Exploiting Target Syntax with Structured Decoding

by

Yilong (Jetic) Gū

B.Sc., Simon Fraser University, 2018
B.Eng., Zhejiang University, 2018

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in the School of Computing Science Faculty of Applied Sciences

© Yilong (Jetic) Gū 2019
SIMON FRASER UNIVERSITY
Summer 2019

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Yilong (Jetic) Gū
Degree: Master of Science
Title: Exploiting Target Syntax with Structured Decoding

Examining Committee: Chair: Qianping Gu
                    Professor
                    Anoop Sarkar
                    Senior Supervisor
                    Professor
                    William Sumner
                    Supervisor
                    Assistant Professor
                    Oliver Schulte
                    Internal Examiner
                    Professor

Date Defended: 18 July, 2019
Abstract

Incorporating structured target syntax in a neural machine translation model requires an effective tree-structured neural generator. We exploit a top-down tree-structured model called DRNN (Doubly-Recurent Neural Networks) first proposed by Alvarez-Melis and Jaakola (2017) to create an NMT model called Seq2DRNN that combines a sequential encoder with tree-structured decoding augmented with a syntax-aware attention model. Unlike previous approaches to syntax-based NMT which use dependency parsing models, our method uses constituency parsing which we argue provides useful information for translation. In addition, we use the syntactic structure of the sentence to add new connections to the tree-structured decoder neural network (Seq2DRNN+SynC). We compare our NMT model with sequential and state of the art syntax-based NMT models and show that our model produces more fluent translations with better reordering. Since our model is capable of doing translation and constituency parsing at the same time we also compare our parsing accuracy against other neural parsing models. We also show that our proposed model is capable of learning even more strictly defined programming language syntax by modelling its Abstract Syntactic Trees, reaching new state-of-the-art exact match accuracy on Django dataset without having to resort to syntactic rule-based decoder.

Keywords: Neural Machine Translation; Constituency Parsing; Code Generation; Syntax; Machine Learning
Acknowledgements

I would like to thank Dr. Anoop Sarkar, who has been my supervisor since I joined the SFU Natlang lab. Thanks to you, I was able to begin my journey in the field of computational linguistics. The knowledge that I gained from thence continues to benefit and bring glimpses of smile on my otherwise plain face.

I would also like to thank Dr. William Sumner, your countless valuable input seriously opened my mind and improved the quality of my work extensively.

Special thanks to my good friend Hassan S. Shavarani, without you I would not be able to have so much fun doing research. I most sincerely hope you will consider getting a cat named Monty, preferably a silly one.

Finally, I would like to thank all my family and friends. A life without you all would be utterly dull... well not dull, just depressing.
# Table of Contents

Approval ............................................. ii  
Abstract ............................................ iii 
Acknowledgements ................................... iv  
Table of Contents ................................. v  
List of Tables ..................................... vii  
List of Figures .................................... viii  

## 1 Introduction ................................. 1

## 2 Neural Machine Translation and Code Generation .......... 3  
2.1 Neural Networks ........................................ 3  
2.1.1 Artificial Neurons: Basic Neural Network Components ...... 4  
2.1.2 Feed-Forward Neural Networks ............................. 6  
2.1.3 Neural Network Training: Backpropagation .................. 8  
2.1.4 Recurrent Neural Networks ............................... 10  
2.2 Encoder-Decoder Architecture and Seq2Seq .................... 12  
2.3 Attention Mechanisms in Sequence-to-Sequence Models ......... 15  
2.4 Word-Level Sequential Recurrent Neural Network Language Model .... 16  
2.5 Machine Translation Evaluation ................................ 16  
2.6 Using Sequence-to-Sequence for Code Generation ............... 18  
2.6.1 Capturing Rigid Syntax of Programming Languages .......... 18  
2.6.2 Copy Mechanism .................................... 19  

## 3 NMT Using Doubly-Recurrent Neural Network ............... 21  
3.1 Background: Syntactic Information for Natural Language ....... 21  
3.2 Prior Research on Using Syntax in NMT ....................... 22  
3.3 NMT with a Tree-Structured Decoder (Seq2DRNN) ............... 23  
3.4 Doubly-Recurrent Neural Network ............................ 23
3.5 Parsing and Translating with DRNN ........................................... 25
3.6 Attention Mechanism .................................................................. 26
3.7 SynC: Syntactic Connections for Language Generation (Seq2DRNN+SynC) 27
3.8 Code Generation with DRNN and Sketching ................................. 29

4 Experiment and Analysis ................................................................. 31
  4.1 Natural Language Translation Experiments ................................... 31
    4.1.1 Model Training .................................................................. 31
    4.1.2 Modelling Detail .................................................................. 32
    4.1.3 Results ............................................................................... 33
    4.1.4 Attention Module ................................................................ 34
    4.1.5 Parsing Quality .................................................................. 36
  4.2 Code Generation Experiments ..................................................... 37
  4.3 Trainer Experiments .................................................................. 38

5 Related Work and Conclusion ......................................................... 40
  5.1 Related Work ........................................................................... 40
  5.2 Conclusions and Future Work ................................................... 41

Bibliography .................................................................................. 42
List of Tables

Table 2.1  Code Generation task example ........................................ 18
Table 2.2  Code Generation sketching example (Dong and Lapata, 2018) .... 19
Table 4.1  Dataset information .......................................................... 32
Table 4.2  News Commentary v8 dataset information ............................ 32
Table 4.3  IWSLT17 translation experiment results ............................... 32
Table 4.4  News Commentary v8 translation experiment results ............... 33
Table 4.5  Translation samples ........................................................... 34
Table 4.6  Unknown noun experiment samples .................................... 35
Table 4.7  Parser scores ................................................................. 36
Table 4.8  IWSLT translation result constituency scores. ........................ 37
Table 4.9  Results on Django dataset ............................................... 38
List of Figures

Figure 2.1 Architecture of an artificial neuron . . . . . . . . . . . . . . . . . . 4
Figure 2.2 Plot of sigmoid function . . . . . . . . . . . . . . . . . . . . . . . . 6
Figure 2.3 Single-layer feed-forward neural network . . . . . . . . . . . . . . 7
Figure 2.4 Plot of local/global minimum in a function . . . . . . . . . . . . . . 9
Figure 2.5 A Feed-Forward Neural Network Unit and an RNN unit. . . . . . . 11
Figure 2.6 An RNN unit with input $f = x_1, x_2, ..., x_n$, output $e = y_1, y_2, ..., y_n$. 11
Figure 2.7 Bags-of-Words in Action . . . . . . . . . . . . . . . . . . . . . . . . 12
Figure 2.8 Word Embedding Example . . . . . . . . . . . . . . . . . . . . . . . 13
Figure 2.9 RNN Encoder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Figure 2.10 BiRNN Encoder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
Figure 2.11 Finding the target word using $o_j$ vector. . . . . . . . . . . . . . . 15
Figure 2.12 Code Generation with sketching by Dong and Lapata (2018) . . . 19
Figure 3.1 Constituency Tree for “Andrei likes Cheese”. . . . . . . . . . . . . . 22
Figure 3.2 Seq2DRNN on Constituency Tree . . . . . . . . . . . . . . . . . . . 25
Figure 3.3 Seq2DRNN Encoder-Decoder Architecture . . . . . . . . . . . . . . 26
Figure 3.4 SynC information flow example . . . . . . . . . . . . . . . . . . . . 29
Figure 3.5 SynC in DRNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
Figure 4.1 Seq2DRNN+SynC attention visualisation . . . . . . . . . . . . . . . 36
Figure 4.2 Training loss curve (Momentum SGD VS Adam) . . . . . . . . . . . 39
Figure 4.3 Validation loss curve (Momentum SGD VS Adam) . . . . . . . . . . 39
Chapter 1

Introduction

Machine Translation is one of the most prominent research topics in Natural Language Processing. It is the task of generating an equivalent translation in a target language given some input in a source language.

With machine learning, translation models are usually trained to maximising the probability of the output sequence given input. Early phrase-based statistical solutions (Koehn et al., 2003) often exploit language specific syntactic information, including POS(Part-of-Speech) tags (§3.1) and tree structures (§3.1). As research into neural networks (Hagan et al., 1996) advances, Bahdanau et al. (2014) proposed syntax-agnostic neural machine translation (NMT) models, surpassing statistical models in both automated translation metrics and fluency. Bahdanau et al. (2014) use a Recurrent Neural Network (Schuster and Paliwal, 1997) to encode the source passage into hidden distributed representations, then passed to another RNN decoder to produce the target passage. These hidden representations bridging the encoder and decoder are assumed to contain all semantic information of the passage.

In this thesis, we propose an NMT model under the encoder-decoder architecture called Seq2DRNN model (Sequence to Doubly-Recurrent Neural Networks). Recent publications show that even for neural models, syntax could improve translation quality. The drawback however, is the difficulty of designing structured neural models to do so. In this thesis, we attempt to overcome that by using a tree-structured neural decoder proposed by Alvarez-Melis and Jaakkola (2017) called Doubly-Recurrent Neural Networks (DRNN). The resulting Seq2DRNN is an NMT model capable of translation and syntactic parsing at the same time, and we show that it demonstrates overall stronger performance against state-of-the-art baselines. We also propose syntactic connections (SynC), an extention to Seq2DRNN. On top of existing syntactic trees, SynC strengthens clausal relation between words and subclauses. Our experimental results show that the combination of Seq2DRNN and SynC significantly improves translation quality.

