2.10}$$

$$o = \sigma(x_t U^{(o)}, m_{t-1} W^{(o)}) \tag{2.11}$$

$$\widetilde{c_t} = \tanh(x_t U^{(g)}, m_{t-1} W^{(g)}) \tag{2.12}$$

$$c_t = c_{t-1} \odot f + \widetilde{c_t} \odot i \tag{2.13}$$

$$m_t = tanh(c_t) \odot o \tag{2.14}$$

- **input gate $i^{(t)}$:** The input computes a new cell state based on the current input and the previous hidden state and decides how much of this information to "let through" via. a sigmoid activation.
- **forget gate $f^{(t)}$:** The forget gate decides what to remember and what to forget for the new memory based on the current input and the previous hidden state.The sigmoid activation acts like a switch where 1 implies remember everything while 0 implies forget everything.
- **output gate $o^{(t)}$:** The output gate then determines determines (via a sigmoid activation) the amount of this internal memory to be exposed to the top layers (and subsequent timesteps) of the network.
- **The input modulation $g^{(t)}$** computed based on the present input and the previous hidden state (which is exposed to the output) yields candidate memory for the cell state via a tanh layer. The hadamard products of input gate and candidate memories is added to the hadamard product of forget gate and previous cell state to yield the new cell state.
- **hidden state $m^{(t)}$:** A hadamard product of the hyperbolic tangent of current cell state and output gate yields the current hidden state.

2.3.2 GRU

The Gated Recurrent Unit (GRU) introduced by Cho et al. [2014a] are a new type of gated RNN architecture whose details are as follows:

$$z_t = \sigma(x_t U^{(z)}, m_{t-1} W^{(z)}) \tag{2.15}$$

$$r_t = \sigma(x_t U^{(r)}, m_{t-1} W^{(r)}) \tag{2.16}$$

$$\widetilde{m_t} = \tanh(x_t U^{(g)}, r_t \odot m_{t-1} W^{(g)}) \tag{2.17}$$

$$m_t = (1 - z_t) m_{t-1} + z_t \widetilde{m_t} \tag{2.18}$$

- **update gate** $z_{(t)}$**:** The update gate is the filter which decides how much of the activations/memory to be update at any given time step.
- **reset gate** $r_{(t)}$**:** The reset gate is similar to the forget gate in a LSTM. When its value is close to zero it allows the cell to forget the previously computed state.
- **The input modulation** $g_{(t)}$ just as in the case of the LSTM serves the purpose of yielding candidate memories for the new cell state.
- **hidden state** $m_{(t)}$**:** The current hidden state is a weighted average of the previous hidden state and the candidate hidden state weighted by (1 - update gate) and the (update gate) respectively.

While there are a lot of similarities between a GRU and a LSTM the most striking difference is the lack of an output gate in a GRU. Unlike a LSTM a GRU doesn't control how much of its internal memory to expose to the rest of the units in the network. The GRU therefore has fewer parameters due to the lack of an output gate and is computationally less intensive in comparison to a LSTM.

2.4 Seq2Seq Models

Sequence-to-sequence (seq2seq) models introduced by Sutskever et al. [2014], Cho et al. [2014a] are a class of probabilistic generative models that let us learn the mapping from a variable length input to a variable length output. While initially conceived for machine translation, they have been applied successfully to the tasks of speech recognition, question answering and text summarization [Vinyals et al., 2015] [Anderson et al., 2018] [Lu et al., 2017].

Neural networks have been shown to be excellent at learning rich representation from data without the need for extensive feature engineering [Hinton and Salakhutdinov, 2006]. RNNs are especially adept at learning features and long term dependencies in sequential data (section 2.1). The simple yet effective idea behind a seq2seq model is learning a fixed size (latent) representation of a variable length input and then generating a variable length output by conditioning it on this latent representation and the previous portion of the output sequence.

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}, v) \tag{2.19}$$

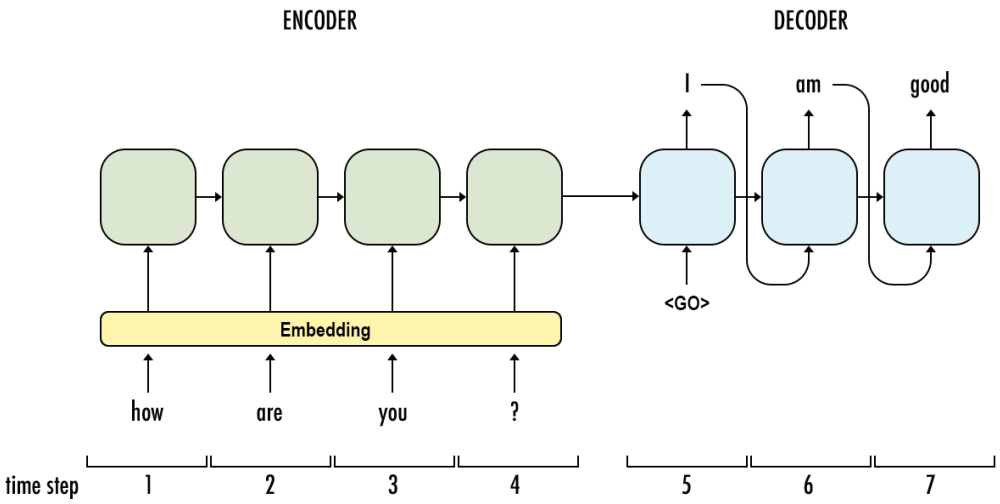


Figure 2.3: Schematic of a Seq2Seq [Chablani, 2017]

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, ..., x^{(1)}) = g(x^{(t)}, h^{(t)}, v) \tag{2.20}$$

It can be seen from equation 2.20 that the decoder is auto-regressive with the long term temporal dependencies captured in its hidden state. The term **v** represents the summary of the entire input state compressed into a fixed length vector (last hidden state of encoder) viz. the **latent space**. This encoder- decoder network is then jointly trained via. cross entropy loss between the target and the predicted sequence.

2.4.1 Se2Seq with Attention

It was shown by Cho et al. [2014b] that the performance of a basic encoder-decoder models as explained in 2.4 is inversely related to the increase in length of the input sentence. Therefore in lines with concepts of the selective attention and attentional blink in human beings, Bahdanau et al. [2014] and later Luong et al. [2015] showed that soft selection from source states where most relevant information can be assumed to be concentrated during a particular translation step in NMT leads to improved performance. In the Bahdanau [Bahdanau et al., 2014] framework of attention, the equations 2.19 and 2.20 are modified as follows:

$$h^{(t)} = f(x^{(t)}, h^{(t-1)}, c^{(t)}), \tag{2.21}$$

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, ..., x^{(1)}) = g(x^{(t)}, h^{(t)}, c^{(t)}). \tag{2.22}$$

Here unlike the traditional seq2seq model the probability of emission of the output at time step t isn't conditioned on a fixed summary representation **v** of the input sequence. Rather it is conditioned on a context vector $\mathbf{c^{(t)}}$ which is distinct at each decoding step. The context vector is calculated using a sequence of encoder outputs $(s^1, s^2, ..., s^N)$ to which

