UNIVERSITAS INDONESIA

SEMANTIC ROLE LABELING IN INDONESIAN CONVERSATIONAL LANGUAGE

SKRIPSI

VALDI RACHMAN
1306381862

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
JUNI 2017

UNIVERSITAS INDONESIA

SEMANTIC ROLE LABELING IN INDONESIAN CONVERSATIONAL LANGUAGE

SKRIPSI
Diajukan sebagai salah satu syarat untuk memperoleh gelar
Sarjana Ilmu Komputer

VALDI RACHMAN
1306381862

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
JUNI 2017

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri,
dan semua sumber baik yang dikutip maupun dirujuk
telah saya nyatakan dengan benar.

Nama	:	**Valdi Rachman**
NPM	:	**1306381862**
Tanda Tangan	:	
Tanggal	:	**5 Juni 2017**

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh	:	
Nama	:	Valdi Rachman
NPM	:	1306381862
Program Studi	:	Ilmu Komputer
Judul Skripsi	:	Semantic Role Labeling in Indonesian Conversational Language

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Ilmu Komputer pada Program Studi Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing 1	:	Rahmad Mahendra	(	)
Pembimbing 2	:	Alfan Farizki Wicaksono	(	)
Penguji	:	-	(	)
Penguji	:	-	(	)

Ditetapkan di	:	Depok
Tanggal	:	-

PREFACE

First and foremost, I would like to thank those who support and contribute in the process of writing this report, including:

1. Rahmad Mahendra, S.Kom., M.Sc and Alfan Farizki Wicaksono S.T., M.Sc. as the supervisors from Universitas Indonesia who endlessly give inputs and ideas for this research.
2. Ahmad Rizqi Meydiarso as the supervisors from Kata.ai who always supports and discusses ideas for this research.
3. All participants of Information Retrieval Lab as a team which always support each other.
4. My family: Achmad Friscantono, Dian Warasati, and Neysha Adzhani.
5. My friends: Citra Dara Malia, Ilham Feti, Febriyola Tambunan, and all Fasilkom batch 2013 (Angklung).

Depok, Juni 2017

Valdi Rachman

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Valdi Rachman
NPM	:	1306381862
Program Studi	:	Ilmu Komputer
Fakultas	:	Ilmu Komputer
Jenis Karya	:	Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty Free Right*)** atas karya ilmiah saya yang berjudul:

Semantic Role Labeling in Indonesian Conversational Language

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyatan ini saya buat dengan sebenarnya.

Dibuat di : Depok
Pada tanggal : 5 Juni 2017

Yang menyatakan

(Valdi Rachman)

ABSTRACT

Name : Valdi Rachman
Program : Computer Science

Title : Semantic Role Labeling in Indonesian Conversational Language

Semantic Role Labeling (SRL) has been extensively studied, mostly for understanding English formal language. However, only a few reports exist for informal conversational language, especially for language being used in the chatbot system. In Indonesian, both formal and conversational language are barely tapped for building SRL system. In this work, we focus on solving SRL on Indonesian conversational language. Our contributions are proposing a new set of semantic roles and an attention mechanism on top of Long Short-Term Memory Networks architecture. The challenges of Indonesian conversational language include a wide variety of slangs and abbreviations, short sentences, as well as disorganized grammars. Although this is a pilot task, we obtained a really promising result with F1 score of 82.68%.

Keywords:
Semantic Role Labeling, deep learning, conversational language, RNNs

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PSEUDOCODES

CHAPTER 1
INTRODUCTION

1.1 Background

Semantic Role Labeling (SRL) is a task in Natural Language Processing (NLP) which aims to automatically assign semantic roles to each argument for each predicate in a given input sentence. As for a brief definition, given an input sentence, SRL system will give an output of *"Who did what to whom"* with *what* as the predicate and *who* and *whom* being the argument of the predicate. SRL is an integral part of understanding natural language as it helps machine to retrieve semantic information from the input. In practice, SRL has been widely used as one of the intermediate steps for many NLP tasks, some of which are information extraction (Emanuele et al., 2013; Surdeanu et al., 2003), machine translation (Liu and Gildea, 2010; Lo et al., 2013), question-answering (Shen and Lapata, 2007; Moschitti et al., 2003).

In the chat bot industry, the bots need to understand semantic information of the user's text in order to generate more personalized response. To illustrate, suppose that the user send a text chat to the bot and the SRL system extracts the semantic roles as presented bellow.

Input: *"I just ate chicken rice! Haha"*
Roles:
Predicate: *eat*
Agent: *I*
Patient: *chicken rice*

By knowing that the user just ate a chicken rice, the bot can thus response with *"That's great! how was the chicken?"*. This way, the user will be more engaged to the conversation with the bot.

SRL has been extensively studied for English formal language. Most of the traditional SRL systems are built based on language-dependent features such as syntactic parsers (Gildea and Jurafsky, 2002; Gildea and Palmer, 2002; Pradhan et al., 2005). This syntactic information plays a pivotal role in solving SRL problem for traditional systems as it addresses SRL's long distance dependency.

Unfortunately, this approach hardly depends on the linguistic experts experience assigning the correct syntactic information to the training data, which is costly. Moreover, if we want to build such system for another language, we have to define the syntactic information all over again. In order to address such problem, Jie Zhou et al. proposed an end-to-end learning of SRL using Recurrent Neural Networks (RNN) (Zhou and Xu, 2015). In their research, they used Deep Bi-Directional Long Short-Term Memory (DB-LSTM) as the approach for RNN. The advantage of their system is that it only needs words of sentences as the input feature, and does not need any syntactic parsing since LSTM approach addresses the long distance relationship property of SRL problem. The research result outperforms the previous state-of-the-art traditional SRL systems as it achieved F1 score of 81,07%. Other works involving deep learning for SRL are done by Collobert et al. and Foland Jr. et al.