In addition to machine translation, we also exploit Seq2DRNN to perform programme synthesis: the task of given natural language description, generating the corresponding code.
in a general purpose programming language. We show that without any rule-based syntactic constraints, Seq2DRNN can learn programming language syntax by looking at Abstract Syntactic Trees (AST), and improve overall exact match accuracy against state-of-the-art neural + rule-based syntax baseline.

We review the concept of Neural Network, and introduce Encoder-Decoder architecture for machine translation in §2. In §3, we introduce our proposed model that incorporates structural syntactic information in an NMT system, and our novel Syntactic Connections (SynC). We present the experimental results in §4. In §5, we give the conclusions of the thesis and discuss the related works.
Chapter 2

Neural Machine Translation and Code Generation

Machine translation is the task of given input passage (source) in language 1, generating the semantically equivalent passage (target) in language 2. In computational models, it is described as given input sequence $f$, producing target sequence $e$ that conveys the same meaning.

Machine translation often utilises huge bilingual parallel corpora, in which sentence pairs consisting of both the source and target languages are used to train the model. The model is expected to recognise the patterns within these parallel text, and use them in translation.

Studies by Kalchbrenner and Blunsom (2013); Sutskever et al. (2014); Bahdanau et al. (2014) show that Neural Machine Translation (NMT) models that do not explicitly model sentence structure can produce better results both in terms of quality and speed than Statistical Machine Translation systems which do (Chiang, 2005; Zollmann and Venugopal, 2006; Galley et al., 2006). These NMT systems operate under the assumption that explicit modelling of syntactic features such as Part-Of-Speech (POS) tags is not necessary.

A brief introduction to neural networks is given in §2.1. We introduce the Encoder-Decoder NMT system architecture in §2.2. Modern NMT systems often utilise attention modules (Bahdanau et al., 2014; Luong et al., 2015) to guide sequential generation. We explain the basics of attention mechanisms in §2.3. The word-level neural language model is discussed in §2.4. In §2.5 we discuss the commonly used evaluation metrics for machine translation.

2.1 Neural Networks

Neural Networks are very powerful models designed for pattern recognition. The architecture of Neural Networks is inspired by the way human brains work.

Hebb (1949) describes a human brain as a huge network of neurons connected with each other. When the brain receives sensory information, some of the neurons are activated and
pass information to the connected neurons. Eventually, the information passed to specific neurons dictate our behaviour. In turn, the neural network improves as the amount of information from the feedbacks of behaviours increases.

Artificial neural networks are inspired by real neurons. In these neural network models, neurons are abstracted as interconnected computational units. Together, they form a network.

Like most statistical machine learning models, a neural network model has to be trained on prepared samples prior to deployment. Metaphorically speaking, by providing a neural network model a dataset of samples, it learns the patterns within these samples and gains the ability to apply these patterns on unseen samples. The former step is called training, the latter testing or inference.

In §2.1.1 we briefly introduce the components in an artificial neural network. §2.1.2 introduces Feed-Forward Neural Network (Hornik et al., 1989), and §2.1.3 describes a method for training neural networks called backpropagation. Finally in §2.1.4 we introduce Recurrent Neural Networks. For more details of neural networks, readers may refer to the book by Goodfellow et al. (2016).

2.1.1 Artificial Neurons: Basic Neural Network Components

A neural network is a network of artificial neurons, usually in the form of a directed graph. Each neuron has an integer state, or output, for example 1 means active and 0 means inactive. Such state is determined by the states of all connected neurons (Equation 2.1), as illustrated in Figure 2.1.

\[ y = f(x_1, x_2, ..., x_n) \]  

One very simple type of neurons is called a Perceptron (Rosenblatt, 1958). A perceptron takes several binary inputs \((x_1, x_2, ..., x_n, \forall i, x_i \in \{0,1\})\), and produces one single binary output \((y \in \{0,1\})\). The output of the neuron is an activation function conditioned on the weighted sum of all its input: if the sum is greater than a specific threshold value \(p\) then the output is 1, if not it is 0. All of the weights \(w_j\) are real numbers.
\[ y = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq p \\ 1 & \text{if } \sum_j w_j x_j > p \end{cases} \quad (2.2) \]

Equation 2.2 can also be written in vector form (Equation 2.3). Here is usually referred to as the bias, and is equivalent to \(-p\) in Equation 2.2.

\[ y = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (2.3) \]

By adjusting the weights and bias of a perceptron unit, many different functions can be defined. For example, a perceptron that takes two inputs can mimic the behaviour of a logical OR gate (Equation 2.4), AND gate (Equation 2.5), or NOT gate (Equation 2.6).

\[ y = \begin{cases} 0 & \text{if } 0.5x_1 + 0.5x_2 + (-0.49) \leq 0 \\ 1 & \text{if } 0.5x_1 + 0.5x_2 + (-0.49) > 0 \end{cases} \quad (2.4) \]

\[ y = \begin{cases} 0 & \text{if } 0.5x_1 + 0.5x_2 + (-0.5) \leq 0 \\ 1 & \text{if } 0.5x_1 + 0.5x_2 + (-0.5) > 0 \end{cases} \quad (2.5) \]

\[ y = \begin{cases} 0 & \text{if } (-0.5)x + 0.5 \leq 0 \\ 1 & \text{if } (-0.5)x + 0.5 > 0 \end{cases} \quad (2.6) \]

Perceptrons are trained using a very simple algorithm, and a large enough amount of training samples. A training sample is defined as \(S_t = (d_t, x_t)\), with \(d_t\) being the desired output. First we initialise the weights of a perceptron unit using randomisation. Then, we go through all of the training samples, calculate the output result \(y_t\) and update the weights and bias using Equation 2.7.

\[ w(t + 1) = w(t) + (d_t - y_t)x_t(j) \]

\[ b(t + 1) = b(t) + (d_t - y_t) \quad (2.7) \]

Further, we can also treat bias \(b\) as a weight \(w_0\) where its corresponding input is always \(x_0 = 1\).

Perceptrons have wide applications in machine learning. In the area of Natural Language Processing (NLP) alone, they are often used in statistical models where each binary input represents a feature of the input sample. The quality of the model thus depends heavily on the selection of these features.

Aside from perceptrons, a more common type of neurons is the sigmoid neurons. Sigmoid neurons are similar to perceptron neurons and are defined in Equation 2.8. Here, \(\sigma\) is the sigmoid function defined in Equation 2.9.
A very desirable property of the sigmoid function is that the output of the sigmoid function is always between 0 and 1. Thus this value can be viewed as the probability of the sigmoid unit being activated.

### 2.1.2 Feed-Forward Neural Networks

Neurons in §2.1.1 alone are very powerful, but feature engineering is not always feasible. As the task at hand gets complicated, tailoring thousands of useful features is simply impractical. One way of looking at neural networks is that instead of determining what features to use manually, we are using a network of neurons to detect useful information to look at from samples.

The architecture of a single-layer Feed-Forward Neuron Network is presented in Figure 2.3. This construction of neural networks is called feed-forward, because the neurons at each layer all look back at previous layers for input and generates input for the next layer. That is, there are no cycles in the network graph. Each layer of a neural network has multiple neurons. Each neuron is only connected with neurons at the adjacent layers but not within the same layer.
Figure 2.3: Single-layer feed-forward neural network. The Input Layer and the Output Layer each has 4 units, the hidden layer has 5 units. Each neuron is a sigmoid neuron.

The input layer handles the input of the model. For example if the input is a grey-scaled 128 pixel image, then we might need 128 neurons in the input layer, each represents the grey-scaled value of a pixel.

Then there is the hidden layer. It is more often to have multiple hidden layers, each neuron on level \( l \) is connected to each of the neurons on level \( l - 1 \) for input. For the sake of simplicity, Figure 2.3 only contains a single layer.

The output layer also has multiple neurons, each denoting the possibility of making a single decision. If the task is to recognise handwritten digits (0-9), then we might need 10 units in the output layer, each denoting the possibility of the digit being one of the numbers from 0 to 9.

This seemingly simple construction is very powerful at specific tasks. As we increase the number of hidden layers and the number of neurons in each layer, the additional parameters may help capturing more useful information, thus improving the model’s representational power. Take the neural network in Figure 2.3 as an example, the calculation on the hidden layer and output layer are presented in Equation 2.10 and Equation 2.11.
\[ h = \sigma(W \times x + b) \]
\[ = \sigma(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\
 w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\
 w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \\
 w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} \\
 w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4} \end{bmatrix} \begin{bmatrix} x_1 \\
 x_2 \\
 x_3 \\
 x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\
 b_2 \\
 b_3 \\
 b_4 \\
 b_5 \end{bmatrix}) \] (2.10)

\[ y = \sigma(U \times h + c) \]
\[ = \sigma(\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} \\
 u_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} \\
 u_{3,1} & u_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} \\
 u_{4,1} & u_{4,2} & u_{4,3} & u_{4,4} & u_{4,5} \end{bmatrix} \begin{bmatrix} h_1 \\
 h_2 \\
 h_3 \\
 h_4 \\
 h_5 \end{bmatrix} + \begin{bmatrix} c_1 \\
 c_2 \\
 c_3 \\
 c_4 \end{bmatrix}) \] (2.11)

\[ = \sigma(U \times \sigma(W \times x + b) + c) \]

A neural network is essentially a function \( f(x) \) that goes through a series of matrix operations, calculates the possibilities of each output unit \( y \). The actual constructions of neural network models may vary and are task specific, for different layers one might even want to use different functions instead of sigmoid functions on neurons for various purposes, such as softmax. Sometimes the network may also appear not fully connected (e.g., adaptive connection) as oppose to the example in Figure 2.3, where each neuron is connected to all neurons on the previous and next layer. The neural network model for handwritten digit recognition proposed by LeCun et al. (1990) has four adaptively connected hidden layers, a total of 4,635 neurons, 98,442 connections. But the idea is the same. Just like performing linear regression or curve fitting on a set of coordinates to approximate the pattern, neural networks apply the same principle to higher dimensions using more complex equations with more parameters.

