University of Amsterdam

MSc Artificial Intelligence

Track: Natural Language Processing

Master Thesis

Learning to follow instructions

by

Rezka Aufar Leonandya

11390069

July 24, 2018

36 ECTS

January - August 2018

Supervisor:
Dr Germán Kruszewski
Dieuwke Hupkes
Dr Elia Bruni

Examiner:
Dr Willem Zuidema
Assessor:
Dr Efstratios Gavves

Institute of Logic, Language, and Computation

Abstract

An artificial intelligence (AI) systems will be useful to humans if it can communicate with us: understanding our intention, performing actions that we assigned to it, and learning new skills from interactions. Hence, understanding and executing instructions given by humans is an important step to develop general useful AI. A system should be able to carry out new instructions by harnessing previously acquired knowledge of other instructions. It also needs to be flexible to master a continuous flow of new instructions while at the same time maintaining a data-efficiency in understanding and executing new instructions. This thesis describes our research in exploring deep learning models to follow instructions that it has never encountered before. We explore two task consisting of a different type of instructions. The first task consists in understanding and processing descriptions of subregular languages. In this task, the instructions come in the form of grammar specifications. The second task that we explore is the SHRDLURN task. In this task, the instructions come in the form of natural language utterances. Our results on the first task show that to some extent, a learning agent can generalize to unseen situations. We found that an LSTM is capable of generalizing if it can make use of attention mechanism to attend to the relevant parts of the input grammar. Our results on the second task demonstrate that a learning agent can also learn some useful information to adapt to instructions it has never seen before by exploiting prior knowledge which is learned automatically from another set of instructions. We found that when neural network models are exposed to similar language instructions to the instructions they have been trained on, they can adapt efficiently to unseen words. Even if the unseen language instructions are very different, the models can still make use of the automatically learned prior knowledge.

Contents

Chapter 1

Introduction

This thesis describes an exploration to create learning agents that can learn to follow instructions based on input-output examples. In order to follow instructions, a learning agent needs to acquire the right interpretation function for the task. We explored two different tasks, namely subregular language and the SHRDLURN task. For the subregular language task, the goal is to explore a method to infer how to recognize strings that match a given grammar. For the SHRDLURN task, the goal is to interpret natural language instructions on the SHRDLURN environment. We explored various architectures that best suits each task with a recurrent model as the main component.

In this chapter, we present the motivation of this thesis and outline the objectives we seek to achieve.

1.1 Motivation

The current landscape of state-of-the-art deep learning systems in most natural language processing and natural language understanding tasks remains constrained to domains where the amount of training data is substantially abundant. Although the state-of-the-art deep learning systems achieve impressive performance on variety of tasks, these systems have yet to exhibit humans efficiency in learning a new tasks, as humans do not need a lot of data to learn a certain tasks and to solve a new problems [Lake et al., 2016]. Moreover, humans can produce a potentially infinite number of novel combinations from known components [Chomsky, 1957, Montague, 1970], making them effective learners that can rapidly acquire and generalize knowledge to new tasks and situations. Hence, in the long run, it is desirable to have a learning agent that can learn from small data and is also able to generalize to unseen situations, just like humans do.

[Baroni et al., 2017] describe in their paper that progress in AI needs to be measured on mastering a continuous flow of tasks, with data-efficiency in solving new tasks as a fundamental evaluation component. In the same spirit as them, we aim to create a learning agent to learn to follow instructions as an endeavor to create general useful artificial intelligence systems.

Learning to follow instructions is an important ability especially if we want an AI that is useful for us. A desired outcome to have in the future is a learning agent that can interact with humans and learn to solve the task at hand by interpreting the instructions just from a handful of interactions. In order to understand the instructions, an agent needs to be able to induce an algorithm to solve the task directly from a handful of input-output examples provided by the human user. Furthermore, having acquired the right algorithm for the task, an agent also needs to be able to generalize to unseen situations as humans do. To be able to generalize, an agent needs to exploit prior knowledge which is learned automatically from previous interactions. This learning problem is particularly challenging because an agent has to be able to acquire the right interpretation function to be able to generalize in data-efficient settings.

To this end, we explored two different task, namely subregular language and SHRDLURN task. The former is a controlled setting where the instructions are given in the form of grammar specifications, whereas the latter features more varied interactions and its instructions come in the form of (natural or unnatural) language utterances. For each task, we perform an architecture search to determine which recurrent based architecture best suits each task.

1.2 Objectives

The objectives of this thesis is to explore a method to create a learning agent that can learn to follow instructions. Specifically, we aim to develop a method that can make learning agents able to generalize to unseen tasks and situations and able to learn from small data. To this extent we experiment with two tasks and here we specify each tasks objectives which are:

- Subregular languages: exploring a method to interpret grammars' specifications and learning to generalize to new unseen grammars in a zero-shot way.
- SHRDLURN: Exploring a method to learn inductive biases automatically from training data and use the pretrained systems to adapt to the language instructions of a new speakers.

1.3 Overview

The motivation of this thesis is presented in Section 1.1 and the objectives that we seek are mentioned in Section 1.2.

In Chapter 2, we provide the necessary background of the models that we used in this thesis. Specifically, we discuss feedforward neural networks, bag-of-words model, recurrent neural networks along with its variant long short-term memory networks, sequence to sequence architecture, attention mechanism, pondering, and convolutional sequence model.

In Chapter 3, we present: an overview of the task, several related works to our experiment, the dataset that we use for this experiment, the details of an architecture used in this experiment, the result of the experiments itself, and our findings and analysis regarding our experiment. The chapter is closed by a conclusion.

Chapter 4 is where we present our detailed work on SHRDLURN task. We describe: the introduction and the overview, several related works to our experiment, an approach that we propose to tackle existing problem, the dataset that we use in this experiment, the details of an architecture used in this experiment, the result of the experiment itself, our findings and analysis regarding the experiment, and the conclusion of the experiment.

Chapter 5 is the last chapter of this thesis, where we conclude our work and present some possible direction for future work.

Chapter 2

Models

In this chapter, we present several architectures or models that we used in this thesis. As stated in Chapter 1, in this work we are dealing with two different types of instructions that the learning models need to follow. First is subregular languages, where the instructions come in the form of a grammars specifications consisting of k-factors, and second is SHRDLURN, where the instructions come in the form of natural or unnatural language utterance. For both tasks, the goal is to have a general learning system that can adapt to new situations it has never seen before. The following models are models that we use to perform an architecture search to determine which architecture best suits each task.

2.1 Feedforward neural networks

Feedforward neural networks are computing systems made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs [Caudill, 1987]. Feedforward neural networks are typically organized in layers. Layers are comprised of a number of interconnected nodes containing an activation function, which typically is a non-linear function. The network receives a pattern via the input layer. The input layer then transmits the input pattern to the upper hidden layers which transform the input pattern via a linear combination and squashing nonlinearity. The hidden layers is linked to an output layer which produces a prediction for regression or classification task using the inputs from the hidden layers. Feedforward neural networks can also be seen as a directed acyclic graph describing how functions are composed together. For example, we might have three functions $f^{(1)}, f^{(2)}, f^{(1)}$ connected in chain to form a network $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ [Goodfellow et al., 2016]. During training, the goal is to force $f(x)$ to match $f^*(x)$, which is the true distribution of the data approximated by the noisy, observable examples of the training data at hands.

Training a feedforward neural network is not much different from training any machine learning model. Most machine learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing function $f(x)$ by altering the parameters of the function. The way to minimize or maximize these function is done by taking a derivative with respect to the parameters and updating them by subtracting a small fraction of that gradient. This technique is called gradient descent. In neural networks, the method to compute the gradient is known as backpropagation. Backpropagation is simply an error propagation from the last layer to the first layer by computing the gradients of the loss function, which is the starting point to compute the gradients. If we know how to calculate the gradient of each function that composes the loss function, we can propagate the error from the back to the start.

Learning in feedforward neural network consists of repeatedly presenting it with input-output examples over a timespan which is known as epoch. When a feedforward neural network is initially presented with an input, it makes a random prediction based on its current weights. It then receives the correct label and sees how far its answer from the actual one as measured by loss function. Based on that, it appropriately adjusts the connection weights. The weights adjustment are performed with gradient descent by taking a derivative of the loss function with respect to the parameters. Loss function or cost function is the objective function that the model needs to minimize. Typically, the

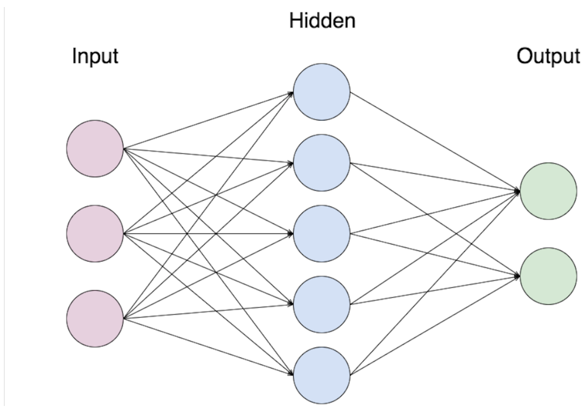

Figure 2.1: Architecture of a feedforward neural networks

objective is to fit a curve or a plane with respect to the data points.

Formally, we can formulate feedforward neural network as follows: given an i.i.d input $\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^m$, a feedforward neural network with a single layer and n hidden units would produce an output $\hat{\mathbf{y}} = (y_1, y_2, ..., y_n) \in \mathbb{R}^o$

$$\mathbf{h} = \phi(\mathbf{W}_x\mathbf{x} + \mathbf{b}_x) \tag{2.1}$$

$$\hat{\mathbf{y}} = \psi(\mathbf{W}_h\mathbf{h} + \mathbf{b}_h) \tag{2.2}$$

where $\mathbf{W}_x \in \mathbb{R}^{nxm}, \mathbf{b}_x \in \mathbb{R}^n, \mathbf{W}_h \in \mathbb{R}^{oxn}, \mathbf{b}_h \in \mathbb{R}^o$ are parameters, $\psi(.)$ and $\phi(.)$ are a nonlinear activation function. The predicted and the correct output is then used to adjust the network's weight by taking the derivative of the loss function $\mathcal{L}$ using backpropagation.

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} y_i \log \hat{y}_i \tag{2.3}$$

What makes neural networks different from any other machine learning model is that within each hidden layer of the network, the node contains a non-linear activation function which causes the loss functions to become non-convex. In theory, convex optimization converges starting from any initial parameters, whereas in non-convex optimization, the loss function has no such convergence guarantee and is sensitive to the values of initial parameters.

2.2 Bag of words models

Word embeddings, real-valued vector representations of words produced by distributional semantic models, are one of the most popular tools in modern natural language processing. They are trained on natural language corpora using the context in which words occur, i.e., by looking at lots of word in context with no other information provided about their semantics. The first popular embedding model is that of [Mikolov et al., 2013]. The embedding model calculates the probability of a word given the surrounding context or the other way around, and learn vector representations of the word. The learned word representations are able to capture meaningful syntactic and semantic regularities. The regularities are observed as a simple vector offset based on cosine distance. For example, if we denote the vector for word i as w_i, we can do a vector composition to answer a word analogy task like $w_{king} - w_{man} + w_{woman} =?$ and arrive at the result w_{queen}.

A bag of words model is a straightforward method to represent phrases and sentences using word embeddings [Joulin et al., 2017]. In addition to representing words as in the word embedding model, the bag of words model also represents phrases as n-grams to capture some partial information about the local word order. Formally, we can define the bag of words model as follows:

$$w_x = \phi(\frac{1}{N}\sum_{i}^{N} Ax_i) \tag{2.4}$$

Sentence	Unigram	Bigram	Trigram
I work a lot	I work a lot	I work work a a lot	I work a work a lot

Table 2.1: Bag of words example

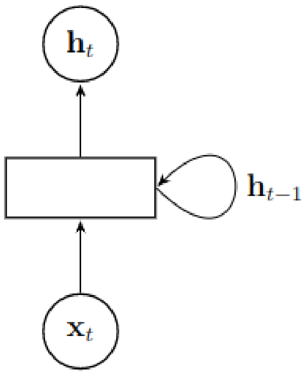

Figure 2.2: Architecture of an RNN. To compute the hidden state $\mathbf{h}_t$, the previous hidden state $\mathbf{h}_{t\text{-}1}$ is included as input along with $\mathbf{x}_t$

where $w_x \in \mathbb{R}^d$ is the sentence x representation in d dimension, N is the total number of words and ngrams in the sentence, x_i is the generated word or ngrams of the sentence, A is the weight parameter, and ϕ is a linear function which generates the sentence representation. Table 2.1 shows the generated words and ngrams from a sentence.

2.3 Recurrent neural networks

Recurrent neural networks (RNN) are neural networks designed to make use of sequential information by modeling dependencies in an input sequence [Elman, 1990]. RNNs are called recurrent because of their recurrent connections which allow the output to be dependent on the previous computations. The recurrent connections serve as a memory of what has been computed so far by including the hidden state from the previous timestep as input to the current timestep. The advantage of using RNN over feedforward neural networks is that RNN can have traveling signals in both directions. The advantage of using RNN over feedforward neural networks is that RNN can have signals traveling in both directions by introducing loops in the network, whereas feedforward neural networks allow signals to travel one way only: from input to output. There is no feedback in feedforward neural networks. The output of any layer does not affect that same layer.