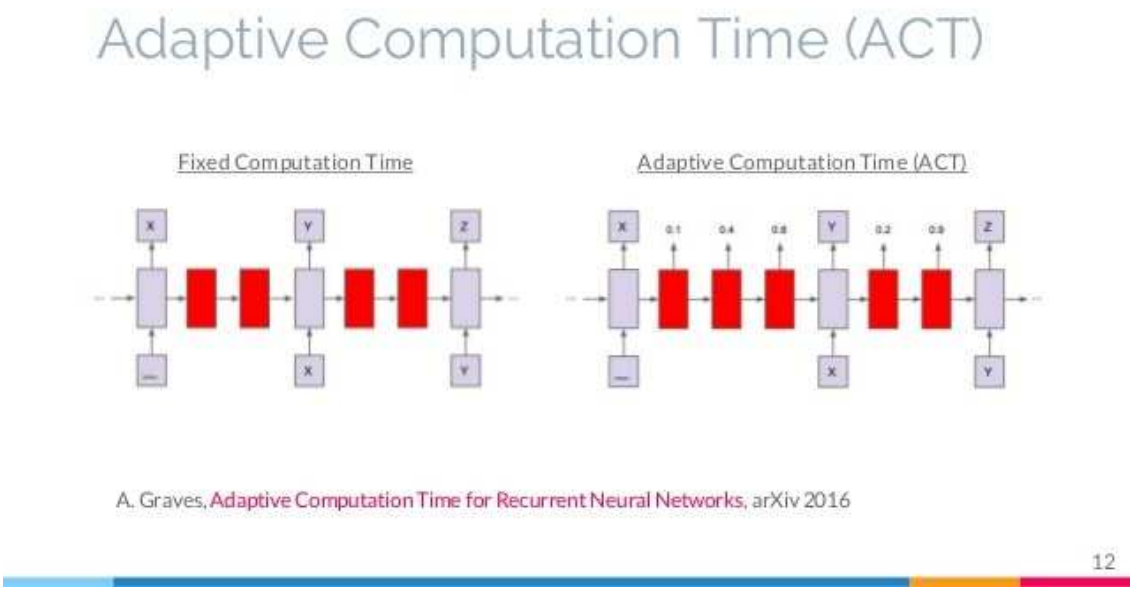


Figure 2.4: Adaptive Computation Time

the the input sequence is mapped such that an encoder output s^i contains a representation of the entire sequence with maximum information pertaining to the i_{th} word in the input sequence. The context vector is then calculated as follows:

$$c^{(t)} = \sum_{j=1}^{N} \alpha_{tj} s^j, \tag{2.23}$$

where:

$$\alpha_{tj} = \text{softmax}(e(h^{(t-1)}, s^j)). \tag{2.24}$$

where e is a alignment/matching/similarity measure between the decoder hidden state $h^{(t-1)}$ i.e. just before the emission of output $x^{(t)}$. The alignment function can be a dot product of the two vectors or a feed-forward network that is jointly trained with the encoder-decoder model. The α_{tj} is an attention vector that weighs the encoder outputs at a given decoding step. Vaswani et al. [2017] view the entire process of attention and context generation as a series of functions applied to the tupe (query, key, value), with the first step being an **attentive read** step where a scalar matching score between the query and key $(h^{(t-1)}, s^j)$ is calculated followed by computation of attention weights α_{tj}. Weighted averaged of the 'values' using the attention weights is then done in the **aggregation** step.

2.5 Pondering

The task of positioning a problem and solving belong to different classes of time complexity with the latter requiring more time than the former. Graves [2016] argued that for a given RNN unit it is reasonable to allow for variable computation time for each input in a

sequence since some parts of the input might be inherently more complex than the others and thereby require more computational steps. A very good example of this would be *spaces between words and ends of sequences.*

Human beings overcome similar obstacles by allocating more time to a difficult problem as compared to a simpler problem. Therefore a naive solution would be to allow a RNN unit to have a large number of hidden state transitions (without penalty of amount of computations performed) before emitting an output on a given input. The network would therefore learn to allocate as much time as possible to minimize its error thereby making it extremely inefficient. Graves [2016] proposed the concept of **adaptive computation time** to have a trade-off between accuracy and computational efficacy in order to determine the minimum number of state transitions required to solve a problem [1].

Adaptive **C**omputation **T**ime (ACT) achieves the above outlined goals by making two simple modifications to a conventional RNN cell, which are presented as follows:

Sigmoidal Halting Unit If we revisit the equations of a vanilla RNN from section 2.8, they can be summarized as:

$$\begin{aligned} h_t &= f(Ux_t + Wc_{t-1}), \\ y_t &= g(Vc_t). \end{aligned} \tag{2.25}$$

ACT now allows for *variable state transitions* $(c_t^1, c_t^2, ..., c_t^{N(t)})$ and by extension an *intermediate output sequence* $(y_t^1, y_t^2, ..., y_t^{N(t)})$ at any given input step t as follows:

$$\begin{aligned} c_t^n &= \begin{cases} f(Ux_t^1 + Wc_{t-1}) \text{ if } n = 1 \\ f(Ux_t^n + Wc_t^{n-1}) \text{ if } n \neq 1 \end{cases}, \\ y_t^n &= g(Vc_t^n). \end{aligned} \tag{2.26}$$

A sigmoidal halting unit (with its associated weight matrix S) is now added to the network in order to yield a halting probability p_t^n at each state transition as follows:

$$\begin{aligned} h_t^n &= \sigma(Sc_t^n), \\ p_t^n &= \begin{cases} R(t) \text{ if } n = N(t) \\ h_t^n \text{ if } n \neq N(t) \end{cases}, \end{aligned} \tag{2.27}$$

where:

$$\begin{aligned} N(t) &= min\{m : \sum_{n=1}^{m} h_t^n \geq 1 - \epsilon\}, \\ R(t) &= 1 - \sum_{n=1}^{N(t)-1} h_t^n, \end{aligned} \tag{2.28}$$

and ϵ is a small constant.

Each (n^{th}) hidden state and output transition at input state t are now weighted by the corresponding halting probability p_t^n and summed over all the updates $N(t)$ to yield the

[1] Theoretically this is akin to halting on a given problem or finding the Kolmogorov Complexity of the data, both of which are unsolvable

final hidden state c_t and output y_t at a given input step. Figure 2.4 outlines the difference between a standard RNN cell and an ACT RNN cell by showing variable state transitions for input x and y respectively with the corresponding probability associated with each update step. It can be noted that $\sum_n = 1^N(t)p_t^n = 1$ and $0 \leq p_t^n \leq 1 \ \forall n$, and therefore it constitutes a valid probability distribution.

Ponder Cost If we don't put any penalty on the number of state transitions then the network would become computationally inefficient and would '**ponder**' for long times even on simple inputs in order to minimize its error. Therefore in order to limit the variable state transitions ACT adds a ponder cost $\mathcal{P}(x)$ to the total loss of the network as follows: given an input of length T the ponder cost at each time step t is defined as:

$$\rho_t = N(t)/+ \ R(t) \tag{2.29}$$

$$\begin{aligned} \mathcal{P}(x) &= \sum_{t=1}^{T} \rho_t, \\ \widetilde{\mathcal{L}}(x,y) &= \mathcal{L}(x,y) + \tau\mathcal{P}(x), \end{aligned} \tag{2.30}$$

where τ is a penalty term hyperparameter (that needs to be tuned) for the ponder loss.

2.6 Formal Language Theory

The field of formal language theory (FLT) concerns itself with the syntactic structure of a formal language (=set of strings) without much emphasis on the semantics. More precisely a formal language L is a sequence of strings with the constituent units/words/morphemes taken from a finite vocabulary Σ. It is more apt to define the concept of a formal grammar before proceeding further. A formal grammar G is a quadruple $\langle \Sigma, NT, S, R \rangle$ where Σ is a finite vocabulary as previously defined, NT is a finite set of non-terminals, S the start symbol and R the finite set of valid production rules. A production rule can be expressed as $\alpha \rightarrow \beta$ and can be understood as a substitution of α with β with α, β coming from the following sets for a **valid** production rule:

$$\alpha \in (\Sigma \cup NT)^* NT (\Sigma \cup NT)^* \qquad \beta \in (\Sigma \cup NT)^*[2]. \tag{2.31}$$

From equation 2.31 it is easy to see that the left hand side of the production rule can never be null (ϵ) and must contain at-least one non-terminal. Now a formal language $L(G)$ can be defined as the set of all strings *generated* by grammar G such that the string consists of morphemes only from Σ, and has been generated by a finite set of rule (R) application after starting from S. The *decidability* of a grammar, is the verification -by a Turing machine or another similar computational construct e.g. a finite state automaton (FSA)- of whether a given string has been generated by that grammar or not (the *membership* problem). A grammar is decidable if the membership problem can be solved for all given strings.