### 2.1.3 Neural Network Training: Backpropagation

In this section we introduce some basic ideas of a principle algorithm behind neural network training: backpropagation.

We use \( \theta \) to denote the set of all parameters in a given model \( M \). We use \( d \) to denote the reference output given input \( x \), and \( y \) to denote the actual output. The goal to achieve during training is to find the parameter set \( \hat{\theta} \), so that \( y = M(\theta, x) \) is as close to \( d \) as possible. We define a loss function \( E(y, d, x, \theta_t, M) \) which measures how incorrect \( \theta_t \) at step \( t \) is. The objective thus becomes finding the \( \hat{\theta} \) that minimises the loss \( E(\cdot) \). The design of the loss function is of vital importance since it is the only measure we are using for evaluation during training.
A common approach to achieve this is to use gradient descent. We compute the derivative of $E(\cdot)$ with respect to $\theta_t$. The property of positive and negative of the derivative shows in which direction (increase or decrease) do we change the weights may we reduce the loss. We update each parameter $w \in \theta$ in that direction using Equation 2.12.

$$w := w - \epsilon \frac{\partial E(\cdot)}{\partial w}$$

(2.12)

Here $\epsilon$ is called the learning rate. When $\epsilon$ is greater, $\Delta \theta$ would also be greater.

The nature of gradient descent dictates that the above method can only guarantee the discovery of a local minimum (Figure 2.4). The methods one could use to tackle this issue are out of the scope of this thesis.

As the number of layers increases, we apply chain rule (Equation 2.13) to compute the gradients.

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

(2.13)

With each sample, we first do a forward pass, calculating $y = M(\theta_t, x)$ given current parameter set $\theta_t$. This is called forward propagation. Then, we do backward propagation, calculate the gradients at each step of the network from the bottom layer (output layer) back to the top layer (input). During backward propagation, the parameters are updated using the gradients.

We use the neural network in Equation 2.11 as an example. For the purpose of this example we define the loss function as $E(\cdot) = (d - y)^2$. During the forward pass, $h$ and $y$ are calculated. Then during the backward pass, we first update the parameters connecting the hidden layer and the output layer:
Using the same method, we then update the parameters connecting the input layer and the hidden layer:

\[
W := W - \epsilon \frac{\partial E(\cdot)}{\partial W}
\]

\[
:= W - \epsilon \frac{\partial (\alpha^2) \partial (d - \sigma(\beta)) \partial (U \times h + c)}{\partial \beta} \frac{\partial (U \times h + c)}{\partial W}
\]

\[
\text{where } \frac{\partial \beta}{\partial W} = \frac{\partial (U \times \sigma(\gamma) + c)}{\partial \gamma} \frac{\partial (W \times x + b)}{\partial W}
\]

(2.15)

As one can see, parts of Equation 2.15 are already computed in Equation 2.14 and can be directly utilised. At each step of the backward propagation, we will only need to calculate the derivative of parameters connecting that particular layer and the next layer.

One of the most important factors in training a neural network using backpropagation is the choice of learning rate \(\epsilon\). Different techniques that dynamically adjusts the learning rates may perform very differently when applied to different models. In this thesis, all experiments unless otherwise specified use the Adam trainer (Kingma and Ba, 2015).

### 2.1.4 Recurrent Neural Networks

One issue of a Feed-Forward Neural Network is that it can only process fixed-length input. For language processing tasks where the input might be a variable length sequence, we will need to make the neural network preserve sequential information.

One of the solutions is Recurrent Neural Networks (RNN). RNNs differ from Feed-Forward Neural Networks in the sense that their network graphs contain feedback cycles. If we treat a feed-forward neural network as a single unit which takes an input and produces an output, a recurrent neural network would in addition also take its previous hidden layer values (hidden representation, which contains information regarding previously processed entries) as input (Figure 2.5).

Given a sequence of input \(f = x_1, x_2, ..., x_n\), we use an RNN unit to process each of \(x_t\), and produce output sequence \(y_t\) simultaneously. It will look like Figure 2.6. At time
Figure 2.5: A Feed-Forward Neural Network Unit and an RNN unit.

Figure 2.6: An RNN unit with input $f = x_1, x_2, ..., x_n$, output $e = y_1, y_2, ..., y_n$.

For $t = 1$, the RNN unit receives initial value $h_0$ as its previous state input alongside $x_1$. When producing $y_1$, the values of its hidden states are stored as $h_1$ to be utilised at time $t = 2$, etc.

Here is an example of a basic RNN unit construction:

$$h_t = \tanh(Wh_{t-1} + Ux_t + b)$$
$$y = \text{softmax}(Vh_t)$$

(2.16)

In Equation 2.16, $\text{softmax}(\cdot)$ is a function that applies on a vector, making the normalised values exponentially more distant from each other. $\text{softmax}(\cdot)$ is often applied on the output layer of a neural network.

The training of an RNN is done by building a computational graph, where a complete sequence is processed and all values calculated (forward pass). Then we calculate the
gradients, update the parameters using the averaged gradients. This technique is called backpropagation through time.

2.2 Encoder-Decoder Architecture and Seq2Seq

In this thesis, a source passage will be written as \( f = \{x_1, x_2, ..., x_n\} \), and a target passage as \( e = \{y_1, y_2, ..., y_m\} \). The concatenation of vectors \( v \) and \( u \) is denoted as \([v; u]\). \( W(x) \) is the vector representation of a natural language word \( x \) (word embedding).

Neural machine translation models generally consist of an encoder and a decoder (Bahdanau et al., 2014; Luong et al., 2015). The idea behind it is that any sentence can be represented by hidden representations (vector), incorporating all useful source information. The encoder’s purpose is to compute the representation given input, and the decoder’s to output in target language accordingly.

Many models have been proposed to encode the source sentence. Early attempts include using Bag-of-Words (BoW) representation. Bag-of-Words is the method of encoding a sequence by combining representations of all individual words regardless of their order. Figure 2.7 shows an example of NMT using BoW as encoder. The hidden representation’s dimension in this case equals the size of the vocabulary, with each row representing the presence of a word in the sentence. If the vocabulary size is 50000, then the hidden representation would be a 50000-dimensional vector. If the word is in the sentence, its entry value would be 1, if not then 0.

A very obvious problem of BoW is that it ignores word order, which may cause ambiguity. The representation of the sentence \( \text{may likes mike} \) would be identical to \( \text{mike likes may} \), but their meanings are not the same.
A better option would be to use Recurrent Neural Network (RNN) units. Recall in §2.1, we describe an RNN unit as a function (Equation 2.17), in which the hidden representation $h$ at step $t$ is calculated using previous hidden representation ($h_{t-1}$) and the $t$-th input $v_t$.

$$h_t = \text{RNN}(h_{t-1}, v_t) \quad (2.17)$$

We regard a sentence as a sequence of words, and encode the sentence following the direction of the sequence. To utilise an RNN as encoder, words need to be represented in a distributed vector space. This is called a word embedding (Figure 2.8).

We use function $W(x)$ to represent the word embedding vector of word $x$.

Then, an encoder to encode source sentence $f = \{x_1, x_2, ..., x_n\}$ can be written as:

$$h_i = \text{RNN}_{\text{enc}}(h_{i-1}, W(x_i)) \quad (2.18)$$

And the representation of the entire sentence would be $h_n$ where $n$ is the length of the sequence. Figure 2.9 shows the construction of an RNN encoder.

The problem of RNN is that as the sequence length gets longer, information at earlier stages (e.g., the first few words in a sentence) could get lost due to the limited dimension of hidden vectors. A common solution is to use bidirectional RNNs. We produce forward hidden states $\overrightarrow{h}_i$ and backward hidden states $\overleftarrow{h}_i$, and representation $h_i^{\text{enc}}$ as the concatenation of both:

$$\overrightarrow{h}_i = \overrightarrow{\text{RNN}}_{\text{enc}}(\overrightarrow{h}_{i-1}, W_x(x_i))$$

$$\overleftarrow{h}_i = \overleftarrow{\text{RNN}}_{\text{enc}}(\overleftarrow{h}_{i+1}, W_x(x_i)) \quad (2.19)$$

$$h_i^{\text{enc}} = [\overrightarrow{h}_i; \overleftarrow{h}_i] \quad (2.20)$$
Figure 2.9: RNN Encoder. We often add an Start-Of-Sentence token <SOS> to mark the beginning of a sentence and an End-Of-Sentence token <EOS> to mark the end.

Figure 2.10: BiRNN Encoder

The representation of the sentence is the concatenation of $\overrightarrow{h_n}$ and $\overleftarrow{h_0}$. (Forward and backward hidden representation of the sentence)

$$h^{enc} = [\overrightarrow{h_n}; \overleftarrow{h_0}] \quad (2.21)$$

Figure 2.10 shows the construction of BiRNN encoder.

After obtaining the hidden representation of the source sentence, we will need a decoder to generate the output sequence in target language.

For a sequential decoder, the generation at each step $j$ is essentially based on the probability of next word conditioned on i) all previously generated words $o_{<j}$; and ii) the hidden representation of the source sentence $h^{enc}$.