An illustration of an RNN architecture is shown in Figure 2.2. The loop in the figure depicts a recurrent connection in an RNN. Looking at the figure, we can see that the hidden state $\mathbf{h}_{t-1}$ is fed back as input at the next timestep to the network. When an RNN is unrolled over time, it looks a lot like a regular feedforward neural networks. The recurrent connection can be unrolled to produce a similar architecture to that of a feed-forward network, as shown in Figure 2.3. The figure depicts an unrolled RNN architecture, which looks very much alike to that of a 3-layer feedforward neural network with shared weights between layers. It is easy to imagine that unrolling an RNN for longer time steps would result in an architecture similar to that of a deep feedforward network with shared weights between layers. This effectively allows RNN to have an arbitrary number of layers compared to a feedforward network.

Given an input sequence $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_t \in \mathbb{R}^m$, at time step t, and an RNN with hidden n, we can formally define an RNN computation as follows:

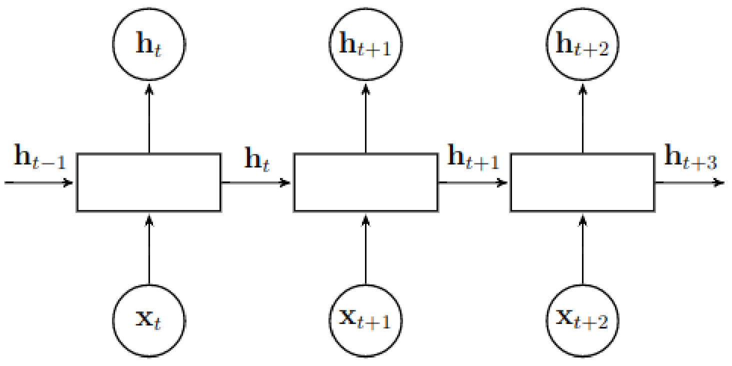

Figure 2.3: Architecture of an unrolled RNN for three time steps. This architecture is similar to that of a 3-layer feed-forward neural network.

$$\mathbf{h}_t = \phi(\mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t\text{-}1} + \mathbf{b}) \tag{2.5}$$

where $\mathbf{W}_x \in \mathbb{R}^{nxm}, \mathbf{W}_h \in \mathbb{R}^{nxn}, \mathbf{b} \in \mathbb{R}^n$ are parameters, and $\phi(.)$ is a nonlinear activation function. The initial hidden states $\mathbf{h_0} \in \mathbb{R}^n$ is usually initialized to a zero vector.

Like any other neural networks, training an RNN is usually done via gradient-based methods such as stochastic gradient descent (SGD). In order to use such methods, the gradients of the parameters are computed by taking a derivative of the loss function with respect to its parameters. These gradient computations involve a multiplication chain, which grows longer as the timestep increases. [Pascanu et al., 2013] proved that the derivative in RNN is bounded by one. Thus, with small values in the matrix multiplication chain, the gradients shrink exponentially fast, eventually vanishing completely when the input sequence is sufficiently long. Gradients contributions from "far-away" steps become zero, which makes the RNN ended up not learning the long-term dependencies. This problem is known as the vanishing gradient problem [Hochreiter, 1991, Bengio et al., 1994].

2.3.1 Long short term memory networks

Long short-term memory networks (LSTM) are modified RNN which are designed specifically to tackle the vanishing gradient problem [Hochreiter and Schmidhuber, 1997]. LSTM are able to remember information for a longer period of time by employing a memory cell. LSTM are able to read from or write to this cell, regulated by several gates. There are three components of these gates, each with its own purpose, namely:

- *forget gate* which decides how much contribution the memory cell from the previous time step is worth remembering, and decides which irrelevant stuff to forget in the cell state.
- *input gate* which selectively update the memory cell with the current input
- *output gate* which decides how much of that value is going to be the output at the current time step.

Given a sequential input $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_t \in \mathbb{R}^m$, at each time step t, an LSTM with n hidden units can be formulated as follows:

$$\begin{pmatrix} \mathbf{i} \\ \mathbf{o} \\ \mathbf{f} \\ \mathbf{g} \end{pmatrix} = \mathbf{W}_x\mathbf{x}_t + \mathbf{W}_h\mathbf{h}_{t\text{-}1} + \mathbf{b} \tag{2.6}$$

$$\mathbf{c}_t = \sigma(\mathbf{f}) \odot \mathbf{c}_{t\text{-}1} + \sigma(\mathbf{i}) \odot \tanh(\mathbf{g}) \tag{2.7}$$

$$\mathbf{h}_t = \sigma(\mathbf{o}) \odot \tanh(\mathbf{c}_t) \tag{2.8}$$

where $\mathbf{c}_t$ and $\mathbf{h}_t$ are the memory cell and the output cell at time step t, $\mathbf{W}_h \in \mathbb{R}^{4nxn}$, $\mathbf{W}_x \in \mathbb{R}^{4nxm}$, $\mathbf{b} \in \mathbb{R}^{4n}$ are parameters, symbol $\odot$ denotes an element-wise multiplication, and functions $\sigma(.)$ and $\tanh(.)$ are applied element-wise, $\sigma(\mathbf{f}), \sigma(\mathbf{i})$, and $\sigma(\mathbf{o})$ are the forget, input, and output gate respectively.

From the equations, we can see that the LSTM can control how much contribution the previous and current time step has by modulating the three gates. One thing worth noticing is that in the derivative of the internal state of LSTM (Equation 2.7), there is no repeated weight as in RNN. As long as $\sigma(\mathbf{f}) = 1$, the error signal can be propagated perfectly to the previous timestep. This ability allows LSTM to model long-term dependencies much more effectively than RNN [Pascanu et al., 2013].

2.4 Sequence to sequence

Sequence-to-sequence (seq2seq) [Sutskever et al., 2014, Cho et al., 2014] was first introduced on the task of Neural Machine Translation (NMT) and have achieved great success in a variety of tasks since then. Seq2seq relies on the encoder-decoder paradigm. In encoder-decoder paradigm, a system first reads the input sequence using the encoder to build a vector representation of the input sequence, then a decoder produces the target sequence using the representation from the encoder. The encoder and decoder can be of any architecture, but typically it is an LSTM. Formally, we can define seq2seq as follows: given an input sequence $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_t \in \mathbb{R}^m$, the encoder would process the input sequentially as in Equation 2.6, 2.7, and 2.8, producing the final vector representation of the input sequence $\mathbf{e}$. The decoder then uses the vector representation $\mathbf{e}$ to generate the target sequence word-by-word. The decoder LSTM receives $\mathbf{e}$ as initial hidden states and a special start token $\mathbf{w}_{start}$ as input. The LSTM then computes the next hidden states with its internal function $\mathbf{h}_t = \phi(\mathbf{e}, w_{t-1})$. The resulting hidden states are fed to the output layer g to produce a vector of the same size as the vocabulary. Then a softmax is applied to the resulting vector to normalize it into a vector of probabilities (Equation 2.9).

$$i_t = \begin{cases} \text{argmax}_\theta \, \text{softmax}(g(\phi(\mathbf{e}, \mathbf{w}_{start}))) & t = 0 \\ \text{argmax}_\theta \, \text{softmax}(g(\phi(\mathbf{e}, \mathbf{w}_{t-1}))) & t > 0 \end{cases} \tag{2.9}$$

The resulting index is then used to pick the input for the next timestep $\mathbf{w}_{i_t} = \mathbf{w}_t$. This method aims at modeling conditional distribution over the output sequence $p(Y = (w_1^y, w_2^y, ..., w_t^y)|X)$, given an input sequence $X = (w_1^x, w_2^x, ..., w_t^x)$.

2.5 Attention

Attention is a mechanism that allows neural networks, particularly seq2seq model, to refer back to the processed input sequence by the encoder, instead of encoding all information to one fixed-length vector. It gives the decoder access to the internal memory of the encoder, which is the hidden states. The attention-based mechanism was first introduced in neural machine translation (NMT) by [Bahdanau et al., 2014]. As explained in Section 2.3, at each time step the RNN outputs a hidden states $\mathbf{h}_t$. On a typical seq2seq system, the decoder of the RNN maintains it's own internal hidden states $\mathbf{z}_{t'}$. At each time step t', the decoder uses the attention mechanism to compute weighted linear combination of the encoder hidden states. The combination is used by the decoder to produce an output.

There are two ways of computing the attention, which we will refer to here by using the terms *pre-rnn* and *post-rnn*. The difference between the two is whether to apply the attention before the recurrent step or after. In case of the former, the hidden states that are used to calculate the attention is of the previous timestep, whereas in case of the latter is of the current timestep.

2.5.1 Pre-rnn

In pre-rnn, the attention mechanism takes an input both the previous decoder hidden states $\mathbf{z}_{t'-\mathbf{1}}$, and one of the encoder hidden states $\mathbf{h}_t$, and returns a relevance scores $e_{t',t}$:

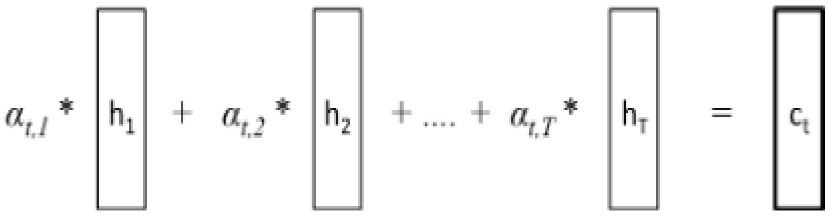

Figure 2.4: Attention Mechanism

$$e_{t',t} = f_{Att}(\mathbf{z}_{t'-1}, \mathbf{h}_t) \tag{2.10}$$

which is referred to as *score function* [Luong et al., 2015]. The function f_{Att} can be implemented by fully connected neural networks with a single layer where tanh() is the activation function [Bahdanau et al., 2014], or it can be a dot product between the one of the encoder hidden states $\mathbf{h}_t$ and the previous decoder hidden states $\mathbf{z}_{t'-1}$ [Luong et al., 2015]. These relevance scores are normalized to be positive and sum to 1.

$$\alpha_{t',t} = \frac{\exp(e_{t',t})}{\sum_{k=1}^{T} \exp(e_{t',k})} \tag{2.11}$$

Then the normalized scores are used to compute the weighted sum of the encoder hidden states

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha_{t',t} \mathbf{h}_t \tag{2.12}$$

which will be used by the decoder to update its own hidden states at the current time step by

$$\mathbf{z}_{t''} = \phi(\mathbf{y}_{t'-1}, \mathbf{c}_{t'-1}) \tag{2.13}$$

$$\mathbf{z}_{t'} = \psi(\mathbf{z}_{t''}, \mathbf{c}'_{t}) \tag{2.14}$$

where $\mathbf{y}_{t'-1}$ is the previous predicted label which becomes the current input and ϕ is the RNNs internal function and ψ is a linear layer which combines the temporary hidden states and the context-extended vector.

2.5.2 Post-rnn

In post-rnn, the attention mechanism takes an input the current temporary hidden states $\mathbf{z}_{t''}$ of the decoder and one of the encoder hidden states and calculate its relevant score.

$$\mathbf{z}_{t''} = \phi(\mathbf{y}_{t'-1}, \mathbf{c}_{t'-1}) \tag{2.15}$$

$$e_{t',t} = f_{Att}(\mathbf{z}_{t''}, \mathbf{h}_t) \tag{2.16}$$

Then the relevance scores and the weighted sum of the encoder hidden states are computed the same as in Equation 2.11 and 2.12. Finally, the $\mathbf{c}_{t'}$ is combined with the temporary hidden states to calculate the context-extended final hidden states fed to the next step as the hidden states and to the output layer.

$$\mathbf{z}_{t'} = \psi(\mathbf{z}_{t''}, \mathbf{c}_{t'}) \tag{2.17}$$

2.6 Pondering

In recurrent neural networks, the measure of computation time is the number of computation at every time step that is defined by the output length. Forcing a network to output at every time step means that the computation time it can use is decided by the length of the output, not by the difficulty of the problem. Alex Graves [Graves, 2016] introduced Adaptive Computation Time (ACT), an algorithm that allows RNN to adaptively decide how much computation to be carried out per time step. This algorithm is also known as "pondering". These pondering steps defines the number of computation times that are passed through the same recurrent layer (connected to the same input) before producing an output. Pondering augments neural networks output with a sigmoid *halting unit* whose activation determines the probability that computation should continue. The resulting *halting distribution* is used to produce the final vector for both the network output and the internal hidden state network propagated along the sequence.

Formally, a pondering step can be formulated as follows: given a sequential input $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x}_t$, at each time step t, a recurrent neural network computes the state sequence $\mathbf{h_1}, \mathbf{h_2}, ..., \mathbf{h}_t$ (Equation 2.8) and outputs the output sequence $\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y}_t$ by iterating the following equations from $t = 1$ to T:

$$\mathbf{y}_t = \mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y \tag{2.18}$$

where $\mathbf{W}_y$ and $\mathbf{b}_y$ are the output parameters.