[2]The '*' denotes Kleene Closure and for a set of symbols say X, X^* denotes a set of all strings that can be generated using symbols from X, including the empty string ϵ

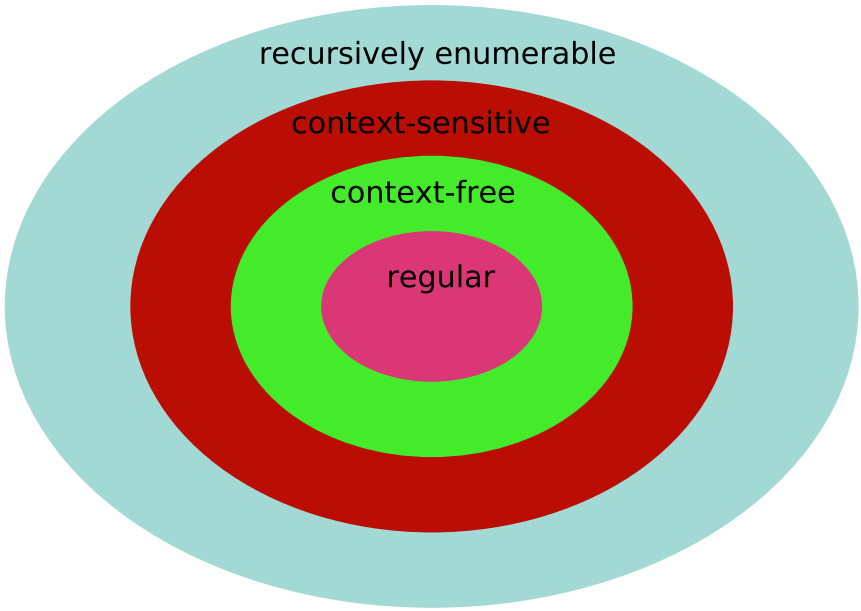


Figure 2.5: Chomsky Hierarchy

2.6.1 Chomsky Hierarchy

Chomsky [1956] introduced a nested hierarchy for different formal grammars of the form $C_1 \subsetneq C_2 \subsetneq C_3 \subsetneq C_4$ as shown in figure 2.5. The different classes of grammar are progressively strict subsets of the class just above them in the hierarchy. These classes are not just distinguished by their rules or the languages they generate but also on the computational construct needed to decide the language generated by this grammar. We now take a closer look at the classes in this hierarchy. Please note that for each of these classes the grammar definition is $\mathrm{G} = \langle \Sigma, NT, S, R \rangle$.

Recursively Enumerable. Any grammar characterized by no constraints on the production rules $\alpha \rightarrow \beta$. Therefore any valid grammar is recursively enumerable. The language generated by this grammar is called recursively enumerable language (REL) and is accepted by a Turning Machine.

Context-Sensitive. In this grammar the left hand side of the production rule i.e. α has the same definition as above (equation 2.31) but an additional constraint of the form $|\alpha| \leq |\beta|$ is now imposed on the grammar. This is turn leads to $\beta \in (\Sigma \cup NT)^+$, i.e. the right hand side of the production rule is now under Kleene Plus closure [3]. The non production of ϵ in context-sensitive grammars poses a problem to the hierarchy because the production of null symbol isn't restricted in its subclasses. While keeping the hierarchy as it is Chomsky [1963] resolved this paradox by defining noncontracting grammmar which is *weakly equivalent*(generates same set of string) to the context sensitive grammar. Noncontracting grammars allow the $S \rightarrow \epsilon$ production. Context sensitive grammars generate context sensitive languages which are accepted by a linear bounded turing machine. While in principle this grammar is decidable, the problem is PSPACE hard and can be so complex, that it is practically intractable [Jäger and Rogers, 2012].

Context-Free. Such a grammar is described by production rules of the form $A \rightarrow \alpha$ where $A \in NT$ and $\alpha \in (\Sigma \cup NT)^*$, such that $|A| = 1$. Context free grammar lead to context free languages (CFL) which are hierarchical in structure, albeit it is possible that same CFL can be described by different context free grammars, leading to different hierarchical syntactic structures of the language. A CFG is decidable in cubic time of length of string by push down FSA. A psuh down automaton employs a running stack of symbols to decide its next transition. The stack can also be manipulated as a side effect of the state transition.

Regular. Characterized by production rules of the form $A \rightarrow \alpha or A \rightarrow \alpha B$ where $\alpha \in \Sigma^*$ and $(A, B) \in NT$. . The non terminal in production can therefore be viewed as the next state(s) of a finite state automaton (FSA) while the terminals are the emissions. Regular grammars are decidable in linear time of length of string by a FSA.

[3] $(\Sigma \cup NT)^+ = (\Sigma \cup NT)^* - \epsilon$

2.6.2 Subregular Hierarchy

The simplest class of languages encountered in section 2.6.1 were regular languages that can be described using a FSA. Jäger and Rogers [2012] however argue that the precursor to human language faculty would require lower cognitive capabilities and it stands to reason that even simpler structures can exist in the 'Regular' domain. They therefore introduced the concept of subregular languages. If a language can be described by a mechanism even simpler than the FSA then it is a subregular language introduced by While far from the expressive capabilities of regular languages which in turn are the least expressive class in the Chomsky hierarchy, subregular languages provide an excellent benchmark to test basic concept learning and pattern recognition ability of any intelligent system.

Strictly local languages. We start with a string w and we are given a lookup table of k-adjacent characters known as *k-factors*, drawn from a particular language. The lookup table therefore serves the role of the language description. A language is k-local, if every k-factor seen by a *scanner* with a windows of size k sliding over the string w, is included in the aforementioned lookup-table. A SL_k language description, is just the set of k-factors prefixed and suffixed by a start and end symbol, say #. E.g. $SL_2 = \{\#A, AB, BA, B\#\}$

Locally k-testable languages. Instead of sliding a scanner over k-factors we consider all the k-factors to be atomic and build k-expression out of them using *propositional logic*. This language description is locally k-testable. As in the case of strictly local languages, scanner won window size K slides over the string and records for every k-factor in vocabulary it's occurrence or nonoccurence in the string. The output of this scanner is then fed to a boolean network which verifies the k-expressions. E.g. 2-expression $(\neg\#B) \wedge A$, is a set of strings that doesn't start with B and consists of atleast one A.

Remarks on Chomsky Hierarchy: It is easy to see by looking at the production rules of all the grammars in Chomsky Hierarchy that, the languages generated by them are compositional in nature. This allows us to create artificial languages such as SCAN [Lake and Baroni, 2017] which are context-free, in order to test compositionality in deep neural networks. That said, it is worth noticing that while the grammars in Chomsky Hierarchy are finite, the languages they generate can be infinite. For an infinite language one can argue that a model that can infer the grammar from the given strings is the one that will generalize well to unseen strings. However if the language itself only contains a finite number of strings then albeit compositional, it can be solved also by pure memorization.

Chapter 3

Attentive Guidance

In this chapter I review some pitfalls of RNNs and seq2seq models in their current state. I go on to show how they fail at compositional learning. Finally I present a novel idea for overcoming this. I also present two datasets which serve as our benchmarks. A comprehensive view of these problems paves the way towards distinguishing human intelligence from machine intelligence.

Liška et al. [2018] using lookup tables (section 3.2) as the testbed tried to asses the ability of a RNN network (more specifically a 60 unit LSTM followed by a 10 unit sigmoid layer) to search for compositional solution to the lookup table task. Lookup tables exhibit functional nesting and therefore a model that can find a compositional solution from the search space of all possible solutions is more likely to zero shot generalize to novel compositions. The authors established that by having additional supervision on the weights of hidden state transitions, theoretically a finite-state automata (FSA) can be induced such that the recurrent layers encode the states of this automaton. This FSA can in principle solve the lookup table task upto a finite number of compositions. They further showed that this theoretical setup can achieve zero state generalization on unseen inputs on known compositions i.e. *heldout inputs* (section 3.2.1).