We use another RNN unit to achieve this:

$$h^{dec}_j = \text{RNN}_{dec}(h^{dec}_{j-1}, o_{j-1}) \quad (2.22)$$

$$o_j = \text{softmax}(Uh^{dec}_j + b) \quad (2.23)$$
Here, $U$ is the readout matrix and $b$ is the bias vector. $o_j$ is the output word embedding, which is the probability distribution of the next word $y_j$. The row number of the highest probability in $o_j$ is the index of the word in the target language lexicon (Figure 2.11).

The Encoder-Decoder model described above is often called a sequence-to-sequence model or Seq2Seq (Sutskever et al., 2014; Luong et al., 2015).

### 2.3 Attention Mechanisms in Sequence-to-Sequence Models

Attention mechanisms aid decoding by providing extra contextual information (usually in the forms of context vectors) to the decoder. At each step $j$ of the decoding, the attention module would generate a context vector $c_j$ to be concatenated with the previous hidden state $h_{j-1}^{dec}$, then fed into the decoder as input.

$$h_j^{dec} = \text{RNN}_{dec}(h_{j-1}^{dec}; c_j, o_{j-1}) \quad (2.24)$$

Luong et al. (2015) proposed a very simple attention mechanism that improved the performance of Seq2Seq greatly. In this section, we describe one of the attention mechanisms proposed by Luong et al. (2015) that is going to be used during our experiments. It is a global attention mechanism, in the sense that it looks at all of the words in the source sentence for contextual information.

The mechanism is defined as follows:

$$\beta_{j,i} = V_a \tanh(W_a h_{j-1}^{dec} + U_a h_i^{enc}) \in \mathbb{R}$$

$$\alpha_j = \text{softmax}(\beta_j) \quad (2.25)$$
\[
\mathbf{c}_v = \sum_{i=1}^{n} \alpha_{j,i} \mathbf{h}_{i}^{\text{enc}}
\]  

(2.26)

Essentially, we use the representation of the decoder’s previous state and each of the hidden representations of the encoder to calculate a relevance score ($\beta$), then pass it through a softmax layer for $\alpha$. Then we produce a $\alpha$ weighted sum of all the hidden states from encoder as the context vector. $\alpha$ here in Equation 2.25 is also referred to as the attentions score.

2.4 Word-Level Sequential Recurrent Neural Network Language Model

A language model is the probability distribution over a sequence of words (Bengio et al., 2003). Take the target side as an example, it would be represented as

\[
P(e) = P(y_1, y_2, ..., y_n)
\]

(2.27)

Language models (LM) form an important part of a decoder, or any other language generation model. It shows exactly on what the generation of each component of the sentence is conditioned, determines the fluency of the generated sentence.

The standard Seq2Seq described in §2.2 uses an RNN language model conditioned on the input representation produced by the encoder to generate the output one word at a time from left to right (see Equation 2.28). The prediction of a word $y_j$ is directly conditioned on all previously generated words where $c_j$ is the context vector from the input.

\[
P(e) = \prod_j P(y_j | y_{<j}, c_j)
\]

(2.28)

RNN word-level language models are able to produce more fluent sentences than $N$-gram language models (Mikolov et al., 2010). The assumption goes, since RNN LMs are not restricted to fixed length input, it can pay more attention to long-range dependencies (Khandelwal et al., 2018).

The problem with word-level language models is that it does not explicitly model structural information, which could lead to the same issues suffered by $N$-gram models, where the sentence is only fluent on a micro-level (e.g., I happen to be or not to be that is the question, where if looking at each word and its neighbour it appears fluent, but the sentence in its entirety is not).

2.5 Machine Translation Evaluation

The most common evaluation metric for machine translation is BLEU, which is based on modified $N$-gram precisions. $N$-gram precision measures the occurrence of every subsequence
of $N$ consecutive tokens. This way, generating incorrect words or missing reference words can be penalised.

Take unigram precision, which is single word precision ($N = 1$). We have the following sentence:

- Output: I love cheese very much
- Reference: I like cheese very much

Among the 5 words in the output, 4 occur in the reference, the unigram precision is $4/5$.

If we use bigram, which is bi-word precision ($N = 2$), then there are 4 bi-words in the output sentence: \{I love, love cheese, cheese very, very much\}. This time only one occurs in the reference, so the bigram precision is $1/4$.

In machine translation, models could cheat bigram precision evaluation by generating high precision words and ignoring the information it is supposed to preserve:

- Output: the cat the cat the cat the cat
- Reference: the cat lying on the table

In this case since all of the words in the output occur in the reference, it would score a unigram precision of 1.0.

With BLEU, the modified $N$-gram precision would remove matched pairs of words from the reference so they do not get counted a second time. In the above example, the cat the cat the cat the cat would score a modified unigram precision of only $3/8$. The final BLEU score of the sentence is the weighted sum of modified unigram, bigram, 3-gram,…, $N$-gram precisions, where $\text{max}(N)$ is usually smaller than 5.

A problem of BLEU is that it disregards word order. Take unigram precision, output I like cheese and cheese like I would have the same unigram precision, but they convey very different meanings. Even when considering 3-grams and more words, the MT could still score highly by intentionally generating high precision single words and pairs of words to overweight 3-gram and 4-gram scores.

To address this issue, Isozaki et al. (2010) proposed the RIBES score, which is more sensitive to reordering. It adds a rank correlation coefficient prior to unigram matches without the need for higher order $N$-gram matches.

Both BLEU and RIBES have a lot of problems. They are all word-level metrics that regard synonyms and near synonyms as mismatches, they all disregard the context of the sentence and the actual meanings. Tan et al. (2015) in their work discuss the cases where systems producing higher BLEU and RIBES scores sometimes get worse human evaluations. However, the advantages of BLEU and RIBES, being that they are efficient and language independent still make them indispensable in the research of NMT.
Intent | if length of bits is lesser than integer 3 or second element of bits is not equal to string 'as'
---|---
Python | if len(bits) < 3 or bits[1] != 'as':

Table 2.1: Code Generation in the scope of this thesis refers to generating general purpose programming language based on natural language description. Example shown here is English to Python, which could be modelled using machine translation seq2seq models.

2.6 Using Sequence-to-Sequence for Code Generation

Seq2Seq can model arbitrary length input and output, as long as they can be serialised into sequence of tokens. In recent years Seq2Seq has been used for many tasks, including summarisation (Gu et al., 2016), syntactic parsing (Vinyals et al., 2015b), code generation (Ling et al., 2016; Iyer et al., 2017; Yin and Neubig, 2017; Iyer et al., 2018), etc.

In this paper, we are interested with code generation, or programme synthesis. Code generation in this thesis refers to the task of given natural language description, and generating equivalent code that is in a prespecified general-purpose programming language. It can be easily formulated as a Machine Translation task, the only difference is that instead of natural language, code generation has programming language on the target side. Differing from natural language, the unique properties of programming languages introduces new challenges.

2.6.1 Capturing Rigid Syntax of Programming Languages

Programming languages, unlike natural language, follow strictly designed rigid grammar to allow for consistent and efficient interpretation by computing machinery. This rigid grammar is hard to capture using grammar agnostic Seq2Seq models.

Recent work has shown that relying on rule-based enforcers to make sure the decoder conforms (Ling et al., 2016; Yin and Neubig, 2017) may yield in better results. The problem with this approach is that the design and derivation of such rule-sets and enforcers can also lead to error propagation and complication with flexibility, feasibility, and extensibility. Ideally, a system that can automatically learn syntax without relying on ad hoc rule-based constraints would be much more desirable.

Dong and Lapata (2018) propose to utilise sketches of programming code during decoding. The idea is that the sketch would carry essential structural information of the programming code while leaving the coarser details to be filled by a separate RNN (Table 2.2). Strictly speaking, sketching does not model syntax, it rather uses a separate component to learn structural knowledge. This is shown to improve general performance beyond systems with rule-based constraints.

For each piece of programming code, a pre-defined set of keywords are kept in the sketch while the rest are replaced by placeholders.
| Intent | if length of bits is lesser than integer 3 or second element of bits is not equal to string 'as' |
| Sketch | if len(NAME) < NUMBER or NAME | NUMBER | != STRING : |
| Python | if len(bits) < 3 or bits[1] != 'as': |

Table 2.2: Sketching (Dong and Lapata, 2018). All variable names, non built-in functions, and literal values are replaced with placeholders.

Figure 2.12: Code Generation with sketching by Dong and Lapata (2018). The encoder and the sketch decoder form a seq2seq model. The encoder and the final decoder also form a seq2seq model, with sketch encoder providing extra information to the decoder.

In addition to Seq2Seq, Dong and Lapata (2018) add two more components: a sketch decoder and a sketch encoder. At inference time, the source encoder processes the input passage, and the sketch decoder generates the sketch accordingly. Then, the sketch is encoded with the sketch encoder, and fed into the final decoder alongside the encoded natural language input (Figure 2.12).

With the sketch decoder reaching relatively high accuracy (77.4 for django dataset), the sketch encoder may provide useful structural information to the final decoder without requiring ad hoc rules.

### 2.6.2 Copy Mechanism

Neural machine translation models using word-level embeddings tend to have fixed vocabulary size. On the other hand, segments of the programming code may also be directly copied from the input passage, such as variable names, function names, numerical values, strings, etc. To be able to leverage that, copy mechanisms (Vinyals et al., 2015a; Gu et al., 2016; See et al., 2017) are often used to ensure maximum lexical coverage.