Pondering modifies RNN computation step by allowing it to perform a variable number of state transitions and compute a variable number of outputs at each input step. Let $N(t)$ be the total number of updates performed at step t. Then define the *intermediate step sequence* $\mathbf{h}_t{}^1, \mathbf{h}_t{}^2, ..., \mathbf{h}_t{}^{N(t)}$ and *intermediate output sequence* $\mathbf{y}_t{}^1, \mathbf{y}_t{}^2, ..., \mathbf{y}_t{}^{N(t)}$ at step t as follows:

$$\mathbf{h}_t^n = \begin{cases} \phi(\mathbf{h_{t-1}}{}^N, \mathbf{x}_t{}^n) & n = 1 \\ \phi(\mathbf{h}_t{}^{n-1}, \mathbf{x}_t{}^n) & n > 1 \end{cases} \tag{2.19}$$

$$\mathbf{y}_t{}^n = \mathbf{W}_y \mathbf{h}_t{}^n + \mathbf{b}_y \tag{2.20}$$

where $\mathbf{x}_t{}^n$ is the pondering time-annotated input vector at time t, and ϕ is a parametric *state transition model* or the RNN internal function. In the original paper, the time-annotated input vector is augmented with a binary flag that indicates whether input step has just been incremented. To determine how many number of variable state transitions at each time step, an extra sigmoid halting unit is added to g the network output with an associated weight matrix $\mathbf{W}_g$ and bias $\mathbf{b}_g$.

$$\mathbf{g}_t^n = \sigma(\mathbf{W}_g \mathbf{h}_t^n + \mathbf{b}_g) \tag{2.21}$$

The activation of the halting unit is then used to determine the halting probability $\mathbf{p}_t^n$ of the intermediate steps:

$$\mathbf{p}_t^n = \begin{cases} R(t) & n = N(t) \\ \mathbf{h}_t^n & otherwise \end{cases} \tag{2.22}$$

where

$$N(t) = \min\{n^{'} : \sum_{n=1}^{N(t)-1} \mathbf{g}_t^n >= 1 - \epsilon\} \tag{2.23}$$

and the remainder $R(t)$ is defined as follows

$$R(t) = 1 - \sum_{n=1}^{N(t)-1} \mathbf{g}_t^n \tag{2.24}$$

and ϵ is a small constant whose purpose is to allow computation to halt if $h_t^n >= 1 - \epsilon$. Finally, the final hidden states and the output at the current time step t can be computed as follows:

$$\mathbf{h}_t = \sum_{n=1}^{N(t)} \mathbf{p}_t^n \mathbf{h}_t^n \qquad \mathbf{y}_t = \sum_{n=1}^{N(t)} \mathbf{p}_t^n \mathbf{y}_t^n \tag{2.25}$$

2.7 Convolutional sequence models

Convolutional neural networks (CNN) were first introduced for digit recognition [LeCun et al., 1989] and it resulted in a huge success in computer vision [Krizhevsky et al., 2012]. For sequence modeling, convolutional networks are less used compared to recurrent networks, but it has been applied to sequence problem for a long time [Waibel et al., 1989, Lecun and Bengio, 1995]. Recent research even shows that convolutional networks outperform recurrent networks on a range of different tasks [Bai et al., 2018]. Convolutional networks operate over a fixed size window of the input sequence which enables the simultaneous computation of all features for an input sequence. This contrasts to RNN which maintain a hidden state of the entire past that prevents parallel computation within a sequence. The convolutional network for sequence modeling that is further described here was intially introduced by [Gehring et al., 2017]. There are three main parts of the convolutional sequence models, namely position embeddings, convolutional block structure, and multi-step attention.

2.7.1 Position embeddings

Position embeddings are useful in convolutional sequence models since they give the model a sense of which portion of the sequence in the input or output it is currently dealing with. Formally, let $x = (x_1, x_2, ..., x_m)$ be the input elements and it's embeddings in distributional space as $\mathbf{w} = (\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_m)$, where $\mathbf{w}_j \in \mathbb{R}^f$ is a column in an embedding matrix $D \in \mathbb{R}^{V \times f}$. Furthermore, let $\mathbf{p} = (\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_m)$ where $\mathbf{p}_j \in \mathbb{R}^f$ be the embeddings of the input elements' absolute position. The absolute position is necessary to give the model sense of order. Then both of the embeddings are combined to obtain input element representations $\mathbf{e} = (\mathbf{w}_1 + \mathbf{p}_1, \mathbf{w}_2 + \mathbf{p}_2, ..., \mathbf{w}_m + \mathbf{p}_m)$.

2.7.2 Convolutional block structure

Convolutional networks operate over a fixed window size and perform a matrix multiplication between multiple windowed views of the input and the weights. Formally, let l be the total number of layers in the convolutional network, and $\mathbf{h^l} = (\mathbf{h}_1^l, \mathbf{h}_2^l, ..., \mathbf{h}_n^l)$ be the hidden states at each layer l where n is the length of the input sequence. Each layer performs a one dimensional convolution followed by a non-linearity. Each convolution kernel is parameterized as $W \in \mathbb{R}^{dxkd}, b_w \in \mathbb{R}^d$ and takes an input $X \in \mathbb{R}^{kxd}$ where k is the kernel size and d is the hidden size dimension. To enable deep convolutional networks, a residual connections is also added from the input of each convolution to the output of the sequence. Mathematically, the convolutional operation is defined as follows:

$$\mathbf{h}_i^l = \begin{cases} \tanh(\mathbf{W}^l[\mathbf{h^{l-1}_{i-k/2}}, ..., \mathbf{h^{l-1}_{i+k/2}}] + \mathbf{b^l_w} + \mathbf{h}_i^{l-1}) & l > 1 \\ \tanh(\mathbf{W}^l[\mathbf{x_{i-k/2}}, ..., \mathbf{x_{i+k/2}}] + \mathbf{b^l_w} + \mathbf{x}_i) & l = 1 \end{cases} \tag{2.26}$$

where i is the index of the input sequence and x is the input sequence. Zero padding is also needed to make sure the convolution operation outputs the same number of elements (n) at every layer.

2.7.3 Multi-step attention

Convolutional sequence models use the same attention mechanism as described in Section 2.5. The only differences is that it computes the context-extended hidden states at every convolutional layer. Also, in convolutional sequence models, the attention is computed *post-rnn* only. First, the resulting hidden states from applying convolution at each layer as in Equation 2.26 are calculated. Then, the attention can be applied by replacing Equation 2.16 with

$$e^{(l)}_{t',t} = f_{Att}(\mathbf{z}^{(l\text{-}1)}_{t'}, \mathbf{h}_t) \tag{2.27}$$

where $\mathbf{z}^{(l)}_{t'}$ is the resulting hidden states at layer l and then $\mathbf{h}_t$ is the hidden states of the encoder. Then, to compute the context-extended vector, replace Equation 2.11 and 2.12 with:

$$\alpha^{(l)}_{t',t} = \frac{\exp(e^{(l)}_{t',t})}{\sum_{k=1}^{T} \exp(e^{(l)}_{t',k})} \tag{2.28}$$

$$\mathbf{c}_{t'}^{(l)} = \sum_{t=1}^{T} \alpha_{t',t}^{(l)} \mathbf{h}_t \tag{2.29}$$

The $\mathbf{c}_{t'}$ vector is then combined with the temporary hidden states to calculate the context-extended final hidden states fed to the next step as the hidden states and to the output layer.

$$\mathbf{z}_{t'}^{(l)} = \psi(\mathbf{z}_{t''}^{(l\text{-}1)}, \mathbf{c}_{t'}) \tag{2.30}$$

Chapter 3

Subregular Languages

In this chapter we describe our experiment on task involving subregular languages. In order to be able to interpret and understand instructions, machines need to be able to acquire the correct algorithms. In learning situations, it is preferable that the machines learn these algorithms directly from data without human interventions. Here, we experiment with a controlled setting where we develop a learning agent to classify input string given a grammar specifications. Specifically, the task for the learning agent is to recognize whether an input string belongs to a certain language or not according to a given grammar. Given a grammar specifications and an input string, a learning agent needs to determine whether the given input string matches the grammar or not. An illustration is given in Figure 3.1.

3.1 Definition

Subregular languages are a class of formal languages that extend the classical Chomsky hierarchy [Jäger and Rogers, 2012]. In terms of formal complexity, they lie within the class of regular languages. They are characterized by a set of strings that can be described without using the full power of Finite State Automata (FSA's). In regular languages, on the other hand, to detect whether a string belongs to a certain language, FSA's needs to be employed.

There are several well-studied classes of subregular languages. They are the Strictly Local (SL), Locally Testable (LT), Non-Counting (NC) which were studied by [McNaughton and Papert, 1971], Locally Threshold Testable (LTT) [Thomas, 1982], Piecewise Testable (PT) [Simon, 1975], and Strictly Piecewise (SP) [Rogers et al., 2010] languages. In this work, we only define and experiment with k-LT class, k-locally testable class, which we define below.

Definition k-locally testable languages can be defined as follows: Let Σ denote a finite set of symbols, the alphabet, and Σ^k the set of elements of strings of length k that is created by concatenating elements from Σ. We refer to the latter as k-factor. k-locally testable languages can be characterized by a finite set of k-factors as follows. A k-locally testable grammar is a set of k-factors $G \subseteq \Sigma^k$ and the language of G is the stringset $L(G) = \{w | w \subseteq \Sigma^*\}$ where w is a string that does not contain one of the k-factors in G. The grammar G is the set of forbidden k-factors. Any k-factors v in Σ^k which is not in G is thus permissible. Consequently, all strings containing one of the k-factors in G as a substring are not in $L(G)$ [Avcu et al., 2017]. For this to work, the length of w must be greater than k, i.e., $* \geq k$.

Abstract processing models for subregular language are called scanners [Jäger and Rogers, 2012]. Scanners have a sliding window of width k, which is the same as the length of the k-factor. The scanner moves across the target string moving one character at a time, choosing out the substring and matching it with k-factors that are in $L(G)$. Scanners have no internal state. Their behavior, at each point of time in the computation, depends only on the symbols which fall within the window at that point.

3.2 Related Work

The subregular language task used in this chapter was heavily influenced by CommAI Framework, introduced by [Baroni et al., 2017]. In the paper, they introduce a set of desiderata for achieving general AI, together with a platform to test machines on how well they satisfy such desiderata. The framework provides an environment which presents a simplified regular expression to the learner. It then asks the learner to either recognize or produce a string matching the description. The learner produces a response to the environment and receives a linguistic feedback based on it's performance (e.g. reward function). The framework also provides the learner a different grammar on test time and the grammars become more complex with time. This complexity allows to test the learner's capability to perform a compositional action. Skills such as chunking sequences into characters and parsing them into understandable parts would play an important role to help the learner to generalize across tasks.

Although commAI is a great framework to develop learning agents, it contains several challenging aspects for a model to learn from. First, it contains a produce command where a model needs to be able to produce strings given grammar specifications. Second, its feedback comes in a linguistic form, meaning that the learner initially does not know whether the feedback contains a positive or negative signal. The learner has to figure out the meaning of the feedback by itself through interaction. Here we opt for a less challenging setup to test learning agents' generalization capabilities. We chose subregular language because here we only try to ask the learner to verify if a string belongs to a certain language given grammar specifications.

Another related study was presented by [Avcu et al., 2017]. They tested the capabilities of RNN and LSTM in recognizing subregular language, particularly the SL and SP classes. They define a training set and a testing set with different lengths in order to test the model's generalization. While they study the generalization skills of a model trained on a single grammar, we focus on learning to interpret the grammars themselves and learning to generalize to new unseen grammars in a zero-shot way. We aim to achieve this by using a non-overlapping instances between the training and testing data.

3.3 Dataset

For this task, we generated a new dataset. The dataset consists of a set of tuples containing a subregular grammar G, a target string x, and the corresponding label y, as shown in Figure 3.1. The goal of this task is to develop a model which can read a grammar and a string and decide whether the string belongs to the language defined by the grammar. In practice, the model needs to be able to detect whether a target string contains a k-factor of the grammar G.

We generated two version of datasets that differ in how the data is split between train, validation, and test. One has an overlapping (**overlap**) grammar across its train, validation, and test set, and the other one has a non-overlapping grammar (**non-overlap**) across its train, validation, and test set. The **non-overlap** dataset contains different grammars for each train, validation, and test set but each grammars in each set are generated from the same alphabet characters. In other words, a system will not encounter an exact same grammar during training, validation, and testing but it is exposed to the alphabet characters. An example is presented in Figure 3.2.

This dataset is used to test how far the model could generalize to unseen situations. This means that, if the model acquires the right interpretation function, which in this case is a string matching function, the model should be able to understand grammars' specifications it has not seen during training. For the experiment we vary the number of grammar and the number of examples for each of the grammar, as shown in table 3.1. For both datasets, we generate 40000 instances, where 32000 are for training, 4000 are for validation, and 4000 are for testing, following a common split of 80-10-10. Each grammar in the dataset contains an equal number of positive and negative examples. For this work, we limit the number of k-factors in the grammar to be 5, the length of one k-factor to be 5, and length of the input string to be 15.