However when trained purely on input/output mappings without this additional supervision, the authors noted that only a small percentage of networks could converge to a compositional solution (models capable of zero shot generalizing to unseen inputs on known compositions). Additional these small percentage of models also only showed a weak form of composition wherein if for instance t1, t2 are atomic tables, then a composition task t1 t2 is indexed to prompt t1t2, instead of solving in a nested fashion viz. t1(t2(.)). (Get feedback here, see if this is clear)

Lake et al. [2015] introduced Hierarchical Bayesian Program Learning (HBPL) to learn complex characters (concepts) from few samples by representing them as probabilistic programs which are built compositionally via. bayesian sampling from simpler primitives, subparts, parts and the relations between them, respectively. This approach led to human level generalization on the **omniglot** dataset [Lake et al., 2015] which is a dataset containing 1623 characters (concepts) with 20 samples each. Omniglot is therefore not

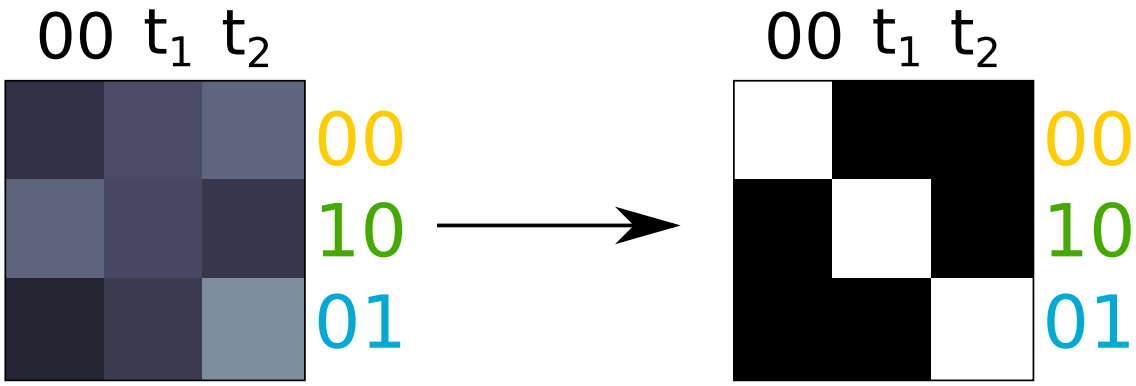


Figure 3.1: Diffused vs Hard Attention

sample intensive and hence ideally suited to test one-shot generalization capabilities of a model. This work served as the major motivation for learning nested functions such as *lookup tables* of the form t1(t2(.), by learning the compositions from simpler primitives i.e. atomic tables and then stacking them hierarchically. The procedure for learning the **trace** of the above-mentioned is described subsequently.

3.1 Attentive Guidance

Attention (section 2.4.1) based seq2seq models produce a 'soft' alignment between the source (latent representation of the input) and the target. Furthermore seq2seq models require thousands of samples to learn this soft alignment. However in light of the aforementioned arguments presented in favor of concentrating on primitives to construct a complex 'composition' I propose the concept of **A**ttentive **G**uidance (AG). AG argues that the decoder having perfect access to the encoder state(s) containing maximum information pertaining to that decoding step, would leave to improved target sequence accuracy.

Revisiting the query-key-value pair view of attention described in section 2.4.1, AG tries to improve the scalar matching score between the query and the keys during the attentive read step. Since the *keys* can be thought of as the memory addresses to the *values* which are needed at a given decoding step, AG tries to ensure a more efficient information retrieval. Similar to Lake et al. [2015] AG induces the trace of a program (albeit not probabilistic) needed to solve a complex composition by solving its subparts in a sequential and hierarchical fashion. This in turn forces the model search for a compositional solution from the space of all possible solutions. AG eventually results in a 'hard' alignment between the source and target as seen in figure 3.1

AG is implemented via an extra loss term added to the final model loss. As shown in figure 3.2, at each step of decoding, the cross entropy loss between calculated attention vector $\hat{a}_{t,i}$ and the ideal attention vector $a_{t,i}$, are added to the model loss. The final loss for an output sequence of length T and an input sequence of length N is therefore

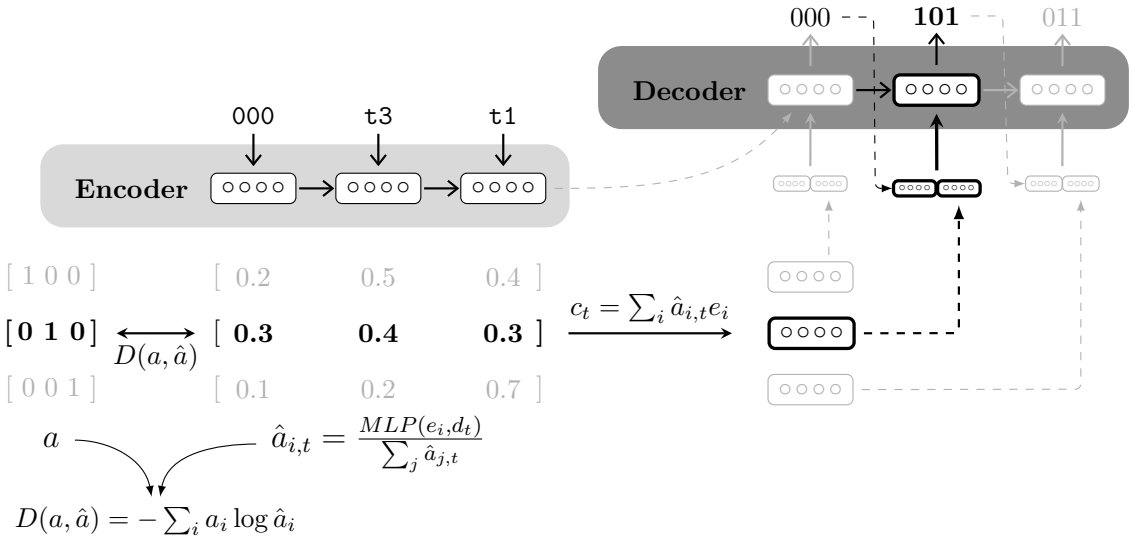


Figure 3.2: Attentive Guidance and calculation of AG loss [Hupkes et al., 2018]

expressed as:

$$\widetilde{\mathcal{L}}(x, y) = \mathcal{L}(x, y) + \sum_{t=1}^{T} \sum_{i=1}^{N} -a_{i,t} \log \hat{a}_{i,t} \tag{3.1}$$

3.1.1 AG and Pondering

How pondering is mocked using AG. implicit in case of LT and explicit in Micro Tasks.

3.2 Lookup Tables

The lookup tables task was introduced by Liška et al. [2018] within the CommAI domain [Baroni et al., 2017] as an initial benchmark to test the generalization capability of a compositional learner. The data consists of atomic tables which bijectively map three bit inputs to three bit outputs. The compositional task can be understood in the form of a nested function $f(g(x))$ with f and g representing distinct atomic tables. To clarify the task with an example, given $t1$ and $t2$ refer to the first two atomic tables respectively; also given that $t1(001) = 100$ and $t2(100) = 010$. Then a compositional task is presented to a learner as $001t1t2 = 010$. Since the i/o strings are three bit, there can be a maximum of 8 input/output strings.

As is very clear from the task description that since the table prompts t_i don't have any semantic meaning in itself, the meaning of each individual table prompt can be correlated only with the bijective mapping it provides. Secondly the dataset is in agreement with the systematic compositionality definition that we have outlined in section 1.1.1. Lastly one can argue that even a human learner might come up with an approach that is different

than solving each function individually and sequentially in a given nested composition, but such an approach will not be able to scale with the depth of nesting.

3.2.1 Data Structure

We generated eight distinct atomic tables $t1.....t8$ and work with compositions of length two, i.e. $t_i - t_j$. This leads to a possible 64 compositions. Since we want our model to not simply memorize the compositions but rather to land on a compositional solution, we propose to use the compositions only from tables $t1 - t6$ for the training set. However since the model needs to know the mapping produced by tables $t7, t8$ in order to solve their compositions we expose the model to the atomic tables $t7, t8$ in the training set. The details of all the data splits and some dataset statistics are presented below. Examples from each split and the size of each split are presented in table 3.1

1. **train** - The training set consists of the 8 atomic tables on all 8 possible inputs. The total compositions of tables $t1 - t6 = 36$. Out of those 36 compositions we take out 8 compositions randomly. For the remaining 28 compositions we take out 2 inputs such that the training set remains balance w.r.t. the compositions as well as the output strings. (Details of this sudoku algorithm in the appendix)

2. **heldout inputs** - The 2 inputs taken out from the 28 compositions in training constitute this test set. However of the 56 data points, 16 are taken out to form a validation set. In creating this split we ensure that the splits i.e. *heldout inputs* and *validation* have a uniform distribution in terms of output strings at the expense of the uniformity in the compositions.