A very simple way of copying is just to look at the attention scores calculated in Equation 2.25, and copy the source word \( f_i \) with highest \( \alpha_{j,i} \) to \( e_j \). When the decoder outputs specific token (such as the unknown token UNK in machine translation), copy mechanism kicks in and assumes that attention scores points to the source word that should get translated now.
A more advanced model called Pointer Generator (See et al., 2017) uses a separate classifier to determine whether to rely on decoder output or copy mechanism for the current target word (Equation 2.29).

\[
P_{gen}(i) = \sigma(u_{copy} \cdot h_i^{dec} + b) \tag{2.29}
\]

Here \( P_{gen} \) is a sigmoid activation unit. It calculates the confidence in decoder output at time \( i \), which in turn determines whether to copy. \( P_{gen} \) decides to use copy, the model like the simple method would look at the attention score and pick the word.

The loss function for training the copy classifier is defined in Equation 2.30

\[
\text{Loss}_{\text{label}} = -\log(P_{dec}(e_i = \hat{e}_i|e_{<i}, f))
\]

\[
\text{CopyAlpha}_i = \sum_{j=\hat{e}_i}^{} \alpha_{j,i}
\]

\[
\text{Loss}_{\text{copy}} = P_{gen}(i) \times P_{dec}(e_i = \hat{e}_i|e_{<i}, f) + (1 - P_{gen}(i)) \times \text{CopyAlpha}_i
\]

\[
\text{Loss} = \text{Loss}_{\text{label}} + \text{Loss}_{\text{copy}} \tag{2.30}
\]

Here, \( \alpha_{i,t} \) is the attention score of target position \( t \) and source word \( i \), while \( P_{dec} \) is the probability distribution computed by the decoder. \( \hat{e}_i \) is the reference.
Chapter 3

Neural Machine Translation Using Doubly-Recurrent Neural Network

The nature of this thesis is to investigate the utilisation of syntactic information with a structured decoder in an NMT architecture. That covers natural language to natural language translation and natural language to programming language translation. For translation with natural language as target, we first introduction the relevant syntactic information.

3.1 Background: Syntactic Information for Natural Language

Syntax deals with the arrangement of words and phrases to create sentences. It is language dependent. Statistical Machine Translation often operate under the assumption that translation models can benefit from statistical analysis of syntactic features.

In this thesis, we utilise two types of syntactic information for natural language translation: Part-of-Speech tags and Constituency parse trees.

Part-of-Speech Tags

Part-of-Speech (POS, sometimes also grammatical tags) is a type of syntactic tag for words. The process of POS tagging looks at both the definition of the word and its context (relation with other words in the sentence). The most common tags include nouns, verbs, adjectives, adverbs, etc.

Part-of-Speech tags often are language specific and there are various POS standards for a single language. For example, English Penn Treebank includes 36 different tags (Santorini, 1990).

Constituency Trees

Constituency parsing is the process of breaking a sentence into clauses of words like noun phrases and verb phrases.
Figure 3.1: Constituency Tree for “Andrei likes Cheese”.

Figure 3.1 shows an example constituency parse tree. In practice, a constituency tree usually comes with POS tags attached to each individual word. The same tree can also be written in the following form:

\[(S
\quad (NP
\quad (NNP Andrei))
\quad (VP
\quad (VBZ likes)
\quad (NP
\quad (NN cheese))))\)

3.2 Prior Research on Using Syntax in NMT

The Seq2Seq model described in §2 is syntax-agnostic. While reaching better performance than statistical models (Bahdanau et al., 2014), recent work by Sennrich and Haddow (2016) show that incorporating source POS tags and dependency information improves NMT. Further, Stahlberg et al. (2016) use source language syntax to guide the decoder of an NMT system to follow hierarchical structures (Hiero) rules (Chiang, 2005). Eriguchi et al. (2016) and Bastings et al. (2017) use tree-structured encoders to exploit source language syntax. These approaches, while showing improvements in translation quality, require additional preprocessing steps using external tools for generating the syntactic information.

Target syntax modelling introduces complications, since NMT models are designed to model sequence, while complex grammar are usually represented as trees. For constituency trees, additional nodes are introduced to phrasal structure. The fact that surface tokens do not have a bijective relation with parse tree nodes requires either linearising the tree or using...
an effective structured decoder. Aharoni and Goldberg (2017) take the approach of serialising the parse trees to train a sequential decoder, relying on vanilla Gated RNNs’ representational power to learn complex structure. Eriguchi et al. (2017) propose an NMT+RNNG model, which explores the possibilities of using dependency information from the target language using StackLSTMs (Dyer et al., 2015, 2016) to aid a sequential decoder. All of these work show that using structural information on the target side improves translation quality. This is of particular interest to code generation, since as discussed in §2.6.1, programming languages follow much stricter syntactic constraints, which can be difficult to learn without explicit modelling.

In this thesis, for natural language translation, we seek to exploit constituency structure of the target side; and for code generation, we model the Abstract Syntax Tree structures automatically retrieved using Python’s AST library. Another distinction from previous work which often fuses sequential RNN language models with additional syntactic information, is that we use a structured decoder that generates the tree in top-down order. This marks a departure from sequential to hierarchical decoding.

### 3.3 NMT with a Tree-Structured Decoder (Seq2DRNN)

The output translation from a translation system should convey the same meaning as the input. This includes the correct word choices but also the right information structure. Sentence structure can be viewed as starting with an action or state (described via verbs or other predicates) and the entities or propositions involved in that activity or state (usually described via arguments to verbs). Thus certain words in the output translation, like verbs, are crucial to the understanding of the target language sentence but only provide marginal value in N-gram matching evaluations like BLEU. Tree representations, produced via dependency parsing and constituency parsing, and for programming language deterministic AST parsing, are useful because they are sensitive to this information structure. Our tree-structured decoder uses a neural network to generate trees (described in §3.4), which is incorporated into an NMT model (our novel encoder-decoder model is in §3.5) which translates and produces a parse tree. Our new syntactic connection (SynC) is described in §3.7, which is combined with the Seq2DRNN model (Seq2DRNN+SynC) and the attention mechanism (§3.6).

### 3.4 Doubly-Recurrent Neural Network

The Doubly-Recurrent Neural Network model (Alvarez-Melis and Jaakkola, 2017) takes a vector representation as input and generates a tree. Alvarez-Melis and Jaakkola (2017) show that the DRNN model can effectively reconstruct trees but they do not use DRNNs within a full-scale NMT system. DRNN decoding proceeds top-down; the generation of nodes at
depth $d$ depends solely on the state of nodes at depth $< d$. Unlike previous work in tree-structured decoding for NMT by Dyer et al. (2016) and Eriguchi et al. (2017), the output sentence generation is not done in sequence, where the target word $y_j$ is generated after all $y_{<j}$ are generated. DRNN first predicts the structure of the sentence and then expands each component to predict words. When generating $y_j$, information regarding the structure of words from 1 to $j - 1$ and $j + 1$ to $m$ can be used to aid prediction of $y_j$.

A DRNN consists of two recurrent neural network units, which separately process ancestral and fraternal information about nodes in the tree. Assuming a node is $v$, its immediate parent node is $P(v)$ and its closest sibling on the left side (appears in the target language sequence just before $v$) is $S(v)$. The label of node $v$ is $z_v$. Then the ancestral hidden representation $h^a_v$ and fraternal representation $h^f_v$ of a node are calculated with Equation 3.2.

$$h^a_v = \text{RNN}_{\text{dec}}^a(h^a_{P(v)}, z_{P(v)})$$

$$h^f_v = \text{RNN}_{\text{dec}}^f(h^f_{S(v)}, z_{S(v)})$$

(3.2)

$h^a_v$ and $h^f_v$ are then combined to produce the hidden state of node $v$ for prediction (predictive hidden state $h_v$, Equation 3.3), which is used to predict the labels of node $v$.

$$h_v = \tanh(U^f h^f_v + U^a h^a_v)$$

(3.3)

During the label prediction, DRNN first makes topological decisions: whether (i) the current node is a leaf node (node with no children, $\alpha_v$); then (ii) whether the current node has siblings on its right-hand side ($\gamma_v$). Both predictions are done using sigmoid activations:

$$\alpha_v = \sigma(u^a h_v)$$

(3.4)

\[ \alpha_v = 1 \text{ if } \alpha_v \text{ is activated.} \]

$$\gamma_v = \sigma(u^f h_v)$$

(3.5)

\[ \gamma_v = 1 \text{ if } \gamma_v \text{ is activated.} \]

Then, label representation $o_v$ is predicted using $\alpha_v$ and $\gamma_v$, and the predictive hidden state $h_v$:

$$o_v = \text{softmax}(U_o h_v + \alpha_v u_a + \gamma_v u_f)$$

(3.6)

At inference time each node at the same depth is expanded independently, it is possible to perform certain predictions in parallel. From a top-down perspective, inference is sequential as all nodes at depth $d$ are predicted before any node at depth $d + 1$. At each depth, all nodes at $d$ can be expanded into their individual subtrees in parallel, as these predictions do not have interdependencies. This parallelism advantage is not observed in any of the sequential decoders that generate output sequence strictly from left to right nor from right to left (§5 has more discussion).
3.5 Parsing and Translating with DRNN

A DRNN is capable of producing a tree structure with labels given an input vector representation. If we train the DRNN to produce parse trees from the output of an encoder RNN, this system will be able to translate and parse at the same time.

We use constituency parse trees to represent sentences in target language (Figure 3.2) because constituency or phrase-structure trees are more amenable to top-down derivation compared to dependency trees.

Each node on the tree represents either a terminal symbol (a word) or a non-terminal symbol (a clause or phrase type). The sub-tree dominated by a non-terminal node is the clause or phrase identified with this non-terminal node label.

A conventional bidirectional RNN (BiLSTM) encoder (Sutskever et al., 2014) is used to produce hidden states for the decoder (see Figure 3.3).