Grammar	**Target string**	**Label**
aff#bdf#cce#cdd#def	baadbdddecdedaffbcde	0
add#bce#bde#cce#dff	ceeeafecbfbcacfbeeed	1

Figure 3.1: Subregular language example. The first row is a negative example (*aff* exists in the input string and in the input grammar) and the second row is a positive example (no k-factors in the grammar exists in the input string)

Dataset	**Set**	**Grammar**	**Target string**	**Label**
overlap	Train	aff#bdf#cce#cdd#def	baadbdddecdedaffbcde	0
	Test	aff#bdf#cce#cdd#def	ceeeafecbfbcacfbeeed	1
non-overlap	Train	aff#bdf#cce#cdd#def	baadbdddecdedaffbcde	0
	Test	add#bce#bde#cce#dff	ceeeafecbfbcacfbeeed	1

Figure 3.2: **non-overlap** and **overlap** dataset example. Same grammar appears in both train and test in **overlap** dataset, but not in **non-overlap** dataset

3.4 Models

We experiment with an LSTM model [Hochreiter and Schmidhuber, 1997]. The goal of the experiment is to see if an LSTM can learn to learn the mechanics of a subregular language, following instructions on how to classify strings given a particular grammar. Specifically, we investigate to what extent an LSTM can remember seen k-factors in the grammar and recognize the same k-factors in the target string to make the correct decision. We experiment with three different models, which we discuss below.

lstm For a detailed description of LSTM networks see Section 2.3.1. Specifically, the LSTM takes a concatenated input of grammar G and the input string, which is shown in Figure 3.1. At the end of the sequence, the hidden state $\mathbf{h}_T$ is fed to an output layer, followed by a softmax, to produce a probability distribution over the class, which in this case is either 0 or 1. The loss function is a cross entropy loss which is defined as in Equation 2.3.

lstm+attn Here, the LSTM is equipped with an attention mechanism, as explained in Section 2.5 except for that the attention is not applied to a seq2seq model, rather, to a vanilla LSTM. We chose not to use seq2seq architecture because we hypothesize this task is more suitable to be tackled with a vanilla LSTM, as we do not need to generate any output sequence. The attention is applied when processing the input string, allowing the LSTM to look back at the input grammar. Formally, let $x_1, x_2, ..., x_g$ be the input grammar, and $x_{g+1}, x_{g+2}, ..., x_m$ be the input string, and m be the length of the concatenated input grammar and input string. The attention is computed as in Section 2.5 by processing the input grammar as the encoder hidden states $\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_g$ and the input string as the decoder hidden states $\mathbf{z}_{g+1}, \mathbf{z}_{g+2}, ..., \mathbf{z}_m$.

Number of grammar	Number of example for each grammar
5	8000
10	4000
50	800
100	400
500	80
1000	40
5000	8
10000	4

Table 3.1: Dataset for subregular language

lstm+attn+act For the third model , we equip **lstm+attn** with a pondering mechanism when processing the input string (See Section 2.6 for details). However, instead of adapting the computation time, we fix the number of pondering step to be the same as the number of k-factors that consists in the grammar. The proposed approach aims to allow the model to attend to each of the N k-factors at each of the timestep of the input string.

Formally, given input sequence $x_1, \cdots, x_m$, that is composed of a grammar string $x_1, \cdots, x_g$ and a target string $x_{g+1}, \cdots, x_m$, we compute a sequence of hidden states $h_1, \cdots, h_g$ of the input sequence. Then, starting from input x_{g+1}, we apply attention and pondering in the computation of the hidden states h_t^n, (where n, $1 \leq n \leq N$, stands for the current pondering step and t starts from $g+1$ in this case), replacing Equation 2.19 as follows:

$$h_t^n = \begin{cases} S(h_{t-1}^N, x_t^n) & n = 1 \\ S(h_t^{n-1}, x_t^n) & n > 1 \end{cases} \tag{3.1}$$

where $x_t^n = f(x_t, n)$ is the pondering time-annotated input vector and we define $h_g^N := h_g$. Then, an attention weights and context-extended vector can be computed as in Section 2.5 to produce the output.

This formalization tries to express that for every input target string symbol $x_{t>g}$, the symbol is fed to the network for N consecutive time steps, with a small alteration to the input each time to feed a clock-like signal.

3.5 Experiments

We conducted a hyperparameter search for each 8 dataset (both overlap and non-overlap) and all three models, exploring the number of LSTM's layers (1, 2), the size of hidden layers (32, 64, 128, 256), and dropout rate (0, 0.2, 0.5). For each dataset, we acquire the best hyperparameters for each models, resulting in 48 best configuration. We report the best hyperparameters for each model in the Appendix.

As can be seen from Table 3.2, all models reach a reasonably good performance on the overlap set both for validation and test set (see column 3,5, and 7 in Table 3.2). They all achieve more than 70 % validation and test accuracy. On the **non-overlap** dataset, being presented with unseen k-factors, we can see that **lstm** reaches only 64 % on validation set, whereas **lstm+attn** and **lstm+attn+act** reaches 88 % and 76 %, respectively. However, on test set, none of the models seems to be generalizing well. The result on validation and test indicates that attention mechanism helps the model match the processed k-factors to the input string, as both **lstm+attn** and **lstm+attn+act** perform better than **lstm** model. On the other hand, it seems that the pondering mechanism is not helping much, except on non-overlap testing accuracy where it reaches 58 %, four percent higher than the **lstm+attn** model.

Figure 3.3 shows both validation and test accuracy for each model on every number of grammar of non-overlap and overlap dataset. Looking at it closely, we can see that **lstm** seems to be failing as the number of grammar grows higher on the **overlap** dataset, even though that the **lstm** has seen the grammar before. On the other hand, **lstm+attn** achieves better and consistent performance irrespective of the number of grammar on the overlap dataset. The **lstm+attn+act** model reaches better accuracy on the **overlap** dataset compared to the **lstm** model, but it is suffering the same problem, namely the declining performance as the number of grammar grows higher. On the non-overlap dataset, we can see that **lstm** accuracy is slightly above chance to a random guess for every number of grammar. On the contrary, **lstm+attn** achieves good performance on the validation set when the number of training grammars is relatively large. It is also the case for **lstm+attn+act** but it reaches lower accuracy compared to that of **lstm+attn**. Overall, **lstm+attn+act** perform no better than the **lstm+attn**.

To investigate the model's capabilities further, we also include the **reject** accuracy and **accept** accuracy of validation and test set in Table 3.2. Reject accuracy represents the model accuracy on examples containing k-factors in the input string, whereas the accept accuracy represents the model accuracy on examples not containing k-factors in the input string. Reject and accept accuracy corre-

	lstm		**lstm+attn**		**lstm+attn+act**	
	non-overlap	**overlap**	**non-overlap**	**overlap**	**non-overlap**	**overlap**
Train accuracy	78	84	99	99	89	94
Validation accuracy	64	75	88	97	76	86
Test accuracy	52	75	54	97	58	86
Validation (reject) accuracy	59	76	88	97	76	86
Validation (accept) accuracy	70	74	87	97	76	86
Test (reject) accuracy	44	76	23	97	47	86
Test (accept) accuracy	60	73	85	96	68	87

Table 3.2: Model's average accuracies with the winning hyperparameters (in percentage)

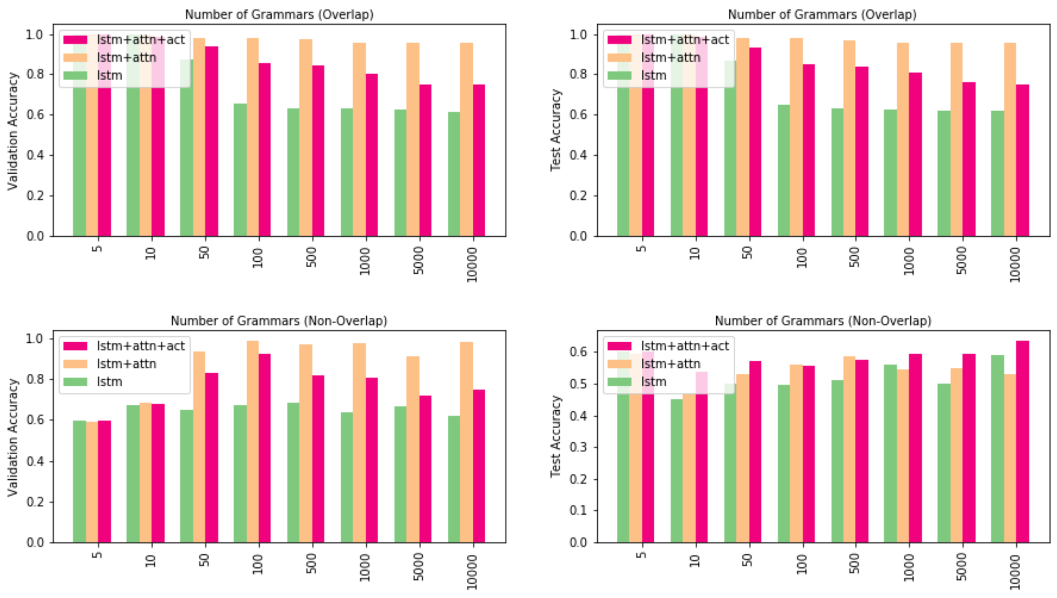

Figure 3.3: Details of validation and test accuracy evaluated on different number of grammar.

sponds with the recall of positive and negative examples, which is computed as:

$$\frac{tp}{tp + fn} \tag{3.2}$$

where tp is the correct prediction and fn is the misclassified prediction for both reject and accept cases.

Perhaps an interesting observation is all three models' average performance on the test accuracy of the non-overlap dataset. As can be seen in Table 3.2, we can see that all models reach a reasonable accept accuracy, but it seems to be struggling on the reject accuracy. This indicates that the challenge of the task lies on matching k-factors on the input string and the input grammar. The aforementioned analysis does not seem to apply for the validation set.

3.6 Discussion

To gain more insight in how the attention mechanism operates in this context, we present the attention mask of both correctly and incorrectly predicted reject examples. Figure 3.4 shows the attention mask of single reject examples for correctly and misclassified instances from the 10000 grammar's non-overlap dataset of the **lstm+attn** model. As we can see that on the correctly classified examples, the model correctly attends to the last character of k-factors and propagates the signal until the last character of the input string. On the misclassified examples, we see that the model fails to attend to the k-factor. For the **lstm+attn+act** model, we found that the attention mask does not differ much across

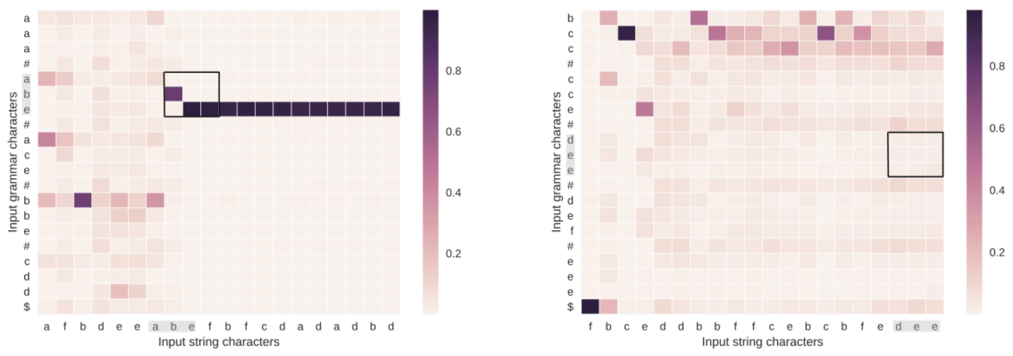

Figure 3.4: Left: Attention mask of correctly classified instance of reject cases. Right: Attention mask of misclassified instance of reject cases. The highlighted characters are of k-factors in the input grammar and input string. The box indicates the attention mask that the model allocates when processing each character of present k-factor in the input string over attending to the character of the same k-factor in the input grammar.

pondering steps , i.e., the model attends to the same part at every pondering step. This revokes our initial hypothesis of the network attending to different k-factors in the grammar at each pondering step. Since the attention mask of both **lstm+attn** and **lstm+attn+act** models the same thing, here we only present the attention mask of **lstm+attn**. The goal is to gain further understanding if the attention learns to attend to relevant parts, not to compare the attention mask of both models.

Recall that the output is produced by feeding the hidden state at the end of the sequence h_T to an output layer. However, in case of reject examples, a decision can be made as soon as the model knows if a k-factor exists in the input string without having to process every character until the end of the sequence. Thus, it might also be interesting to know the model's prediction at the last character of an existing k-factor in the input string. Intuitively, if the model learns the true functions, then it should make a correct decision after knowing that a k-factor exists in the input string. The visualization is made from the reject examples of the 10000 grammar dataset.

In addition to that, we want to know if assigning high weights to the relevant parts in the input grammar helps the model in making a correct final prediction. To evaluate this, we calculated the average attention mask of the k-factor existing in the input string over all instances in the reject examples, i.e., we calculated and visualized the average attention mask inside the bounding box in Figure 3.4 over all instances in the reject examples. Then, we visualized four different cases of average attention mask that can be classified according to two aspects. The first aspect is whether the model makes a correct prediction at the end of the sequence. The second aspect is whether the model makes a correct prediction at the last character of the k-factors in the input string. We refer to the former as **predict** and the latter as **match** in Figure 3.5 and 3.6. This results in four different classes of average attention mask visualization.

Figure 3.5 shows the average attention mask on the validation set. As we can see, when the model allocates high probability to the attention mask of the last character of the k-factor, it makes a correct prediction at that time step and a correct final prediction as well on 2235 examples. This indicates that attending with high probability to the relevant parts helps the model in making the final prediction. We can also observe there are 44 instances where the model makes a correct prediction at the last character of the k-factor, but it fails to propagate the signal to the end of the sequence. On the other hand, Figure 3.6 shows the average attention mask on the test set, where the **lstm+attn** is not performing well. We can see that the model does not allocate high probability to the relevant parts and hence could not make a correct prediction. These results indicate that attention is an essential part for matching the k-factors between the input string and input grammar, meaning that if the model can always attend to the relevant parts, it can classify strings correctly.