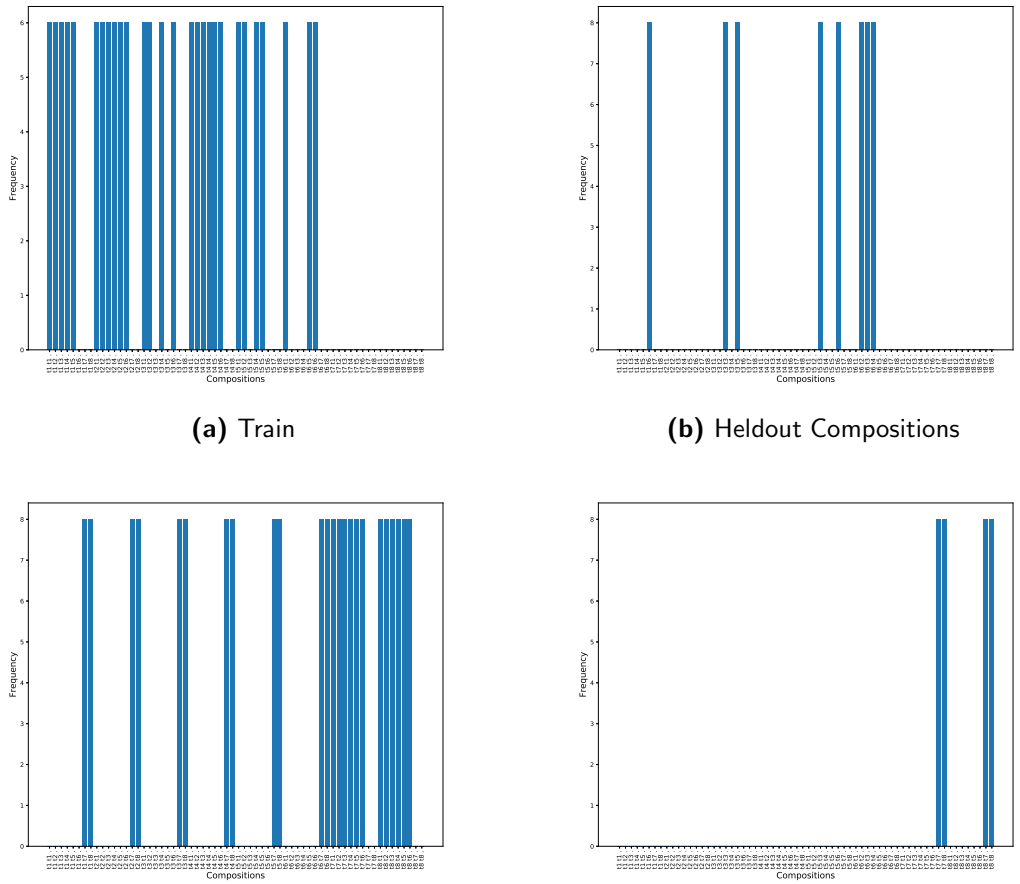
3. **heldout compositions** - This set is formed by the 8 compositions that were taken out of the initial 36 compositions. These 8 compositions are exposed to all 8 possible input strings.

4. **heldout tables** - This test is a hybrid of the tables which are seen in compositions during training i.e. $t1 - t6$ and those which are seen just atomically during training i.e. $t7 - t8$. There are total of 24 compositions in this split which are exposed to all 8 inputs.

5. **new compositions** - This split consists of compositions of $t7 - t8$ and therefore a total of 4 compositions on 8 inputs.

	Example	Size
train	t1 t2 011	232
heldout inputs	t1 t2 001	40
heldout compositions	t1 t3 110	64
heldout tables	t1 t8 111	192
new compositions	t7 t8 101	32

Table 3.1: Lookup Table Splits



(a) Train

(b) Heldout Compositions

(c) Heldout Tables

(d) New Compositions

Figure 3.3: Data distribution of train and test sets

In accordance with the data split described above we present the distribution of all compositions in the train and various test sets. It can be seen that the test sets 'heldout_tables' and 'new_compositions ' are the most difficult and require zero-shot generalisation owing to their significantly different distribution as compared to 'train '.

3.3 Symbol Rewriting

Introduced by Weber et al. [2018] the symbol rewriting dataset is essentially a probabilistic context free grammar (PCFG). It consists of a rewriting a set of input symbols to a set of output symbols based on this grammar. Before proceeding further with the task description,I'll elaborate on PCGFs briefly.

PCFGs are the stochastic version of CFGs (Context free grammars) that we encountered in section 2.6.1. The addition of this stochasticity was motivated by the non-uniformity of words in natural language. Assigning probabilities to production rules, lead

to a grammar more in line with the Zipfian distribution of words in natural language. A PCFG consists of:

1. A CFG $\mathcal{G} = \langle \Sigma, N, S, R \rangle$ where the symbols have the same meaning as defined in section 2.6.
2. A probability parameter $p(a \to b) \mid \sum_{a \to b \mid a \in N} p(a \to b) = 1$
3. $\therefore$ the probabilities associated with all the expansion rules of a given non-terminal should sum up to 1.

The parse tree shown in figure **??** illustrates the production rule for one input symbol. The grammar consists of 41 such symbols each following similar production rules. Weber et al. [2018] showed using this dataset while seq2seq models are powerful enough to learn some structure from this data and generalize on a test set which was drawn from the same distribution as the training set. They posit that given the simplicity of the grammar it should be possible to generalize to test sets (with some hyperparameter tuning) that are atypical of the training distribution while still conforming to the underlying grammar. They however show that this indeed **is not** the case.

3.3.1 Data Structure

The data-splits as furnished [Weber et al., 2018] consists of a training data and different test cases which are non-exhaustive and created by sampling randomly from all possible i/o pairs as described by the PCFG. The different test sets are created to ascertain if the seq2seq models actually understand the underlying grammar from the training data or simply memorize some spurious structure from the training distribution. For hyperparameter tuning a validation set which is an amalgamation of random samples from all the different test sets, is used. The details of the different data splits are presented below. Examples from each split and the size of each split are presented in table 3.2

1. **train** consists of 100000 pairs of input output symbols with input string length ranging between $\langle$ 5 and 10 $\rangle$. Output string length is therefore between $\langle$ 15 and 30 $\rangle$. A crucial feature of this set is that no symbol is repeated in a given input string.
2. **standard test** consists of samples drawn from the same distribution as the training set.
3. **repeat test** includes input strings where repetition of symbols is allowed.
4. **short test** includes input strings which are shorter in length as compared to the input strings in the training data. The input string length ranges between $\langle$ 1 and 4 $\rangle$.
5. **long test** consists of input sequences of lengths in the range $\langle$ 11 and 15 $\rangle$.

	Example	Size
train	HS E I G DS	100000
standard test	LS KS G E C P T	2000
repeat	I I I I I I MS	2000
short	M I C	2000
long	Y W G Q V I FS GS C JS R B E M KS	2000

Table 3.2: Symbol Rewriting Splits

The datasets *repeat, short and long* come from distributions which are different from the training distribution on which the model has learned the data structure. It is expected of a compositional learner that given the sufficient size of the training data it would be able to infer a pattern which is close to the underlying PCFG and therefore generalize of the test sets which comes from different distributions but have the same underlying structure.

Chapter 4

Experimental Setup

- Set up to solve lookup tables tasks with attentive guidance
- briefly revisit the trace used
- Set up to show how symbol rewriting was solved with attention guidance
- explain the trace used
- How AG in seq2seq directly translates to pondering?
- Why looking only at Seq Accuracy in LT and Symbol rewriting acc in SR.

While in the case of look up tables we can just look at the final target we choose to view the entire sequence because that show compositional solution.