We use breadth-first search to implement the Seq2DRNN decoder. Two queues are used here: current queue which is the queue containing all of the nodes on the currently being processed depth, and next queue with nodes on the next depth (Algorithm 1 has all the details).

The decoding process starts from top to bottom, from root to its children, then to its grandchildren, and so on until the leaf nodes which are the output words.

In our implementation, sentence clauses (S nodes) are generated as the children of the root node to generalise over sentence types and in case there are multiple sentences in a single translation pair.

Initially, the current queue will only have one entry: the root node, which is initialised with the hidden representation of the source sentence.

Each node in the current queue is expanded in the following manner: first generate all of its siblings and add them to the current queue, and if any node happens to be non-terminal, generate its first child and add it to the next queue. After the current queue is empty, make next the new current queue and start working on nodes at the next depth.
For training, we use back-propagation through trees using the approach in Goller and Küchler (1996). In the forward pass, the source sentence is encoded into a hidden representation and fed into the decoder. The decoder generates the tree, predicts the labels of every node from root to leaves. Then in the backward pass, gradients are calculated and used to update the parameters. The loss calculation includes losses in topological predictions: $o_v^a$ and $o_v^f$ (Equations 3.4 and 3.5) and label predictions: $o_v$ (Equation 3.6).

$$\text{Loss}(e) = \sum_v \text{Loss}^{\text{label}}(o_v, \hat{o}_v) + \alpha \sum_v \text{Loss}^{\text{topo}}(o_v^a, \hat{o}_v^a) + \alpha \sum_v \text{Loss}^{\text{topo}}(o_v^f, \hat{o}_v^f)$$

(3.7)

### 3.6 Attention Mechanism

Attention mechanisms usually work by adding an additional context vector during label prediction. Luong et al. (2015); Bahdanau et al. (2014) show that attention mechanisms can greatly improve translation quality, without which Seq2Seq will not be able to surpass a statistical translation model (e.g. Phrase-Based MT by Koehn et al. (2003)). We use the variation of global attention first proposed by Luong et al. (2015). In our model, we produce
Algorithm 1 Seq2DRNN Decoder

1: procedure Decode(hiddenRep)
2:    currentQueue ← Node from hiddenRep
3:    nextQueue ← empty
4: loop:
5:    if currentQueue is not empty then
6:        node ← currentQueue.pop()
7:        Generate labels of node
8:        if node has siblings then
9:            currentQueue ← sibling(node)
10:       if node has children then
11:           nextQueue ← child(node)
12:       goto loop.
13:      all nodes at current depth are generated
14:      move on to the next depth
15:    if nextQueue is not empty then
16:        currentQueue ← nextQueue
17:        nextQueue ← empty
18:     goto loop.
19: both queues should be empty now

a context vector $c_v$ for every node $v$ by looking at all hidden states produced by the encoder $h_v^{enc}$, then calculating the weights and adding up the weighted hidden states.

$$\text{weight}_{v,i} = V_a \tanh(W_a h_v + U_a h_i^{enc}) \in \mathbb{R}$$  \hspace{1cm} (3.8)$$

$$c_v = \sum_{i=1}^{n} \text{weight}_{v,i} h_i^{enc}$$  \hspace{1cm} (3.9)$$

This way, our attention mechanism allows the target decoder to consider contextual information in assumingly more useful and relevant source positions during prediction of both terminal nodes and non-terminal nodes.

3.7 SynC: Syntactic Connections for Language Generation (Seq2DRNN+SynC)

A conventional Seq2Seq model uses an RNN language model (Bengio et al., 2003; Sutskever et al., 2014) conditioned on the input representation produced by the encoder to generate the output one word at a time (Equation 3.10). The prediction of a word $y_j$ is directly conditioned on previously generated words where $c_j$ is the context vector.

$$P(e) = \prod_j P(y_j|y_{<j}, c_j)$$  \hspace{1cm} (3.10)$$
The problem with this word-level language model is that it treats a sentence as a plain sequence of symbols regardless of its syntactic construction. Sentences may contain multiple subordinate clauses and their boundaries are not well-modelled by sequential language models.

We propose a new method to connect the hidden units in the Seq2DRNN decoder that pays attention to contextual tree relationships. The prediction of the representation of a word or a constituent $z_j$ (if a constituent then $p_j$, if a word then $y_j$) is defined as follows:

$$
P(z_j \mid y_{<j}, c_j) = \frac{1}{Z} \sum_{y_{<j}} \exp \left( \sum_{k} \alpha(k) \cdot \left( \sum_{z_j \in p_k} \delta \left( \text{precedes}(y_k, z_j) \right) \right) \right)
$$

The generation of the representation of a word/constituent $z_j$, which is part of the clause that contain it ($z_j \in p_k$), with clauses before which (precedes($p_k, z_j$)), is conditioned on the following information: i) Word-level: previously generated words $y_{<j}$; ii) Ancestral Clause: the clauses that contain the current word $p_k$, i.e. ($\forall k, z_j \in p_k$); iii) Fraternal Clause: the clauses that precede the current clause $p_k$, i.e. ($\forall k \text{ precedes}(p_k, z_j)$).

In practice, the generation of a node looks at the following representations:

1. Word-level: an RNN unit that produces the representation of previous words as a sequence $y_{<j}$;
2. Ancestral: treating the ancestors of the current node as a sequence (from root to the immediate parent), the representation of that sequence: $p_k(\forall k, z_j \in p_k)$;
3. Fraternal: treating the previous siblings of the current node as well as the previous siblings of its parent node and so on as a sequence, the representation of that sequence: $p_k(\forall k, \text{precedes}(p_k, z_j))$.

As shown in Figure 3.5, SynC creates additional connections in the tree-structured decoder for nodes originally without fraternal connections. This provides the decoder with mode useful structural context, for both terminal and non-terminal symbol prediction. For example in English, it is common for verb phrases to follow a noun phrase. But that noun phrase could itself be a subordinate clause with its own verb phrases. In this case, our goal is to explicitly model the fact that the previous phrase is a noun phrase instead of just the entire sequence of words.

SynC can be easily incorporated in the proposed Seq2DRNN model (Seq2DRNN+SynC). In addition to the fraternal RNN unit that focuses on preceding sibling nodes, and the ancestral DRNN unit that focus on parent nodes, a node would also look at its parent’s previous sibling state (the hidden vector representation of preceding clauses from the very beginning of the sentence). When a non-terminal symbol is expanded into a sub-tree, its first child will
not have a previous sibling to provide fraternal information ($S(v) = \text{Null}$, as in Eq. 3.2). In this case, SynC establishes connection between its first child and its parent’s fraternal information provider for such fraternal RNN state ($S(v) = S(P(v))$).

$$h_v^f = \begin{cases} S(v) \neq \text{Null}, & \text{RNN}_{\text{dec}}^{f}(h_{S(v)}^{f}, z_{S(v)}) \\ S(v) = \text{Null}, & S(v) := S(P(v)), \\ & \text{RNN}_{\text{dec}}^{f}(h_{S(v)}^{f}, z_{S(v)}) \end{cases}$$ (3.12)

In the example shown in Figure 3.5, a word-level language model will regard *starving* as the previous word, which is less helpful for the prediction of a verb phrase *likes cheese*.

### 3.8 Code Generation with DRNN and Sketching

For code generation task, we use DRNN to model Abstract Syntax Trees. Abstract syntax trees can be automatically acquired using parsers, and can provide valuable syntactic information to the model. This way, explicit syntactic knowledge may be more easily learned by the otherwise grammar-agnostic neural model, without the need for rule-based syntactic constraints such as in Yin and Neubig (2017).

We also incorporate copy mechanism mentioned in §2.6.2. Since abstract syntax trees place all literals on leaf nodes, our copy mechanism is adapted to leaves only. The difference from Equation 2.29 is that the pointer generator now takes DRNN’s predictive hidden state $h_v$ as input instead of decoder hidden state (Equation 3.13).
Figure 3.5: SynC in action. When generating the word \textit{likes} in \textit{Andrei when starving likes cheese}, the prediction will be made knowing that the preceding clause is a noun phrase.

\[ P_{gen}(i) = \sigma(u_{copy} \cdot h_v + b) \]  

(3.13)

We also adopt sketching (§2.6.1) using multi-task learning. Instead of on token level, our sketches are also AST trees with leaf node literals replaced with placeholders.
Chapter 4
Experiment and Analysis

We carried out multiple experiments on machine translation and code generation. We also look at more detailed aspects of our model to provide behavioural analysis of individual modules as well as different hyperparameter settings.

4.1 Natural Language Translation Experiments

4.1.1 Model Training

Translation experiments in this section utilise constituency trees on the target side. Because large bilingual parallel datasets rarely come with constituency information, we have to parse our training dataset prior to training. To that purpose, we use the Stanford Lexical Parser (Chen and Manning, 2014) which we chose for its speed and accuracy.

This procedure of pre-parsing data is not required at test time, our NMT system would take a sentence as input and produces the translation in target language along with its constituency tree as output.

We use the German-English dataset from IWSLT2017 \(^1\) for our experiments, and tst2010-2015 as the test set (Table 4.1).

To compare with other decoders that utilise target-side syntactic information, we also evaluate on three more datasets from News Commentary v8 using newstest2016 as testset (Table 4.2).

We replace all rarely occurring words with UNK (Unknown) tokens. Only the top 50,000 most frequent words are kept. Note this procedure is different for our programming language experiments, in which copy mechanism is used to take care of the rare tokens.