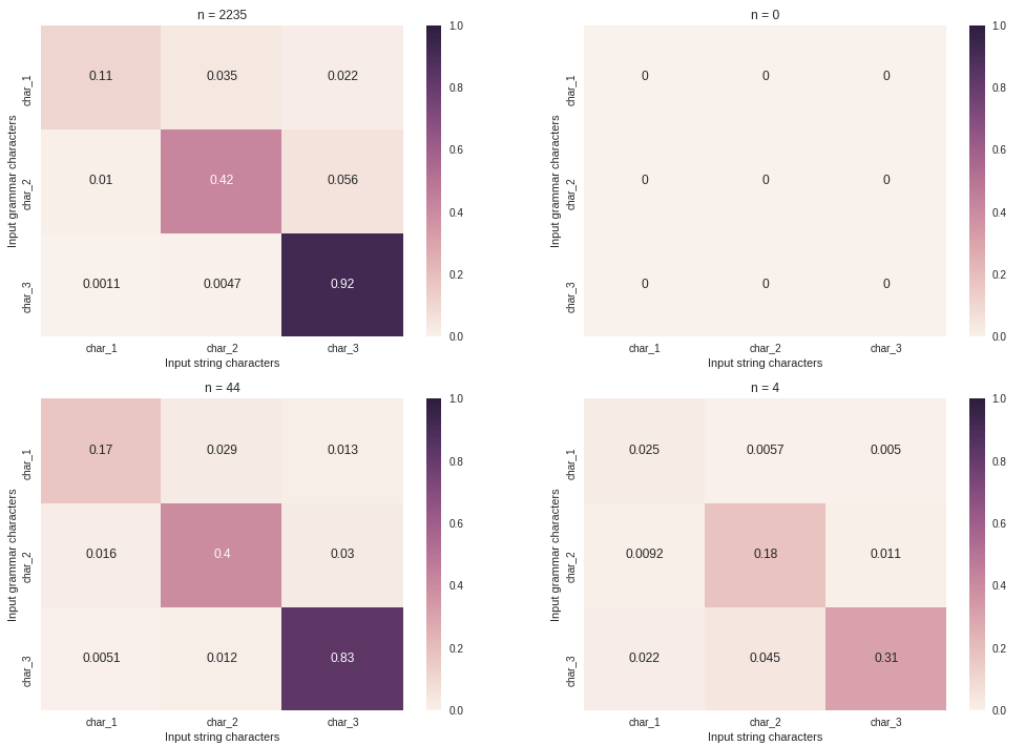

Figure 3.5: Average attention mask of **lstm+attn** model evaluated on 10000 grammar's validation set. Match refers to making a correct prediction at the last character of k-factor, whereas predict refers to making a correct prediction at the end of the sequence. n refers to how many examples belongs to this particular class.

3.7 Conclusions

Learning to follow instructions of subregular languages is quite a challenging task because the model needs to be able to match k-factors occurring in the input string to the k-factors in the grammar. The task becomes more challenging if the model needs to predict a novel grammars and k-factors, and even more so if they were generated from a different distribution (e.g. grammar that are of different lengths). Our experiments demonstrate that attention mechanism is a crucial part to enable the model identify k-factors in the input grammar. Our average attention mask on the validation set shows that a solution to this problem exists if it can solve two problems. First, the model needs to learn how to attend to the relevant parts of the input grammar. In other words, the model needs to attend to the relevant k-factor in the grammar while processing a substring in the input string that is equal to the relevant k-factor. This helps the model predict correctly at the last character of the k-factor. Second, if the model can predict correctly at the last character of the substring that is equal to the relevant k-factor while assigning high probability mask to the relevant k-factor in the input string, the model needs to be able to propagate the signal to the end of the sequence. Perhaps an interesting direction for future work would be to explore an attention augmented architecture which can capture useful bias to match k-factors, i.e., forcing the attention mechanism to attend to the relevant parts of the grammar. We also experimented with pondering mechanism but it does not seem to be of much help.

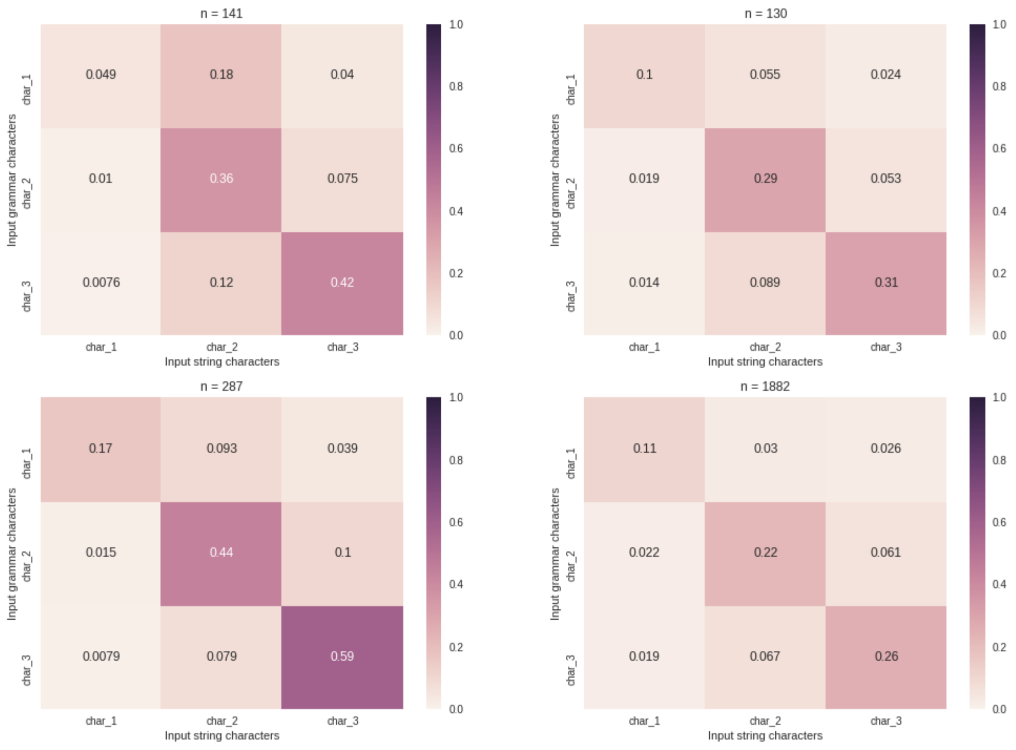

Figure 3.6: Average attention mask of **lstm+attn** model evaluated on 10000 grammar's test set. Match refers to making a correct prediction at the last character of k-factor, whereas predict refers to making a correct prediction at the end of the sequence. n refers to how many examples belongs to this particular class.

Chapter 4

SHRDLURN

In this chapter we present our work on SHRDLURN task. As stated in Section 1.1, one of the motivations of experimenting with SHRDLURN task is to alleviate the problem of using hand-coded rules to interpret instructions that comes in a natural language utterance. SHRDLURN is a task that features interactions of varied language utterances with only a few examples. In order to learn from small data without using human interventions, a model needs to automatically learn the ability to generalize beyond the training data that it sees. The model needs to resort to some assumptions about the tasks being considered. These assumptions that the model uses to predict unseen cases is known as inductive biases. Using a hand-coded rules in a data-scarce setting is an effective method to incorporate these inductive biases, but it requires human technical expertise to construct these rules. It is desirable in the future to have independent machines that can learn directly from data without any human interventions. We aim to allow learning agents to learn strong inductive biases directly from input-output examples, hoping that it can then generalize to unseen situations.

4.1 Introduction

SHRDLURN is a language game introduced by [Wang et al., 2016] which aims to explore the purpose-orientedness of language in a learning setting. The objective of this game is to transform a starting state in the form of colored blocks piles into the desired state using an utterance provided by the user of the game, which in this case is a human. The user types in an utterance and the learning agent in the environment tries to interpret the utterance and perform the corresponding manipulation on the blocks. The learning agent initially knows nothing about the language of the user, but through the signal provided by the user, it learns the user's language while making progress toward the game goal. The model in the original game works by producing a ranked list of possible manipulation on the blocks according to its current parameters. The user scrolls through the list and chooses the correct one, providing a signal to the learning agent. For the learning agent to be successful, it has to learn the user's language quickly within a reasonable amount of turn, so that the user can accomplish the goal more efficiently. Conversely, the user must also be consistent with their language in order to make the learning agent learn more efficiently. See Figure 4.1 for an illustration of SHRDLURN task.

4.2 Related Work

Our work is connected to a grounded language learning system where natural language is used in certain situations to achieve some goal. [Winograd, 1972] was one of the first pioneers in creating systems to learn to follow instructions, in which he developed a rule-based system for this endeavor. [Shimizu and Haas, 2009, Chen and Mooney, 2011, Artzi and Zettlemoyer, 2013, Vogel and Jurafsky, 2010, Andreas and Klein, 2015] are examples of work that have promoted statistical learning approaches towards the mentioned case. They all have on thing in common, namely assuming that all speakers use the same language. Hence, a learning agent can be trained on a set of dialogs of some of the speakers and it is tested on another user.

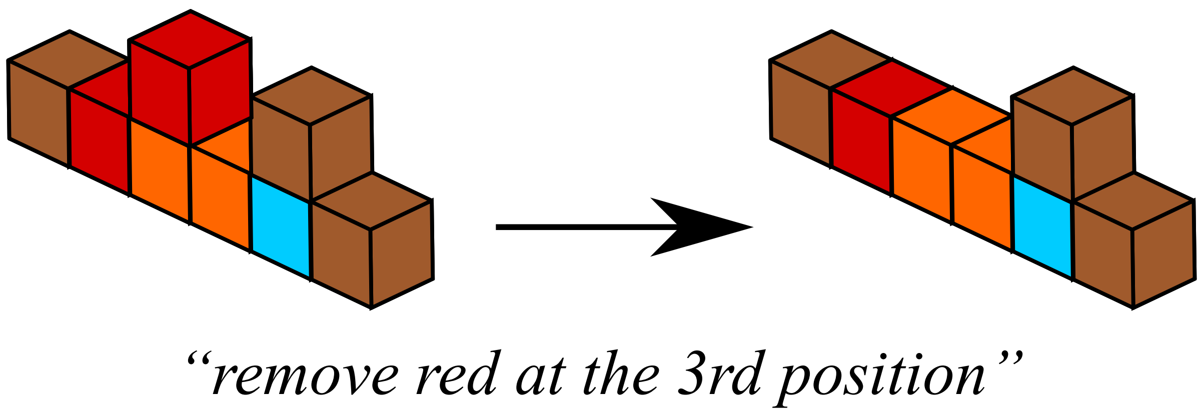

Figure 4.1: Illustration of the SHRDLURN task of [Wang et al., 2016]. The user types in an utterance and the learning agent performs the corresponding manipulation on the blocks configuration.

[Wang et al., 2016] instead took a different approach for this particular case. They introduced a block manipulation task in which a learning agent needs to learn to follow natural language instructions produced by human users. After the learning agent performs the manipulation, the user provides it with a learning signal. What makes this work different from other previous work is that every user can speak in their own language, natural or unnatural, and the learning agent has to adapt to each language of the speaker, understanding the intent and perform manipulation on the task. In their paper, the proposed original system is composed of a set of hand-coded functions that can be executed on the program to manipulate the state of the colored block piles and a log-linear learning agent that learns to map features derived from language instructions to expressions in this hand-coded functions. In this work, we aim to learn the mechanics of the game end-to-end and directly by providing input-output examples, providing no hand-coded functions.

[Lake et al., 2011, Trueswell et al., 2013, Herbelot and Baroni, 2017] are another examples of studies that have close relations to our research. The main goal of their research is to acquire a new concept from a single example of its usage in context. This concept goes by the name of fast mapping. Although the purpose of learning new concepts is not included in our objective explicitly, we do want to learn from few examples to draw an analogy among new terms and previously acquired concepts of terms.

This research can be also considered as an example of the transfer learning paradigm [Pan and Yang, 2010], which has been successful in both linguistic [Mikolov et al., 2013, Peters et al., 2018] and visual processing [Oquab et al., 2014]. However, instead of transferring knowledge from one task to another, we are transferring representation between artificial and natural data.

4.3 Method

There are at least two skills that a learning agent must master in order to learn to follow instructions on SHRDLURN task. First, the learning agent needs to learn the intent of the language of the speaker. Second, the learning agent needs to be able to perform an operation on the domain without trying actions that a human would probably never ask for. We also do not want the learning agent to perform an operation that violates the domain's constraint. In interactive settings like SHRDLURN where data is scarce, training deep learning models alone are not going to work. Hence, we propose a two-step training regime to allow a learning agent to automatically acquire the aforementioned skills directly from data and adapt to the language of the speaker with small data. First, we train a neural network model in an "offline" fashion on an artificially constructed dataset which resembles the target task. This allows the model to learn about the domain and mechanics of the task. Second, we train the model in an "online" fashion with the examples that each user provides. This phase enables the model to independently adapt to the language of a particular human user which can be very different from the language seen during the offline phase.

```
    S     →    VERB COLOR at POS tile
  VERB    →         add | remove
  COLOR   →   red | cyan | brown | orange
   POS    →      1st | 2nd | 3rd | 4th |
                5th | 6th | even | odd |
             leftmost | rightmost | every
```

Figure 4.2: Grammar of our artificially generated language

4.4 Dataset

The dataset in the SHRDLURN task comes in the form of (1) initial state of a group of colored blocks' configuration, (2) natural or unnatural language instructions which are provided by the user, and (3) resulting blocks' configuration that is produced by applying manipulation on the initial blocks' configuration that comply with the given language instruction. The dataset for this task constitutes a list of triples (initial configuration, instruction, target configuration) that describe these three components.