4.1 Lookup Tables

4.1.1 AG Trace

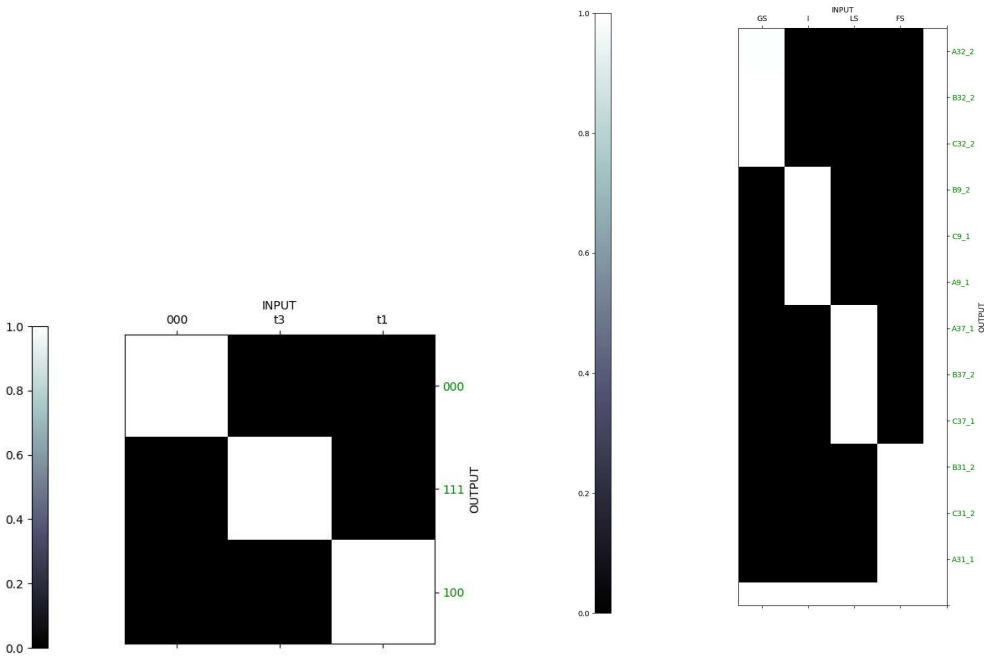
As explained in section 3.1.1 attentive guidance can help in mocking the pondering. The biggest difference from the Pondering in section 2.4 would be that at each pondering step we have an emission, instead of the ponder step being a silent one. The trace for the attentive guidance can be explained as follows and is shown in figure 4.1a:

- The first step is the copy step where the three bit input to the composition is copies as if.
- After this the tables in the composition are applied sequentially to the three bit input preceding them.
- The diagonal trace is mean to capture this sequential and compositional solution of lookup tables. At each step of decoding we force the model to focus on only that input prompt which results in the correct output for that step.

4.1.2 Accuracy

Since the lookup tables can be viewed as nested functions, accuracy of final output of the composition can be an adequate measure of model performance. However since we want to ensure that the model doesn't learn spurious patterns in data to land at an uncompositional solution, we want it to be accurate at each step of the composition. This hierarchical measure of accuracy is a viable test for the compositionality of the network. Therefore in all evaluations *sequence accuracy* is the performance metric of the model.

4.2 Symbol Rewriting



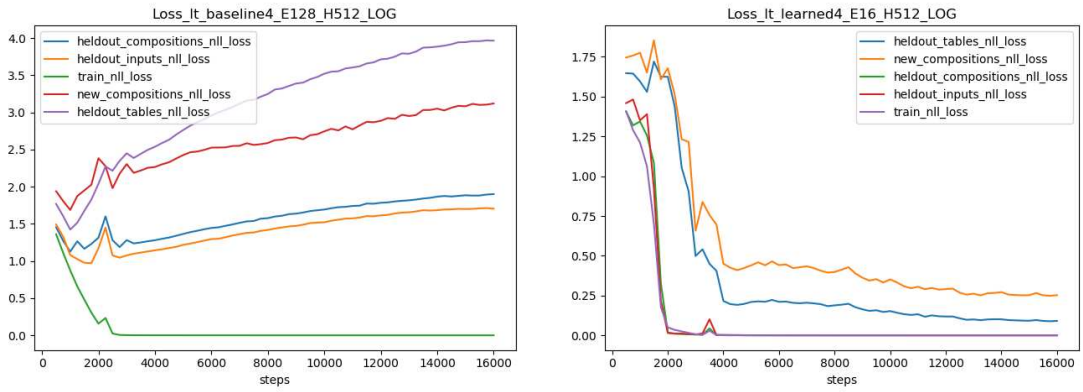
(a) AG Trace for Lookup Tables **(b)** AG Trace for Symbol rewriting

Figure 4.1: Attentive Guidance Trace for Lookup tables and Symbol rewriting tasks

Chapter 5

Results and Discussions

5.1 Lookup Tables



(a) Loss Curves for Baseline

(b) Loss Curves for Learned

Figure 5.1: Learning Curves for Lookup Tables

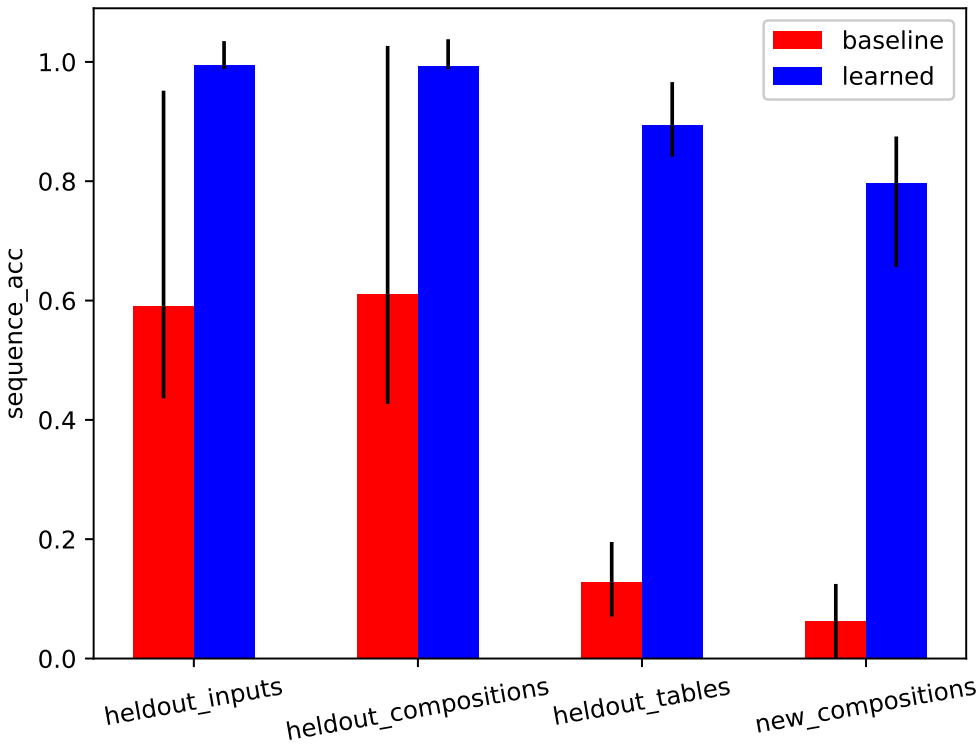


Figure 5.2: Average Sequence Accuracies

Chapter 6

Micro Tasks

Since we have established that AG exhibits compositional learning, and can learn PCFG rules when explicitly asked to do so, we can now test if it can start operating in a rule space. That is learn the rules from the language samples.

Write here about why it makes sense to deal with sub-regular languages. Because it is difficult to test the compositional abilities of a model for CF languages such as SCAN.

CommAI mini tasks [Baroni et al., 2017] are a set of strictly local and locally testable (please refer to section 2.6.2) languages designed to test the generalization capabilities of a compositional learner. Based on the mini tasks we propose a new dataset of sub-regular languages which we call the Micro Tasks. While the CommAI mini task position themselves as strictly local and locally testable tasks of progressive difficulty we add another layer of difficulty to the tasks by setting them up as either a decision problem (verify) or search problem (produce) in the same training domain.