\(^1\)The International Workshop on Spoken Language Translation Evaluation 2017: https://sites.google.com/site/iwsltevaluation2017/TED-tasks
<table>
<thead>
<tr>
<th>Translation pairs</th>
<th>226,572</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique source words</td>
<td>128,857</td>
</tr>
<tr>
<td>Unique target words</td>
<td>61,566</td>
</tr>
<tr>
<td>Average source sentence length</td>
<td>21</td>
</tr>
<tr>
<td>Average target sentence length</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4.1: Dataset information

<table>
<thead>
<tr>
<th>Language pair</th>
<th>DE-EN</th>
<th>CS-EN</th>
<th>RU-EN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train pairs</td>
<td>166,313</td>
<td>134,453</td>
<td>131,492</td>
</tr>
<tr>
<td>Test pairs</td>
<td>2,999</td>
<td>2,999</td>
<td>2,998</td>
</tr>
<tr>
<td>Uniq. src lex</td>
<td>149,318</td>
<td>153,173</td>
<td>159,074</td>
</tr>
<tr>
<td>Uniq. tgt lex</td>
<td>68,415</td>
<td>59,909</td>
<td>64,220</td>
</tr>
<tr>
<td>Avg. src len</td>
<td>25</td>
<td>22</td>
<td>25</td>
</tr>
<tr>
<td>Avg. tgt len</td>
<td>25</td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 4.2: News Commentary v8 dataset information

4.1.2 Modelling Detail

The implementation of all models in this thesis is done using DyNet with Autobatching (Neubig et al., 2017). We use Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) as RNN units. Each LSTM unit has 2 layers, with input and hidden dimension of 256. We use a minibatches of 64 samples. We use early stopping mechanism for all experiments and Adam optimiser (Kingma and Ba, 2015) as trainer.

We use the following acronyms in the results tables:

1. Seq2Seq: standard seq2seq encoder-decoder with attention;
2. Seq2DRNN: BiLSTM encoder + DRNN decoder with attention;
3. Seq2DRNN+SynC: BiLSTM encoder + DRNN decoder with attention and SynC.

For Seq2Seq models, we add an EOS tag at the end of the target sequence to indicate end of translation.

<table>
<thead>
<tr>
<th></th>
<th>BLEU</th>
<th>RIBES</th>
<th>Perplx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>22.83</td>
<td>81.5</td>
<td>1.828</td>
</tr>
<tr>
<td>Seq2DRNN</td>
<td>23.53</td>
<td>80.4</td>
<td>1.644</td>
</tr>
<tr>
<td>Seq2DRNN+SynC</td>
<td><strong>25.36</strong></td>
<td><strong>82.6</strong></td>
<td>1.750</td>
</tr>
</tbody>
</table>

Table 4.3: IWSLT17 translation experiment results
<table>
<thead>
<tr>
<th>Dataset</th>
<th>DE-EN BLEU</th>
<th>DE-EN RIBES</th>
<th>CS-EN BLEU</th>
<th>CS-EN RIBES</th>
<th>RU-EN BLEU</th>
<th>RU-EN RIBES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq</td>
<td>16.61</td>
<td>73.8</td>
<td>11.22</td>
<td>69.6</td>
<td>12.03</td>
<td>69.6</td>
</tr>
<tr>
<td>Str2Tree</td>
<td>16.13</td>
<td>—</td>
<td>11.65</td>
<td>—</td>
<td>11.94</td>
<td>—</td>
</tr>
<tr>
<td>NMT+RNNG</td>
<td>16.41</td>
<td>75.0</td>
<td>12.06</td>
<td>70.4</td>
<td>12.46</td>
<td>71.0</td>
</tr>
<tr>
<td>Seq2DRNN</td>
<td>16.90</td>
<td>75.1</td>
<td>11.84</td>
<td>67.3</td>
<td>12.04</td>
<td>69.7</td>
</tr>
<tr>
<td>Seq2DRNN+SynC</td>
<td>17.21</td>
<td>75.8</td>
<td>12.11</td>
<td>70.3</td>
<td>12.96</td>
<td>71.1</td>
</tr>
</tbody>
</table>

Table 4.4: News Commentary v8 translation experiment results. Seq2Seq and NMT+RNNG are taken from Eriguchi et al. (2017), Str2Tree (string-to-linearised-tree) results (no RIBES scores) come from Aharoni and Goldberg (2017). All numbers reported here are of non-ensemble models.

### 4.1.3 Results

Table 4.3 has the BLEU (Papineni et al., 2002) and RIBES (Isozaki et al., 2010) scores. In our IWSLT2017 tests, both Seq2DRNN and Seq2DRNN+SynC produce better results than the Seq2Seq baseline model in terms of BLEU scores, while Seq2DRNN+SynC also produces better RIBES scores indicating better reordering of phrases in the output. The Seq2DRNN+SynC model performs better than the Seq2DRNN model. Both Seq2Seq and Seq2DRNN+SynC are able to produce results with lower perplexities than the baseline Seq2Seq model on the test data.

In our News Commentary v8 tests, the same relative performance from Seq2DRNN+SynC can be observed. The Seq2DRNN+SynC model is also able to out-perform the Str2Tree model proposed by Aharoni and Goldberg (2017) and NMT+RNNG by Eriguchi et al. (2017) in most cases. Note that Eriguchi et al. (2017) used dependency information instead of constituency information as in this thesis and Aharoni and Goldberg (2017)’s work.

Table 4.5 shows an example translation from all of the models we use in our experiments. Seq2Seq is able to translate with the correct vocabulary, but the sentences are often syntactically awkward. As the sentence length increases the syntactic fluency of Seq2Seq gets worse. Seq2DRNN is able to produce more syntactically fluent sentences since each lowest sub-clause contains typically $\leq 5$ words. Seq2DRNN+SynC produces the best results in this example: produces more syntactically fluent sentences, chooses the right words in the right place more frequently.

We also took several examples from our IWSLT17 experiment and blank out certain nouns by replacing them with unknown tokens (Table 4.6). Note that in our training set, most sentences do not have unknown tokens, and those that do only have at most 1. Our assumptions of the observed patterns in this case are: i) the proposed models are more capable at handling unknown tokens; ii) while Seq2DRNN is more capable at retaining the structure of the sentence, it cannot rely on a wider context to predict certain common phrases with noises in the source sentence; iii) the proposed Seq2DRNN+SynC model is
we repeated this exercise with the same students. now what do you believe happened? now they understand the value of prototyping. so the same terrible team became one of the very best. they produced the tallest construction in the shortest time.

we did the exercise again with the same students. what do you think happened then? so now they understand the value of prototyping. so the same team went from being the very worst to being among the very best. they produced the tallest structures in the least amount of time.

we repeated this with the same students. what happened differently? now, you know, the advantage of the design of the cycle. so, the same one of the team of the team among the best. it produced songs in the slightest building.

we will repeat these queries with the same students. what do you think of this? now it understood the advantage of the interests. that’s been made of the same thing of one thing. they produced the highest construction of the best time at the best time.

we repeated this practice with the same students. what do you think happened? now, they understood the value of prototyping. it was being made of the same thing of one of the best ones. they produced the highest construction in the best time.

Table 4.5: Translation samples. **Gold** is the reference, and **Literal** is produced by a bilingual German-English speaker.

more capable at handling unknown words both in the sense of being better at retaining sentence structure and handling noisy input.

### 4.1.4 Attention Module

We visualise the attention weights of our Seq2DRNN+SynC model. Attention §3.6 computes a context vector for each node in the tree (a weighted sum of the source side vector representations). For the translation pair in Figure 3.3, we show the attention weight of each pair of word and node (Equation 3.8) in Figure 4.1.

The addition of syntax nodes in the output enables the attention model to be used more effectively and is also valuable for visual inspection of syntactic nodes in the output mapping to the input.
Table 4.6: Unknown noun experiment samples. Substituted and correct nouns are marked in **Bold**, while incorrect elements are marked in underline. Examples shown are: no UNK; 1 UNK; 2 UNKs; 3 UNKs. When there are no unknown tokens, all three compared models are able to produce reasonably good if not identical translations. When there is only one UNK token, Seq2DRNN often does not use the context to predict an appropriate word or phrase. In contrast, both the Seq2Seq and Seq2DRNN+SynC were able to correctly predict that *die gute UNK ist* could be translated to the *good news is*. When there are 2 UNK tokens in the source sentence, Seq2Seq produces more incorrect predictions, Seq2DRNN makes some mistakes, while Seq2DRNN+SynC is able to get the most parts correct. Finally, when we replace 3 nouns, all models fail to some degree while Seq2Seq’s output is the worst.
Figure 4.1: Seq2DRNN+SynC attention visualisation. “ich” means “I”, “bin” is “am”, “doktor”’s literal translation is “doctor”. Darker colour means higher weight (relevance score) as calculated in Equation 3.8. The values of each column sum up to 1. The attention weights in this example perfectly align with the appropriate clauses. Additional example is provided in the appendix.

4.1.5 Parsing Quality

To evaluate the parsing quality, we follow the approach by Vinyals et al. (2015b) and train a DRNN(SynC) model to produce English to English(Tree) translation. We use the same data and experiment settings that Vinyals et al. (2015b) used: the Wall Street Journal Penn Treebank English corpus with golden constituency structure, 256 for input/hidden dimension and 3 layers of RNN. We evaluate on section 23 of the aforementioned WSJ data using EVALB\(^2\). The results are presented in Table 4.7.