Offline training dataset For the offline phase, we generate 88 distinct natural language instructions following the grammar in Figure 4.2. This particular grammar is constructed with a limited vocabulary size but enough variation to capture the simplest possible actions in the game. For the initial block configurations, they are comprised of 6 piles containing a maximum of 3 colored blocks each. In other words, each piles can either be empty or contain 1, 2, or 3 colored blocks. To create the initial block configuration, we randomly sample a maximum of 3 colored blocks from a total 85 configurations [1] for each of 6 piles and combine them together. The combined 6 piles are then encoded as a string delimited by a special symbol `#`. Each piles contains a sequence of color tokens or a special `empty` symbol. The resulting block configuration is computed using a rule-based program which we develop to mimic the SHRDLURN game. The rule-based program takes in an input of randomly sampled initial block configuration and randomly sampled language utterances, and produce the resulting block configuration. If the sampled initial block and the language utterances are not compatible with each other, the sampling procedure is repeated. An example of our generated data is depicted in Figure 4.3. We show three columns rather than six for conciseness.

Dataset split To evaluate the models in a challenging compositional setting [Lake and Baroni, 2018], we generated the datasets that differ in how the data is split between train, validation, and test, as in Section 3.3 in the subregular languages experiment. Rather than producing a random split of the data, we split all the 88 possible utterances that can be generated from our grammar into 66 utterances for training, 11 for validation and 11 for testing. For the blocks configuration, we split all possible 85 valid combinations column of blocks into 69 combinations for training, 8 for validation and 8 for testing. Then we sample from the relevant block set 6 times to get 6 piles of colored blocks and combine these piles together to make the input block configurations. We also sample an utterance from the relevant utterances set and pair it with the input block configuration. Then we check if the two are compatible and comply with the rules of the game. If it is compatible then we generate the resulting block configuration, otherwise we repeat the sampling procedure. In this way, we generated 42000 instances for training, 4000 for validation and 4000 for testing.

Online training dataset For the online phase, we use pre-recorded user sessions from [Wang et al., 2016]. This dataset is exactly in the same form as in Figure 4.3, but it can contains an arbitrary number of piles.

[1]Since there are 4 different colors, then $4^0 + 4^1 + 4^2 + 4^3 = 85$. 4^0 being empty, 4^1 being only one block exist, etc.

Instruction	`remove red at 3rd tile`
Initial Config.	`BROWN X X # RED X X # ORANGE RED X`
Target Config.	`BROWN X X # RED X X # ORANGE X X`

Figure 4.3: Example of an entry in our dataset.

4.5 Models

Offline models To model the task we use an encoder-decoder (seq2seq) [Sutskever et al., 2014] architecture as explained in Section 2.4. Formally, the seq2seq architecture can be defined as follows: The encoder reads the natural language utterance $\mathbf{w} = w_1, \ldots, w_m$, and transforms it into a sequence of feature vectors $\mathbf{h}_1, \ldots, \mathbf{h}_m$, where w_i and h_i are the word and the vector representation of a word at index i in the language utterance sequence, respectively.

$$\mathbf{h} = \text{encoder}(\mathbf{w}) \tag{4.1}$$

The decoder takes an input sequence of the initial block configurations $\mathbf{x} = x_1, \ldots, x_n$ and produces a new sequence of the resulting block configuration $\hat{\mathbf{y}} = \hat{y}_1, \ldots, \hat{y}_n$, where x_i and $\hat{y}_i$ are the initial block components and the resulting block components at index i in the block configuration sequence, respectively. The initial and resulting block configuration will always have the same length.

$$\hat{\mathbf{y}} = \text{decoder}(\mathbf{x}|\mathbf{h}) \tag{4.2}$$

To pass information from the encoder to the decoder, we equip the decoder with an attention mechanism as explained in Section 2.5 [Bahdanau et al., 2014, Luong et al., 2015]. This allows the decoder, at every timestep, to "attend" to the relevant parts of the input sequence and calculate a weighted combination of all the encoder hidden states to produce an output. We train the system to match the target block configuration $\mathbf{y} = y_1, \ldots, y_n$ (represented as 1-hot vectors) using a cross-entropy loss:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} y_i \log \hat{y}_i \tag{4.3}$$

Both the encoder and decoder are sequential modules. This means that they take a sequence of inputs and produce a representation of the inputs. The input representation is then processed into a sequence of outputs which can be used to train the model for any given task. We experiment with two state-of-the-art sequence models: A standard recurrent LSTM (Section 2.3) [Hochreiter and Schmidhuber, 1997] and a convolutional sequence model (Section 2.7) [Gehring et al., 2016, Gehring et al., 2017], which has been shown to outperform the former on a range of different tasks [Bai et al., 2018]. For the convolutional model we use kernel size $k = 3$ and zero padding to make the size of the output match the size of the input sequence. We speculate that block configuration does not have any strong sequential cues, in the sense that processing a block configuration in one pile does not help in informing how to process block in another pile. Because of this invariant structure, we expect that the convolutional model to be more suitable to process the block configuration than the recurrent model. As a simple baseline, we also experimented with a bag-of-words encoder (Section 2.2) which represents sentence as an average of word embeddings and n-gram embeddings. We explore all possible combinations of architectures for the encoder and decoder components, resulting in 5 configuration in total.

Online models Once the model has been trained on an artificial dataset, we test the models' capabilities to adapt to the language of a new, real user, who does not know anything about how the model was trained. Additionally, the language of each particular user can be very different as users are encouraged to communicate using their own language. In interactive settings, real users typically do not provide many examples for the model to learn from, hence the model has to adapt to follow instructions with only a handful examples. One of the first challenges it will encounter when adapting to the language of a new speaker is to quickly master the meaning of a new and unseen words. As mentioned in Section 4.2, the challenge of inferring the meaning of a word from a single exposure goes by the name of fast-mapping [Lake et al., 2011, Trueswell et al., 2013].

For this experiment, we adopt a fast-mapping method proposed by [Herbelot and Baroni, 2017]. The original paper proposes a method to learn the embeddings of a new word with gradient descent by freezing all the other network weights except the embeddings of the new words. We further develop it by experimenting with three different variations of this method. First, we try to learn only *new* word embeddings as they did. Second, we also try to learn the full encoder layer, freezing the decoder network, which effectively allows words seen during offline training to shift their meaning and adjust how each word are processed sequentially. Third, we test what happens if we adjust the whole encoder-decoder network, which allows the weights that process the blocks to be adjusted. In the latter two cases, we incorporate L_2 regularization over the embeddings and the model weights to avoid overfitting.

Training algorithms Recall that human users interact with the system by giving instructions to the learning agent on the system in their own language. After the learning agent performs the manipulation, the user provides immediate feedback on what was the intended target block configuration. In our experiments, we simulate the online interactions by using the pre-recorded sessions data from [Wang et al., 2016]. In our system, for every new incoming example that the model observes, the new incoming example is added to a buffer B. After that, the model is trained with a fixed number of gradient descent steps using examples randomly drawn from a subset B_{TR} of this buffer. B_{TR} is a subset of buffer B which stores the training examples from an incoming inputs.

Training in an online setting from just few examples can lead the model to land in local minima due to the noisy gradients which give rise to fluctuations occuring on the loss surface [Goodfellow et al., 2016]. In order to reduce the impact of local minima, we train k different copies (rather than training a single model) each with different initialization of embeddings for new words. Among the k different models, for every incoming example, we pick one model to predict an incoming example according to our model selection strategies using evidence B_{SE}. B_{SE} is a subset of buffer B which stores the examples for model selection strategies. We experiment with two model selection strategies:

- **greedy**, where we pick the model with the lowest loss computed over the full training buffer examples ($B_{\text{SE}} = B_{\text{TR}} = B$).
- **1-out**, where we save the last example for validation and pick the model that has the lowest loss on that example ($B_{\text{SE}} = B[\texttt{LAST}]$, $B_{\text{TR}} = B[0 : \texttt{LAST} - 1]$).

Other than this method, there is a wealth of methods in the literature for model selection (see, e.g., [Claeskens et al., 2008]). We leave this exploration for future work. Algorithm 1 summarizes our approach.

4.6 Experiments

There are three main experiments for the SHRDLURN task. First is **(1)** to explore what is the best architectural choice for solving the SHRDLURN task on our artificial dataset dataset. Second, we run multiple controlled experiments to investigate the adaptation skills of our online learning system. In particular, we first test whether **(2)** the model is able to recover the original meaning of a word that has been replaced with a new arbitrary symbol – e.g. "*brown*" becomes "*braun*"– in an online training regime. Finally, we proceed to **(3)** learning from real human utterances using the dataset collected by [Wang et al., 2016].

4.6.1 Offline training

We explore all possible combinations of encoder and decoder models: LSTM encoder and LSTM decoder (**seq2seq**), LSTM encoder and convolutional decoder (**seq2conv**), convolutional encoder and LSTM decoder (**conv2seq**), and both convolutional encoder and decoder (**conv2conv**). Furthermore, we explore a bag of words encoder with an LSTM decoder (**bow2seq**) as our baseline. We train 5 models with our generated dataset and use the best performing model for the following experiments.

Algorithm 1 Online Training

```
1: Initialize models $m_1, \dots, m_k$
2: Let $B$ be an empty training buffer
3: for $t$ = 1,2,...,$T$ do
4:     Observe the input $(\mathbf{w}_t, \mathbf{x}_t)$
5:     SELECT best model $m_i$ using data $B_{\text{SE}}$
6:     Predict $\hat{\mathbf{y}}_t = m_i(\mathbf{w}_t, \mathbf{x}_t)$
7:     Observe feedback $\mathbf{y}_t$.
8:     Add $(\mathbf{w}_t, \mathbf{x}_t, \mathbf{y}_t)$ to $B$
9:     TRAIN $m_1, \dots, m_k$ on data $B_{\text{TR}}$
10: procedure SELECT($m_1, \dots, m_k$, $B_{\text{SE}}$)
11:     Let $C_i \leftarrow \sum_{(\mathbf{w},\mathbf{x},\mathbf{y}) \in B_{\text{SE}}} \mathcal{L}(\mathbf{y}, m_i(\mathbf{w}, \mathbf{x}))$
        return $m_i$ that minimizes $C_i$
12: procedure TRAIN($m_1, \dots, m_k$, $B_{\text{TR}}$)
13:     for $i = 1, \dots, k$, $s = 1, \dots, S$ do
14:         Draw $\mathbf{w}, \mathbf{x}, \mathbf{y} \sim B_{\text{TR}}$
15:         Compute $\nabla\mathcal{L}(\mathbf{y}, m_i(\mathbf{w}, \mathbf{x}))$
16:         Update $m_i$
```

Model	Val. Accuracy	Test Accuracy
seq2seq	78	79
seq2conv	**99**	**100**
conv2seq	73	67
conv2conv	64	74
bow2seq	57	63

Table 4.1: Model's accuracies evaluated on block configurations and utterances that were completely unseen during offline training. Results are expressed in percentages.

We conduct a hyperparameter search for all these models, exploring the number of layers (1 or 2 for LSTMs, 4 or 5 for the convolutional network), the size of the hidden layer (32, 64, 128, 256) and dropout rate (0, 0.2, 0.5). For each model, we pick the hyperparameters that maximized accuracy on our validation set and report validation and test accuracy in Table 4.1. As can be seen from the table, **seq2conv** achieves the best accuracy on our artificial dataset by a large margin. It performs almost perfectly on this non-overlapping validation and test set featuring only unseen utterances and block configurations. This result validates our hypothesis that the convolutional sequence model is the most suitable model for the decoder to process the block configuration.

4.6.2 Online training - recovering corrupted words

Once the offline training model is at hand, we test if the model can adapt quickly to controlled variations in the language generated from our grammar. In particular, we simulate a learning situation in which the model is presented a simulated user producing utterances drawn from the same grammar as the offline training. However, some words of the utterances are corrupted with unknown symbols so that the model cannot recognize them anymore. The model has to make a guess about the meaning of the corrupted word and progressively learn its meaning during online training. The model is also evaluated on whether it can recover the meaning of these words during online training.

For this experiment, we combine the validation and test set of our dataset, together containing 22 distinct utterances, which serve as the utterances that are corrupted for the online training. We combine these two sets to make sure that the corrupted utterances presented during online training were completely unseen during the offline training phase. We then equally split the vocabulary of our grammar into two disjoint sets of words and corrupt it. The first set of word is used for hyperparameter search (**hyperparameter corrupt set**) and the other one for testing (**test corrupt set**). For validation, we take one verb ("add"), 2 colors ("orange" and "red"), and 4 positions ("1st", "3rd; ; "5th"

Original utterance	**Corrupted utterance**
remove red block	remove *roze* block
add brown block	*blabla* brown block
add orange block 1st	*blabla* oranje block *1*

Figure 4.4: Corrupted dataset example for hyperparameter search. Italic words represent corrupted words.

Combination of words	**Sampled original utterance**	**Sampled corrupted utterance**	**Criteria**
brown	remove brown block	remove *braun* block	1 word corrupted
brown, remove	remove brown block	*blabla braun* block	2 words corrupted
brown, remove, 1st	remove the brown block at 1st piles	*rmv* the *braun* block at *fst* piles	3 words corrupted
all words in **test corrupt set**	remove the brown block at 1st piles	*rmv de braun blax edd fst slp*	all words corrupted

Figure 4.5: Corrupted dataset example for online training-recovering corrupted words evaluation. Italic words represent corrupted words.

and "even"). We use the remaining half of the vocabulary for testing.