Search vs Decision - It has been shown that for a general language $L \in NP$ search is more difficult than decision Bellare and Goldwasser [1994]. The *verification* task is clearly a decision problem while the *production* task is a search problem. Since Micro tasks fall into the domain of sub-regular languages they can be decided at-most in the linear time of length of string by a finite state automaton (section 2.6.1), they clearly are $\notin NP$. However it still makes sense to analyze whether for a prover (seq2seq network, lstm etc.) is the task of computing the witness for the membership in a $L \notin NP$ more difficult than establishing the existence of such a witness (line copied from the ref above. paraphrase).

The **verify** task input is of the form '⟨ string of k-factors ⟩ verify ⟨ witness ⟩' with the output being a standard classification token 'yes/no'Ṫhe witness string is of length $\geq$ the given k-factor string, and a model must verify if the k-factors exist in the witness. The **production** task consists of an input of the form '⟨ string of k-factors ⟩ produce' with the output being a satisfying assignment (equal to the length of the given string) to the criteria laid out in the input string.

Any given ⟨ k-factor string ⟩ in either of the verify or produce tasks can be one out of the following four types:

- **atomic** - consists of a single word/k-factor/n-gram from the vocabulary and task of a learner is either to check the existence of this k-factor in the witness (verify task) or produce exactly this k-factor as target (production) task. In the verification setting this task is strictly local in its description (section 2.6.2) and can be solved by scanner with an access to the witness/lookup-table.

- **or** - consists of k-factors joined by the boolean operator 'or '. This task is again strictly local in its description albeit more memory intensive than the atomic task because it requires storage of multiple k-factors in memory.

- **and** - consists of k-factors joined by the boolean operator 'and '. Now a learner doesn't simply have to verify the existence of a single k-factor in the witness. Instead for each k-factor it has to maintain a record of it's occurence/non-occurence in the witness, and thus requires store of all k-factors in memory. This task is therefore locally k-testable.

- **not** - consists of k-factors acted upon by both 'and ' as well as 'not ' operators. While the *conjunction* acts upon two k-factors, the *negation* only acts upon one. This task also has a locally k-testable description.

6.1 Data Structure

Out of the 128 ascii characters, only 100 are printable and of those 100, 6 refer to newline, space, tab etc. which are not directly observable and therefore our vocabulary Σ consists of 94 printable ascii characters and two binary output tokens 'yes' and 'no'. The vocabulary is split into two equal disjoint subsets. The train and various test sets constructed from this vocabulary are as follows:

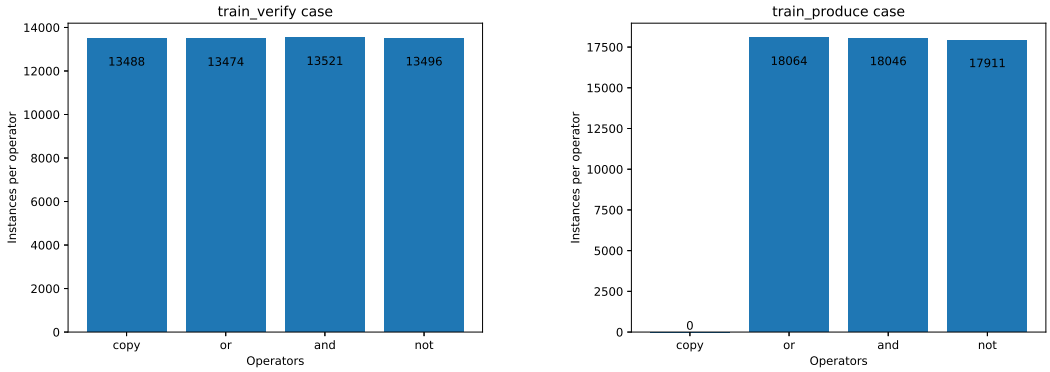
1. **train** consists of 120000 pairs of input output symbols with input composition length (vocabulary symbols excluding operators) ranging between $\langle$ 2 and 4 $\rangle$ (limits inclusive). 10% of this data is used to create the validation set. Any particular input composition is constructed using words from only one of the subsets.

2. **unseen test** consists of 12000 i/o pairs with input composition length ranging between $\langle$ 2 and 4 $\rangle$. The compositions are created by randomly sampling words from both subsets of the vocabulary i.e cross-composition between subsets is allowed

3. **longer test** consists of i/o pairs such that input composition length is between $\langle$ 5 and 7 $\rangle$ (limits inclusive). cross-composition of words from two different subsets isn't allowed.

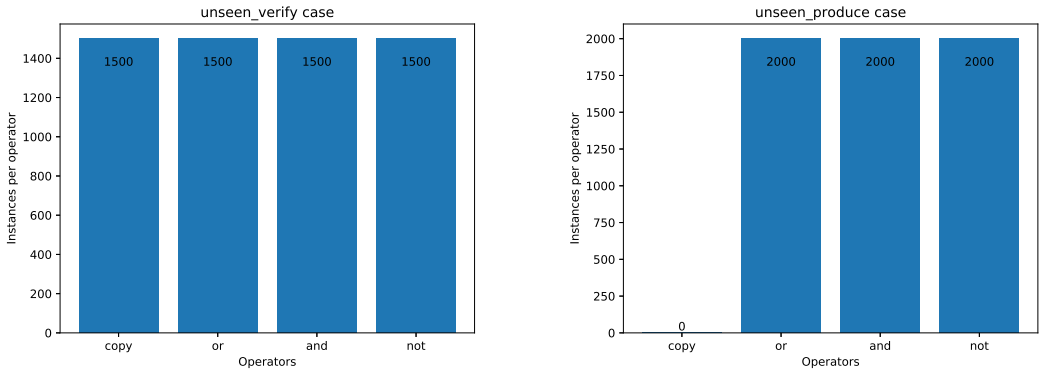
4. **unseen longer test** consist of i/o pairs such that input composition length is between $\langle$ 5 and 7 $\rangle$ and cross-composition is allowed.

The distribution of the boolean operators (i.e. the number of tasks per operator) in the train and unseen test set (the distribution is same for every test set) can be seen in figures 6.1 and 6.2 respectively.



(a) Verification Task **(b)** Production Task

Figure 6.1: Micro Tasks - Training Data



(a) Verification Task **(b)** Production Task

Figure 6.2: Micro Tasks - Unseen Data

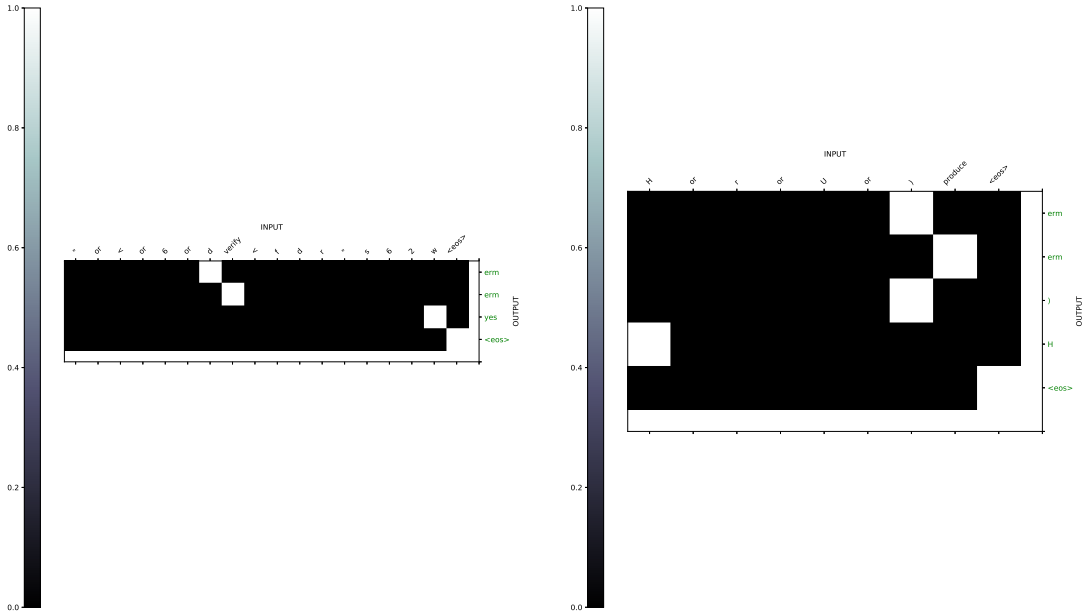
6.2 Experimental Setup

6.2.1 AG Trace

The trace for the hard guidance of micro-tasks is different for verify and produce cases owing to the fact that verify has a single emission (yes/no token) while produce has multiple emissions. Furthermore pondering is explicitly mocked here by augmenting the output with a ponder token. The ponder token helps the network in focusing on the summarization of the entire string and the meta-information i.e. verify/produce, before moving on to the emission of the actual targets. The details of the trace are as follows:

Verify. Here we have a trace of length 3 with the focus at end of string, meta-information and end of witness respectively. While the first two indices of the trace correspond to the ponder tokens in the output, the last index corresponds to the decision token which is the actual target. The trace is visualized in figure 6.3a.