<table>
<thead>
<tr>
<th>Model</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (Vinyals et al., 2015b)</td>
<td>&lt; 70</td>
</tr>
<tr>
<td>LSTM+AD (Vinyals et al., 2015b)</td>
<td>88.3</td>
</tr>
<tr>
<td>Petrov (2010)</td>
<td>91.8</td>
</tr>
<tr>
<td>Dyer et al. (2016)</td>
<td>92.4</td>
</tr>
<tr>
<td>Seq2DRNN</td>
<td>89.4</td>
</tr>
<tr>
<td>Seq2DRNN+SynC</td>
<td>89.9</td>
</tr>
</tbody>
</table>

Table 4.7: Parser scores. Numbers from (Vinyals et al., 2015b) are of non-ensemble models.

Although falling short behind purposefully designed dedicated models like RNNG by Dyer et al. (2016), our model is able to produce better results than LSTM+AD by Vinyals

\(^2\)https://nlp.cs.nyu.edu/evalb/
et al. (2015b), which is more comparable for it is also an NMT model in nature but used to perform constituency parsing. Since for natural language translation, we only do constituency parsing as an auxiliary task, optimising and designing a better parser is not our priority. Nevertheless, it is worth noting that in 50.89% of the cases, Seq2DRNN+SynC was able to produce output that perfectly matches the reference. The same number for sentences with less than 40 words is 52.16%, while the F-measure increases to 90.5. This shows Seq2DRNN(SynC) when doing parsing can produce outputs of similar quality when handling longer sentences.

We also do evaluation on our translation results from the IWSLT dataset. Since translation results do not come with reference parse trees, we parse the output of our decoder using the same parser we used in our other experiments: the Stanford Parser. Constituency parsing evaluation is done using Precision/Recall/F1-scores on the output constituent spans (unlabelled) and spans and labels (labelled). The results are presented in Table 4.8. The parser we use gets F1 score of 87.04 on Penn Treebank English constituency parsing (Klein and Manning, 2003).

<table>
<thead>
<tr>
<th></th>
<th>Unlabelled</th>
<th>Unlabelled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec.</td>
<td>Rec.</td>
</tr>
<tr>
<td>Seq2DRNN</td>
<td>96.87</td>
<td>96.93</td>
</tr>
<tr>
<td>Seq2DRNN+SynC</td>
<td>96.43</td>
<td>95.89</td>
</tr>
</tbody>
</table>

Table 4.8: IWSLT translation result constituency scores. Reference parse trees obtained using Stanford Parser.

The presence of SynC in the decoder influences parse tree construction: the Seq2DRNN+-SynC F1 score is comparable but lower than Seq2DRNN.

4.2 Code Generation Experiments

In code generation task, we use django dataset (Oda et al., 2015), a parallel corpus with English as source and Python with Django library as target. We use the split version published by Dong and Lapata (2018) for fair comparison. It contains 16,000 training, 1,000 dev, and 1,805 test samples, covering a wide range of cases including string manipulation, iteration, exception handling, etc.
## Table 4.9: Results on Django dataset. Block 1 is taken from (Yin and Neubig, 2017).

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval System</td>
<td>14.7</td>
</tr>
<tr>
<td>Phrasal SMT</td>
<td>31.5</td>
</tr>
<tr>
<td>Hierarchical SMT</td>
<td>9.5</td>
</tr>
<tr>
<td>SNM+COPY (Yin and Neubig, 2017)</td>
<td>71.6</td>
</tr>
<tr>
<td>COARSE2FINE (Dong and Lapata, 2018)</td>
<td>74.1</td>
</tr>
<tr>
<td>Seq2DRNN+SynC</td>
<td>25.0</td>
</tr>
<tr>
<td>Seq2DRNN+SynC+Sketch</td>
<td>26.0</td>
</tr>
<tr>
<td>Seq2DRNN+SynC+Copy</td>
<td>73.1</td>
</tr>
<tr>
<td>Seq2DRNN+SynC+Copy+Sketch</td>
<td>76.2</td>
</tr>
</tbody>
</table>

We use the same preprocessing steps as Dong and Lapata (2018). AST Sketches are also generated in the same way as Dong and Lapata (2018): keeping the same set of keywords and replacing all other literals with placeholders. All Seq2DRNN+() models first generate the ASTs; then the final output is compiled by converting the ASTs to code. For parsing the code and reconstructing from ASTs, we use Python’s AST and ASTOR package.

For evaluation, we use exact match accuracy instead of BLEU score: a sample in its entirety is considered correct only if it matches the reference perfectly. To make sure there is no impact on evaluation scores from tokenisation, we normalise all code by converting them to ASTs and then exporting back to code form. The results are in Table 4.9.

It is important to note that for all Seq2DRNN+() experiments, since we generate the ASTs first then convert it back to code using Python’s AST package, there is a chance that the model may output ill-formed convertible trees. The percentage of that happening on the test set is just 1.11% (20 out of 1805), and as shown in Table 4.9, our model still reaches a new state-of-the-art above existing models. We have tried to use Seq2Seq with Copy and Sketching as fallback when our best model’s output tree appears convertible, but the accuracy didn’t change.

### 4.3 Trainer Experiments

Experiments were also carried out to examine the effectiveness of different trainers. We looked at Stochastic Gradient Descent with decreasing learning rate(SGD), Momentum Stochastic Gradient Descent(momentum SGD), and Adam. The experiment are performed using our LSTM decoder, with 64 input dimensions and 128 hidden dimensions. The encoder has 2 layers and the decoder has 1 layer. The dataset using which we run the experiment is IWSLT2017 German-English.
For SGD, we start the training at learning rate 1.0 decrease it by 50% at each epoch. With this approach, the gradient quickly vanished after the fifth epoch.

The comparison of momentum SGD and Adam are presented in Figure 4.2 and 4.3. The initial learning rate for Adam is the default value in DyNet 0.001. The initial learning rate for momentum SGD is the default value in DyNet 0.01.

As the figures show, both training loss and validation loss decrease much faster in Adam than in momentum SGD, and Adam is able to reach minimum validation loss quicker. Momentum SGD’s validation loss although kept decreasing for a few more epochs beyond the thirteenth (which is where Adam terminates), but its’ final minimum validation loss was still higher than Adam’s.
Chapter 5

Related Work and Conclusion

5.1 Related Work

Recent research shows that modelling syntax is useful for various neural models of natural language tasks. Dyer et al. (2015, 2016); Vinyals et al. (2015b); Luong et al. (2016) have works on language modelling and parsing, Tai et al. (2015) on semantic analysis, and Zhang et al. (2015) on sentence completion, etc.

Eriguchi et al. (2017) showed that NMT model can benefit from neural syntactic parsing-based models. In addition, Choe and Charniak (2016) showed that a neural parsing problem share similarity to neural language modelling problem, which forms a building block of an NMT system. We can then make the assumption that structural syntactic information utilised in neural parsing models should be able to aid NMT, which is shown to be true by our experiments.

Zhang et al. (2015) proposed TreeLSTM which is another structured neural decoder. TreeLSTM is not only structurally more complicated but also uses external classifiers. Dong and Lapata (2016) also proposed a sequence-to-tree (Seq2Tree) model for question answering. Both of these models are not designed for NMT and lack a language model. While operate from top-to-bottom like Seq2DRNN(+SynC), TreeLSTM and Seq2Tree produce components that lack sequential continuity which we have shown to be non-negligible for language generation.

Aharoni and Goldberg (2017) and Eriguchi et al. (2017) experimented NMT models that utilises target side structural syntax. Aharoni and Goldberg (2017) treat constituency trees as sequential strings (linearised-tree) and trains a Seq2Seq model to produce such sequences. Eriguchi et al. (2017) proposed NMT+RNNG which in its nature is still a sequential decoder, with additional components including an action RNN unit similar to a shift-reduce parser to provide an extra layer on top of a conventional RNN decoder. Both research show target side syntax can improve NMT systems, but both use word-level language models and are essentially sequential decoder models. Both models can potentially benefit from SynC (with
NMT+RNNG one has to instead use constituency information). Such investigation is yet to be conducted.

For code generation related work, Yin and Neubig (2017) describes a translation model for general-purpose programming languages using rule-based decoder constraints. With other Domain Specific Languages (DSL) like SQL, syntactic rules are often directly modelled into the design of the models (Ling et al., 2016; Iyer et al., 2017; Yin and Neubig, 2017; Iyer et al., 2018). Murali et al. (2017) propose to map a collection of labels into program sketches using a structured decoder. Misra et al. (2018) uses reinforcement learning and rules to compose SQL queries sequentially, and execute those queries on a table to retrieve answers. Recent endeavours have also seen massive code generation datasets being automatically mined from online platforms (Yin et al., 2018), the application of which is yet to be further investigated.

5.2 Conclusions and Future Work

We propose an translation model that exploits target side structural syntax with a strictly top-down tree-structured decoder called Doubly Recurrent Neural Networks (DRNN). We incorporate DRNN into an encoder-decoder NMT architecture, coupled with attention on tree and our novel syntactic connections on tree structures. Our experiments show that our proposed models can outperform strong sequential NMT baseline on natural language translation tasks, and reach a new state-of-the-art on Django code generation task. Our tree-structured decoder can also produce syntactically valid parse trees when compared against a highly trained syntactic parser output. It can also automatically learn programming language syntax, without having to resort to rule-based constraints like previous approaches.

In the future we hope to incorporate source side syntax information into the model. We also plan to explore the applications of SynC in the area of neural machine translation with more structured attention mechanisms, and even potentially a hybrid phrase-based NMT systems with SynC, in which the model can benefit from SynC to be more extensible when handling larger vocabulary sizes.

For code generation, the lack of high quality corpora with scale comparable to natural language corpora remains a problem. A more detailed study on the effects of sketching can also help us understand the limits of current models.
Bibliography