Hyperparameter search We then take a set of distinct utterances from the combined validation and test set which contain words from the **hyperparameter corrupt set**, resulting in 15 distinct utterances. We corrupt each occurrence of the words by replacing them with a new token/symbol. We consistently keep the same new token for each occurrence of the word. An example is presented in Figure 4.4. For each of these 15 distinct utterances, we extract 3 block configurations to pair them with, resulting in simulated user sessions with 45 instruction examples.

We use this simulated user session for a hyperparameter search of the online training phase. In particular, we vary the optimization algorithm to use (Adam or SGD), the number of training steps (100, 200 or 500), the regularization weight (0, 10^{-2}, 10^{-3}, 10^{-4}), the learning rate (10^{-1}, 10^{-2}, 10^{-3}), and the model selection strategy (greedy or 1-out), while keeping the number of model parameters that are trained in parallel fixed to $k = 7$. We report the best hyperparameters for each model in the Appendix.

For this particular experiment, we experiment with three different variation of which weights we want the model to learn: (1) learning only the embeddings for the new words, leaving all the remaining weights frozen (**Embeddings**), (2) learning the encoder with a randomly initialized weights while keeping the decoder fixed (**Encoder**), (3) learning a fully randomly initialized model (**Encoder + Decoder**). We also evaluate the impact of having multiple ($k = 7$) concurrently trained model parameters by comparing it with just having a single set of parameters trained (**Embeddings** $k = 1$).

Evaluation To evaluate the models, we create another simulated user session consisting of utterances drawn from the **test corrupt set**. For every combination of words in the **test corrupt set**, we sample 15 distinct utterances containing these combinations of words and pair each of them with 3 block configurations, resulting in another simulated user sessions with 45 distinct utterances. We group every simulated user sessions containing these combinations of words from the **test corrupt set** into 4 different criteria for testing phase, namely where we corrupt: one single word, two words, three words, and all words. See Figure 4.5 for details. There are 7 simulated user sessions for 1 word, 17 for 2 words, 10 for 3 words and 1 for all words criteria.

To evaluate the performance of the model on these corrupted words dataset, we use online accuracy as our as our figure of merit, which is computed as:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{I}[\hat{\mathbf{y}}_t == \mathbf{y}_t] \tag{4.4}$$

where T is the length of the game. We report the results in Table 4.2.

First, we can see that the **Embedding** models, which only learn the embedding of unseen words perform best. Perhaps this is not surprising, as the other two model used randomly initialized weights instead of the weights acquired from the offline learning phase. Notably, it can reach 73% accuracy even when all words have been corrupted (for reference, the model of [Wang et al., 2016] obtains 55% on the same task). However, in the single corrupted word condition, re-learning the full encoder seems to be performing better than learning only the embeddings of unseen words. In addition, we can also observe that training multiple models with different parameters initialization does help in alleviating the local minima problem. We compare the **Embeddings** models trained with $k = 7$ models with different initialization and $k = 1$ model. We observe that the former is consistently better. We conclude that the model can recover the meaning of corrupted words quite well, albeit the distribution in which the utterance was drawn is comparable to the one seen during training.

Re-training	**1 word**	**2 words**	**3 words**	**all words**
Enc + Dec	43.3	35.5	36.1	36.7
Encoder	**93.8**	85.9	82.4	55.5
Embeddings	90.9	**88.1**	**86.1**	**73.3**
Emb. $k = 1$	86.1	84.3	81.8	55

Table 4.2: Online accuracies (in percentages) for the word recovery task averaged over 7 simulated user sessions for 1 word, 17 for 2 words, 10 for 3 words and 1 for the all words criteria.

4.6.3 Online training - adapting to human speakers

In the previous controlled experiments, we have established the ability of our model to adapt to situations in which it encounters corrupted words and it has to recover its original meaning. Here we move to a more challenging settings in which the model needs to adapt to real human speakers. In this setting, the language of a new speaker can be very different from the one seen during the offline learning phase, both in surface form and in its underlying semantics.

For this experiment, we use the dataset made available by [Wang et al., 2016]. The dataset is collected from real users (turkers) playing the SHRDLURN game which contains their log-linear/symbolic model that gets trained through interactions with the users. The dataset contains 100 games with nearly 10k instruction examples in total.

Hyperparameter search For this experiment, we do our hyperparameters on user sessions that may provide a stronger learning signal. We hypothesize that it is better to focus our hyperparameter exploration on cases where we had a higher chance. To this end, we manually select three games corresponding to unique users session in this dataset. We visually inspect games/users' sessions that were in the top 5% in terms of Wang et al.'s model performance. Out of these games, we choose 3 users' sessions in which the language of the users looks reasonably close to our artificial dataset, even if the words are completely different. The top 3 users' sessions then are used to tune the online learning hyperparameters and the remaining 97 users' sessions are left for testing. For testing, we evaluate the online accuracy as in Equation 4.4. We report the best hyperparameters for each model in the supplementary materials.

		Adapt					
		(1) Embeddings		(2) Encoder		(3) Encoder+Decoder	
		acc.	r	acc.	r	acc.	r
Reuse	(c) Nothing (Random)	-	-	-	-	13.5	0.58
	(b) Decoder (Random Encoder)	-	-	23	0.83	21	0.7
	(a) Encoder + Decoder	18.2	0.74	22.6	0.84	21.3	0.72

Table 4.3: For each (valid) combination of set of weights to re-use and weights to adapt online, we report average online accuracy on [Wang et al., 2016] dataset and correlation between online accuracies obtained by our model and those reported by the authors.

.

We explore 6 different variants of adapting our model to new speakers that can be classified according to two factors of variation. The first factor is which subset of weights from the offline training phase we use as the initial weights of the system on the online training phase. To assess the importance of the first factor, we vary which set of pre-trained weights are used as an initial weights during the online training phase: (a) All the weights in the encoder and all the weights of the decoder; (b) only the weights of the decoder while randomly initializing the encoder; or (c) randomly initializing all weights of the encoder and decoder, i.e., no weights are re-used from the offline learning phase. The second factor is which subset of weights we train during the online training, leaving all the rest frozen: (1) Only the word embeddings[2], (2) the full weights of the encoder or (3) the full network (both encoder and decoder). Combining the two factors result in 9 possible combinations. Among the 9 possible combinations, we leave out 3 combinations which result in random components not being updated. (c-1), (c-2), and (b-3) are 3 combinations which result in a model with randomly initialized weights that are never trained. Leaving them out, we are left with 6 possible combinations.

For each of the remaining 6 valid training combinations, we run an independent hyperparameter search choosing from the same hyperparameter configuration pool as in the word recovery task (Section 4.6.2). We evaluate the average online accuracy for each of the configurations that the models achieved on the three validation games. We report the best hyperparameters for each model in the Appendix.

Evaluation We evaluate each of the model variants on the remaining 97 games in the test set and calculate the average online accuracy acquired by each of the model variants. We hypothesize that the quality of the inductive bias strongly affects the model's performance on adapting to unseen utterances. The inductive bias can be acquired automatically from data through offline learning phase (our model) or it can be programmed and incorporated into the model (Wang et al.'s model). We consider the inductive biases that [Wang et al., 2016] incorporated into their model as a good proxy for what is a useful bias to learn about this task. Hence, the more our model behaves similarly to their system, the more likely it is that it is encoding similar inductive biases. Because of this reason, we also measure the pearson correlation between our model's online accuracy on every single users' sessions and that obtained by Wang et al.'s system. The correlation serves as a way to measure the similarity of both systems in behavior. The results of these experiments are displayed in Table 4.3.

Analysis According to the result, first, it is confirmed that the models which are given the knowledge during the offline training (rows a and b) perform better than the randomly initialized model (3-c).

Second, the randomly initialized encoder unexpectedly showed a slightly better performance than the pre-trained model[3]. This result suggest that the model can ignore the knowledge of our artificial grammar embedded within the encoder and it is better off learning the language from scratch, even from few examples. Finally, we observe that the models that perform the best are those in column (2) which adapt the encoder weights and freeze the decoder weights. They are also those that correlate the most with the Wang et al.'s symbolic model. And we can also see that models performance in column (2) seems to be perfectly aligned with correlation with the original system performance[4]. This indicates that the symbolic model indeed contains useful inductive biases, validating our hypothesis which is: the better the models are at capturing these useful inductive biases, the better they perform in this task. Overall, the result also indicates that all the useful inductive bias in the system is contained in the decoder weights because the model can perform pretty well without the encoder's initialization.

For reference the symbolic model attains an average online accuracy of 0.33 in this dataset, leaving still room for improvement in this task.

[2]Differently from the subregular experiment, here we report adapting the full embedding layer, which performed better than just adapting the embeddings of an unseen words

[3]Recall that the encoder is the component that reads and interprets the user language, while the decoder processes the block configurations conditioned on the information extracted by the encoder.

[4]As a matter of fact the 7 entries of online accuracy and pearson r are themselves correlated with $r = 0.99$, which is highly significant even for these few data points

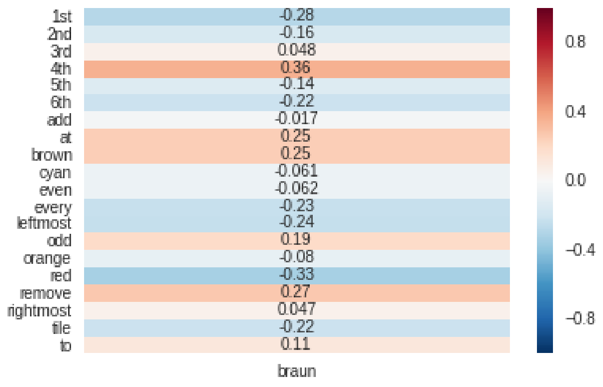

Figure 4.6: Cosine similarities of the newly learned word embedding for the corrupted version of the word "brown" with the rest of the vocabulary.

4.7 Discussion

4.7.1 Word recovery

In order to deepen our understanding of what our model learns, we perform a further analysis on the results of the word recovery task (4.6.2). Specifically, we want to see whether the learned word on the word recovery task have similar meaning to the original word. For this experiment, we use a simulated user session [5] containing three corrupted words, namely "brown", "remove" and "every". For analysis, we measure the cosine similarity of the word "brown" and its corrupted version which we call "braun". The analysis is performed to check how close they are.

Observation In this particular user session, the model encounters this word for the first time on an utterance "rmv braun at evr tile". It fails to predict correctly its meaning on a first encounter yet. However, afterwards it resolves correctly every new occurrence of the word: For example on "rmv braun at 5th tile" or "add braun to 4th tile". Figure 4.6 shows the similarity of the word "braun" with every other word in the vocabulary. Looking at the figure, while the model correctly identified that this word bears some similarity with its original meaning (word "brown"), it also think that this word bears some similarity with "remove" and "4th". We link this result with the observation made by [Lake and Baroni, 2018] who show that neural network systems struggle at capturing systematic compositionality. We also explore a mechanism that allows the model to re-use already known word embeddings rather than learning them from scratch to alleviate the exposed problem. Formally, we can define the proposed method as follows: given E as the original embedding matrix of a known words (offline training) and e_w as a new randomly initialized embedding (online training), we construct a "fast" embedding $\hat{e}_w$ that is forwarded to upper layers as:

$$\hat{e}_w = \lambda_w e_w + (1 - \lambda_w) \sum_j \alpha_j E_j \tag{4.5}$$

$$\alpha_j = \text{softmax}(e_w^\top E) \tag{4.6}$$

where α_j is computed as $\text{softmax}(e_w^\top E)$ and λ_w is an extra parameter. We tried this method but it didn't seem to improve the performance of the model.

4.7.2 Human data

As it is shown in Section 4.6.3, the performance of our system strongly correlates with the symbolic system of Wang et al (2016). However, this correlation is not optimal, and thus, there are games in which our system performs comparatively better or worse on average. We look for examples for such games in the dataset. Figure 4.7 shows a particular case in which our system fails to learn. Notably,

[5]See Section 4.6.2 and particularly Figure 4.5 for explanation on how this user session is generated

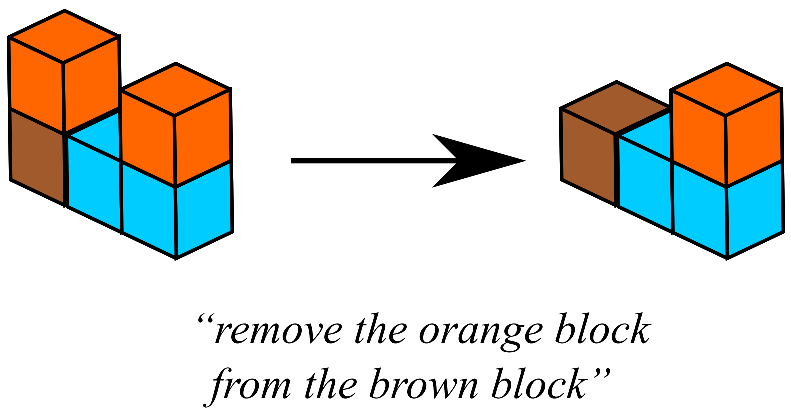

Figure 4.7: Example of failing case for our system. During offline training it had not seen other colored blocks to be used as referring expressions for locations.

the cause of the model's failure is because it is using other blocks as a reference for expressions to indicate positions. The mentioned situation is a mechanism that the model had not seen during offline training, and thus the system did not obtain the skill with the possibility to quickly assign a meaning to it. On the other hand, those cases in which our system obtains better performance are either caused by a usage of a language that is either very similar to our offline training dataset or failure of the players who often pick the same target configuration as input, producing easy and learnable identity transformation of the blocks.