Produce.In the case of produce while the ponder tokens and the trace corresponding to them remain the same as in the case of verify the trace corresponding to the emissions changes. The trace corresponds to the index in input string for an emission in output string. In the case of not, every emission in target which isn't in input is traced in ascending order of index of the words in input which are acted upon by a not operator. The trace is visualized in figure 6.3b



(a) AG Trace for Verification Task **(b)** AG Trace for Production Task

Figure 6.3: Micro Tasks - Attentive Guidance Trace

6.2.2 Micro task Accuracy

Similar to the symbol rewriting dataset (section 3.3) micro tasks is a probabilistic dataset in both the number and order of emissions. Therefore just as we had to define a symbol rewriting metric in section 4.2 we need to define a new accuracy measure for microtasks. While the microtask accuracy for verify task is simply contingent on the final emission, which is binary, the accuracy for 'or ', 'and ', 'not ', tasks are separately defined as follows:

- **or** - by checking the presence of any of the input token in predicted sequence.
- **and** - by checking the presence of all of the input tokens in predicted sequence.
- **not** - by ensuring that the set of all the input tokens preceded by negation and the predicted sequence are disjoint. This is followed by the same test as above for all the tokens not preceded by negation.

6.2.3 Hyperparameters

Test Setting: For baseline, learned guidance and hard guidance (for both gru and lstm)

- Hidden=Embedding = 128
- Attention = pre-rnn
- Attention Method = MLP
- Epochs=30 (early stopping owing to visual inspection of accuracy curves till 100 epochs. Model starts overfitting after 30 epochs.)
- Optimizer = Adam

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Hard AG works learning AG is hard Focus should be on developing methods to learn the trace better.

7.2 Future Work

Understander executor Representation learning Latent space

I would like to break down the task of compositionality into two parallel sub-tasks as follows:

Storing Information meaningfully and efficiently

1. A more efficient data representation.
2. The new RNN cell designed with efficient memory allocation considerations.
3. Information Compression.

Retrieving Information efficiently

1. Attentive Guidance
2. Pondering
3. Curriculum Learning (Understander Executor).

References

Jiasen Lu, Caiming Xiong, Devi Parikh, and Richard Socher. Knowing when to look: Adaptive attention via a visual sentinel for image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 6, page 2, 2017.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. 2014a. ISSN 09205691. doi: 10.3115/v1/D14-1179.

Kyunghyun Cho, Bart Van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation : Encoder Decoder Approaches. *Ssst-2014*, pages 103–111, 2014b. doi: 10.3115/v1/W14-4012.

Zoltn Gendler Szab. Compositionality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2017 edition, 2017.

Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. ISSN 10959203. doi: 10.1126/science.aab3050.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. 2017.

Christopher Olah. Understanding LSTM Networks – colah's blog, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

I. Neath and A. Surprenant. *Human Memory*. Cengage Learning, 2013. ISBN 9781111834807.

JL Elman. Finding structure in time* 1. *Cognitive science*, 211(1 990):1–28, 1990. ISSN 0364-0213.

Noah Weber, Leena Shekhar, and Niranjan Balasubramanian. The Fine Line between Linguistic Generalization and Failure in Seq2Seq-Attention Models. 2018.

Adam Liška, Germán Kruszewski, and Marco Baroni. Memorize or generalize? Searching for a compositional RNN in a haystack. 2018. doi: 10.1145/nnnnnnn.nnnnnnn.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective Approaches to Attention-based Neural Machine Translation. 2015. ISSN 10495258. doi: 10.18653/v1/D15-1166.

E.T.F.I.M. Craik, E. TULVING, F.I.M. Craik, Oxford University Press, and University of Cambridge. *The Oxford Handbook of Memory*. Oxford Library of Psychology Series. Oxford University Press, 2000. ISBN 9780195122657.

Huiting Zheng, Jiabin Yuan, and Long Chen. Short-Term Load Forecasting Using EMD-LSTM neural networks with a xgboost algorithm for feature importance evaluation. *Energies*, 10(8), 2017. ISSN 19961073. doi: 10.3390/en10081168.

Gerhard Jäger and James Rogers. Formal language theory: Refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598): 1956–1970, 2012. ISSN 14712970. doi: 10.1098/rstb.2012.0077.

G Miller. The Magic Number Seven, Plus or Minus Two: Some Limits on our Capcity for Processing Information. *The Psychological Review*, 63(2):81–97, 1956.

G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. ISSN 00368075. doi: 10.1126/science.1127647.

Alex Graves. Adaptive Computation Time for Recurrent Neural Networks. pages 1–19, 2016. ISSN 0927-7099. doi: 10.475/123.

Eric Schulz, Josh Tenenbaum, David K Duvenaud, Maarten Speekenbrink, and Samuel J Gershman. Probing the Compositionality of Intuitive Functions. In D D Lee, M Sugiyama, U V Luxburg, I Guyon, and R Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3729–3737. Curran Associates, Inc., 2016.

Paul J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. ISSN 15582256. doi: 10.1109/5.58337.

Marco Baroni, Armand Joulin, Allan Jabri, Germàn Kruszewski, Angeliki Lazaridou, Klemen Simonic, and Tomas Mikolov. CommAI: Evaluating the first steps towards a useful general AI. pages 1–9, 2017.

Yoshua Bengio, Paolo Frasconi, and Patrice Simard. The problem of learning long-term dependencies in recurrent networks. *IEEE International Conference on Neural Networks - Conference Proceedings*, 1993-January:1183–1188, 1993. ISSN 10987576. doi: 10.1109/ICNN.1993.298725.

Dieuwke Hupkes, Anand Singh, Kris Korrel, German Kruszewski, and Elia Bruni. Learning compositionally through attentive guidance. 2018.

N. Chomsky. Formal properties of language. In D. Luce, editor, *Handbook of Mathematical Psychology*, page 2. John Wiley & Sons., 1963.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. ISSN 21682712. doi: 10.1109/TIT.1956.1056813.

Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June-2015:3156–3164, 2015. ISSN 10636919. doi: 10.1109/CVPR.2015.7298935.

Mihir Bellare and Shafi Goldwasser. The Complexity of Decision Versus Search. *SIAM Journal on Computing*, 23(1):97–119, 1994. ISSN 0097-5397. doi: 10.1137/S0097539792228289.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. (Nips), 2017. ISSN 0140-525X. doi: 10.1017/S0140525X16001837.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. (2), 2012. ISSN 1045-9227. doi: 10.1109/72.279181.

J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. ISSN 0027-8424. doi: 10.1073/pnas.79.8.2554.

Peter Anderson, Xiaodong He, Chris Buehler, Damien Teney, Mark Johnson, Stephen Gould, and Lei Zhang. Bottom-up and top-down attention for image captioning and visual question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

Manish Chablani. Sequence to sequence model: Introduction and concepts, 2017. URL https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. pages 1–15, 2014. ISSN 0147-006X. doi: 10.1146/annurev.neuro.26.041002.131047.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation (No. ICS-8506). *California Univ San Diego La Jolla Inst For Cognitive Science*, 1:318–362, 1986. ISSN 1-55860-013-2. doi: 10.1016/B978-1-4832-1446-7.50035-2.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014. ISSN 09205691. doi: 10.1007/s10107-014-0839-0.

F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.

Sepp Hochreiter and Jurgen Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1–32, 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.