In conclusion, we observe that our offline training regime allowed the model to acquire some useful inductive bias for this task. However, if users use some strategies that are not featured during the offline training phase, the model is not able to capture them.

4.8 Conclusions

In an interactive setting, learning to follow human instructions is a challenging task because humans typically do not provide a lot of data. In order to learn from small data, strong inductive biases are necessary. Previous work has relied on training a model to map language instructions to manually built hand-coded functions which can be executed on the system. Here we aim to allow neural network to acquire this knowledge automatically from data through a two-step training regime: (1) An offline learning phase where the networks are presented with artificial dataset and it has to learn about the general structure of the task and (2) an online learning phase where the networks need to independently adapt to the language of a new specific speaker. Our first controlled experiments on recovering corrupted words demonstrate that the networks are able to adapt to unseen vocabulary very efficiently, albeit that the utterances presented in this experiment is very similar to the one it has been trained on. Our second experiments on adapting to human speakers show that our network can still make use of the inductive bias that has been automatically learned during the offline learning phase, even if the language of the speakers deviates significantly from our artificial language. For the second experiments, a randomly initialized encoder performs equally well or better than the pre-trained encoder. This indicates that the knowledge that the network learns during offline training phase is more specific to the blocks manipulation rather than discovering language universals. Perhaps this is not too surprising given the minimalism of our artificial grammar.

To conclude, we have shown that our system can extract useful inductive bias to generalize to follow linguistic instructions of new speakers using an unconstrained language.

Chapter 5

Conclusions and future work

In this chapter, we conclude our thesis by highlighting the contributions of this research. We relate them with the motivations and objectives mentioned in Chapter 1.

5.1 Conclusions

As stated in Section 1.1, 3.1, and 4.1, there are several reasons that motivate this thesis. First, we argue that in order to have a useful general AI system, it needs to be flexible in learning continuous flow of tasks effectively from small exposure of data as humans do. Second, it also needs to be able to generalize to unseen situations by exploiting prior knowledge which is learned automatically through previous interactions. The objectives of this thesis is formulated around these motivations, namely the exploration of architecture and techniques that can help learning agents generalize to unseen situations. To this end, we worked with two task, namely subregular languages and SHRDLURN task. Based on the results and discussion in Chapter 3 and 4, we present the conclusions of this research.

On subregular languages task, we have found that attention mechanism helps LSTM to identify whether a k-factors exists in the input string. From the experiments, we concluded that although the model learns how to match k-factors between the input grammar and the input string on the validation set, it does not learn the right interpretation function, as shown in the model's overall performance on the test set. Furthermore, the analysis on the attention masks suggest that the right interpretation function can be acquired if the model can solve two problems. First, the model needs to learn how to attend to the relevant parts of the input grammar. In other words, the model needs to attend to the relevant k-factor in the grammar while processing a substring in the input string that is equal to the relevant k-factor. This helps the model predict correctly at the last character of the relevant k-factor in the input string. Second, if the model can predict correctly at the last character of the substring that is equal to the relevant k-factor while assigning high probability mask to the relevant k-factor in the input string, the model needs to be able to propagate the signal to the end of the sequence.

On SHRDLURN task, we have shown that our system can extract useful inductive biases to generalize to follow language instructions of a variety of new speakers. Our controlled experiments on recovering corrupted words indicate that when the model is exposed to a language that is very similar to the generated artificial language seen during offline training, it can adapt very efficiently with only few examples. Moreover, our experiment on human data shows that it can still make use of the inductive biases that has been automatically learned from the data, even if the language deviates significantly from the training data.

In summary, we have accomplished our thesis objectives. We explored some methods to interpret grammar's specifications on subregular languages task, presented a thorough analysis of the result, and found that our model can achieve better performance if we can make it attend to the correct part of the grammar. We also explored some methods to make neural networks learn from small data and generalize to unseen situations on the SHRDLURN task, and found that our system acquired useful inductive biases from small data to generalize to unseen situations.

5.2 Future work

There are several aspects of this thesis that can still be refined for future work. On subregular languages, it is interesting to explore methods to guide the attention mechanism to attend to the correct parts of the grammar. In light of recent research of [Hupkes et al., 2018], it might be interesting if future work can incorporate this approach and see how it impacts the results. As noted in Section 3.3, we did not experiment with dataset that is of longer length. It might also be interesting to see the generalization capability of a model trained on a dataset of fixed length and tested on longer examples. On SHRDLURN, an interesting direction to explore in the future is perhaps adopting meta-learning techniques [Finn et al., 2017, Ravi and Larochelle, 2017], where the network parameters are tuned having in mind that they should serve for adaptation.

Bibliography

[Andreas and Klein, 2015] Andreas, J. and Klein, D. (2015). Alignment-based compositional semantics for instruction following. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1165–1174.

[Artzi and Zettlemoyer, 2013] Artzi, Y. and Zettlemoyer, L. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association of Computational Linguistics*, 1:49–62.

[Avcu et al., 2017] Avcu, E., Shibata, C., and Heinz, J. (2017). Subregular complexity and deep learning.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

[Bai et al., 2018] Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv:1803.01271*.

[Baroni et al., 2017] Baroni, M., Joulin, A., Jabri, A., Kruszewski, G., Lazaridou, A., Simonic, K., and Mikolov, T. (2017). Commai: Evaluating the first steps towards a useful general AI. *CoRR*, abs/1701.08954.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166.

[Caudill, 1987] Caudill, M. (1987). Neural networks primer, part i. *AI Expert*, 2(12):46–52.

[Chen and Mooney, 2011] Chen, D. L. and Mooney, R. J. (2011). Learning to interpret natural language navigation instructions from observations. In *AAAI*, volume 2, pages 1–2.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. cite arxiv:1406.1078Comment: EMNLP 2014.

[Chomsky, 1957] Chomsky, N. (1957). *Syntactic Structures*. Mouton and Co., The Hague.

[Claeskens et al., 2008] Claeskens, G., Hjort, N. L., et al. (2008). Model selection and model averaging. *Cambridge Books*.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.

[Finn et al., 2017] Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135.

[Gehring et al., 2016] Gehring, J., Auli, M., Grangier, D., and Dauphin, Y. N. (2016). A convolutional encoder model for neural machine translation. *CoRR*, abs/1611.02344.

[Gehring et al., 2017] Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Graves, 2016] Graves, A. (2016). Adaptive computation time for recurrent neural networks. *CoRR*, abs/1603.08983.

[Herbelot and Baroni, 2017] Herbelot, A. and Baroni, M. (2017). High-risk learning: acquiring new word vectors from tiny data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 304–309.

[Hochreiter, 1991] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Hupkes et al., 2018] Hupkes, D., Singh, A., Korrel, K., Kruszewski, G., and Bruni, E. (2018). Learning compositionally through attentive guidance. *CoRR*, abs/1805.09657.

[Jäger and Rogers, 2012] Jäger, G. and Rogers, J. (2012). Formal language theory: refining the chomsky hierarchy. *Philos Trans R Soc Lond B Biol Sci*, pages 1956–1970.

[Joulin et al., 2017] Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2017). Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431. Association for Computational Linguistics.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Lake and Baroni, 2018] Lake, B. and Baroni, M. (2018). Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks.

[Lake et al., 2011] Lake, B., Salakhutdinov, R., Gross, J., and Tenenbaum, J. (2011). One shot learning of simple visual concepts. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 33.

[Lake et al., 2016] Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2016). Building machines that learn and think like people. *CoRR*, abs/1604.00289.

[Lecun and Bengio, 1995] Lecun, Y. and Bengio, Y. (1995). *Convolutional networks for images, speech, and time-series*. MIT Press.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551.

[Luong et al., 2015] Luong, M., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025.

[McNaughton and Papert, 1971] McNaughton, R. and Papert, S. (1971). *Counter-free automata*. M.I.T. Press research monographs. M.I.T. Press.

[Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc.

[Montague, 1970] Montague, R. (1970). Universal grammar. *Theoria*, 36(3):373–398.

[Oquab et al., 2014] Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1717–1724. IEEE.

[Pan and Yang, 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

[Pascanu et al., 2013] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA. PMLR.

[Peters et al., 2018] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of NAACL 2018*.

[Ravi and Larochelle, 2017] Ravi, S. and Larochelle, H. (2017). Optimization as a model for few-shot learning. In *Proceedings of the International Conference of Learning Representations(ICLR)*.

[Rogers et al., 2010] Rogers, J., Heinz, J., Bailey, G., Edlefsen, M., Visscher, M., Wellcome, D., and Wibel, S. (2010). On languages piecewise testable in the strict sense. In *Proceedings of the 10th and 11th Biennial Conference on The Mathematics of Language*, MOL'07/09, pages 255–265, Berlin, Heidelberg. Springer-Verlag.

[Shimizu and Haas, 2009] Shimizu, N. and Haas, A. R. (2009). Learning to follow navigational route instructions. In *IJCAI*, volume 9, pages 1488–1493.

[Simon, 1975] Simon, I. (1975). Piecewise testable events. In Brakhage, H., editor, *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 214–222, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Sutskever et al., 2014] Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

[Thomas, 1982] Thomas, W. (1982). Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25(3):360 – 376.

[Trueswell et al., 2013] Trueswell, J. C., Medina, T. N., Hafri, A., and Gleitman, L. R. (2013). Propose but verify: Fast mapping meets cross-situational word learning. *Cognitive psychology*, 66(1):126–156.

[Vogel and Jurafsky, 2010] Vogel, A. and Jurafsky, D. (2010). Learning to follow navigational directions. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 806–814. Association for Computational Linguistics.

[Waibel et al., 1989] Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339.

[Wang et al., 2016] Wang, S. I., Liang, P., and Manning, C. D. (2016). Learning language games through interaction. *arXiv preprint arXiv:1606.02447*.

[Winograd, 1972] Winograd, T. (1972). Understanding natural language. *Cognitive psychology*, 3(1):1–191.

Appendix

Hyperparameter details

Model	Grammar Size	LSTM Layers	Dropout Rate	Hidden Size
lstm	5	2	0.2	128
lstm	10	1	0.2	256
lstm	50	1	0.5	256
lstm	100	2	0.5	128
lstm	500	1	0.2	32
lstm	1000	1	0.2	128
lstm	5000	1	0.2	64
lstm	10000	2	0.5	64
lstm+attn	5	2	0.2	256
lstm+attn	10	1	0.5	128
lstm+attn	50	2	0.2	256
lstm+attn	100	2	0.2	256
lstm+attn	500	2	0.0	256
lstm+attn	1000	2	0.5	256
lstm+attn	5000	2	0.2	256
lstm+attn	10000	2	0.5	256
lstm+attn+act	5	2	0.0	64
lstm+attn+act	10	1	0.5	256
lstm+attn+act	50	2	0.0	128
lstm+attn+act	100	2	0.5	256
lstm+attn+act	500	2	0.5	256
lstm+attn+act	1000	2	0.2	256
lstm+attn+act	5000	2	0.2	256
lstm+attn+act	10000	2	0.5	256

Best hyperparameter on overlap dataset

Model	Grammar Size	LSTM Layers	Dropout Rate	Hidden Size
lstm	5	2	0.5	32
lstm	10	1	0.0	128
lstm	50	2	0.5	256
lstm	100	1	0.5	64
lstm	500	1	0.5	64
lstm	1000	1	0.2	64
lstm	5000	1	0.5	64
lstm	10000	1	0.5	64
lstm+attn	5	2	0.2	64
lstm+attn	10	1	0.0	256
lstm+attn	50	2	0.2	256
lstm+attn	100	2	0.5	256
lstm+attn	500	2	0.5	256
lstm+attn	1000	2	0.2	256
lstm+attn	5000	2	0.2	256
lstm+attn	10000	2	0.0	256
lstm+attn+act	5	2	0.5	32
lstm+attn+act	10	1	0.0	128
lstm+attn+act	50	2	0.2	256
lstm+attn+act	100	2	0.2	256
lstm+attn+act	500	2	0.5	256
lstm+attn+act	1000	2	0.5	256
lstm+attn+act	5000	1	0.5	128
lstm+attn+act	10000	1	0.5	128

Best hyperparameter on non-overlap dataset

Model	Hidden Size	LSTM Layers	Conv Layers	Dropout Rate
seq2seq	64	1	NA	0.2
seq2conv	64	1	5	0.5
conv2seq	128	2	5	0.5
conv2conv	128	NA	4	0.5
bow2seq	128	1	NA	0.5

Best hyperparameter offline training

Model	Optimizer	Learning Rate	Training Steps	Regularization Weight	Model Selection
Embeddings	Adam	1e-2	500	1e-3	1-out
Encoder	Adam	1e-3	500	1e-3	1-out
Enc + Dec	Adam	1e-4	500	1e-3	1-out
Emb. $k = 1$	Adam	1e-2	500	1e-3	1-out

Best hyperparameter online training - word recovery task

Reuse-Adapt	Optimizer	Learning Rate	Training Steps	Regularization Weight	Model Selection
Enc + Dec - Emb	Adam	1e-2	200	1e-2	1-out
Enc + Dec - Enc	SGD	1e-2	200	1e-4	1-out
Enc + Dec - Enc + Dec	SGD	1e-2	200	0	1-out
Dec - Enc	SGD	1e-2	500	0	1-out
Dec - Enc + Dec	SGD	1e-2	100	1e-2	1-out
Nothing - Enc + Dec	SGD	1e-2	200	1e-3	1-out

Best hyperparameter online training - human data