On the other hand, the number of research focusing on SRL for Indonesian language (next, will be called as *Indonesian*) is still low. One example would be a research done by Dewi [x], which proposed SRL for Indonesian using Support Vector Machine. The research done by Dewi used the TreeBank data (in English) translated to Indonesian language using Google Translate. The research result opens a window of improvement as the best result consists of 61,6% precision and 66,8% recall. Another work focusing on SRL for Indonesian was done by Nur Indrawati et al., which used case grammar theory for SRL. However, the research concludes that not all sentences could be labeled as it did not cover all types of verbs in Indonesian. Moreover, little number of instances is used as the test data showing that the data set is relatively small. These facts open an opportunity to explore a more robust approach for Indonesian SRL as well as the need for a more reliable data.

This research is supported by Kata.ai, a technology company focusing on Artificial Intelligence (AI) and NLP development in Indonesian language. Its goal is to empower businesses by leveraging the power of AI and NLP towards customer engagement in a form of chat-bot. In order to achieve it, there has been an ongoing research project by the company focusing on Indonesian NLP. Since it uses chatting platforms as the medium, the scope of the project is for Indonesian conversational language. Conversational language is the most natural way people use to communicate in their daily life and thus, it is interesting to understand the language better through SRL.

Telling from the characteristics, conversational Indonesian language has its own challenges. It has many slang words for daily conversations. For example, the verb 'belikan' ('buy') has its informal form which is 'beliin'. Another example would

be 'berbicara' ('talk') as 'ngobrol'. It happens to many words in Indonesian. Not to mention the variety of ways to express pronoun 'aku' ('I') such as 'gw', 'gue', and 'aq'. Yet, they have many kinds of interjection such as 'eh', 'duuh', 'dong', 'kok' which complexify the sentence structure. These are the challenges that the SRL system should tackle in dealing with conversational language.

Based on the fact that we still lack of Indonesian SRL research, it is an interesting opportunity to build SRL system for Indonesian. Our main contribution in this work would be applying SRL to Indonesian conversational language. We will deep dive into the semantic role characteristics found in the language. After that, we will use deep learning as the state-of-the-art approach that has been emerging in NLP field for doing the SRL task. There is a wide variety of features and model architectures that can be used to train the SRL model. It is thus important for us to find the best feature combinations and model architecture for solving SRL in Indonesian conversational language.

1.2 Problem Statement

Based on the motivation described in the background, we therefore propose following problem statements:

1. How to solve SRL for Indonesian conversational language with sequence labeling approach using deep learning?
2. Which feature combination outputs the best performance?
3. Which deep learning model architecture gives the best result?

1.3 Objectives and Contributions

This research aims to build an SRL system for Indonesian conversational language. The main contributions of this work include creating a new set of semantic roles for Indonesian conversational language as well as proposing features and architectures for solving the SRL problem.

1.4 Methodology

The methodology of this work consists of literature review, data gathering, model development, experiment, evaluations and analysis, and conclusion.

1. Literature Review

   In this step, we did a comprehensive study on Natural Language Processing (NLP) and Machine Learning (ML) aspects. The NLP aspect includes language model and semantic role labeling. For machine learning, we learned deep learning approach such as recurrent neural networks and convolutional neural networks. These knowledge are the basis to support our research

2. Data Gathering

   Since there seems to be no available corpus for SRL on Indonesian, especially conversational language, we therefore annotated our own corpus. We retrieved the real word data from one of Kata.ai's chat bots. For this annotation, we build a new set of semantic roles crafted for Indonesian conversational language.

3. Model Development

   After we gathered our corpus, we then design the model for the experiment in this research. We define the feature extractions and the deep learning model architecture that will be tested. In this section, we also propose our own architecture.

4. Experiment

   In this step, we design our experiment scenarios in order to answer the questions being asked in the /perumusan masalah/. There are two set of scenarios consisting of feature and architecture experiments. The first one aims to find which feature combination outputs the best result, meanwhile the later focuses on comparing deep learning architecture models.

5. Evaluation and Analysis

   The experiment results are then to be evaluated and analyzed. We use precision, recall, and F1 as the metrics for the evaluation. We also conduct error analysis to get a deeper insight on the results.

6. Conclusion

   In the end, we conclude our findings in our research based on the evaluations and analyses of the experiments. We then describe some future works that can be done following the results of this research.

1.5 Organization

This report is organized as follows. In chapter 2, literature review on the language models, deep learning architectures, and semantic role labeling will be shown. The methodology of this research is described in chapter 3, including the features and the model architectures being used. This chapter will also provide the new idea of contribution for this research. In chapter 4, the implementations of our methodology are described with the details of tools used and the measurement setup. The experiment scenarios, results, and analyses are presented in chapter 5. Lastly, the conclusion and possible future works are described in chapter 6.

CHAPTER 2
LITERATURE REVIEW

This chapter focuses on literature study on three aspects including language models, deep learning, and semantic role labeling. In language model section, Part-of-Speech tag (POS tag) and word embedding are described. Deep learning section focuses on the architecture widely used for sequence labeling problem. Finally, we explain semantic role labeling in the last section, including the semantic roles definition, annotation corpus, problem definitions, common features, and the historical perspectives.

2.1 Language Models

This section explains the language models usually used in Natural Language Processing (NLP) applications. We first describe the traditional yet important language model, Part-of-Speech tag (POS tag), followed by the so-called word embedding that is often used in recent NLP systems.

2.1.1 Part-of-Speech Tag (POS Tag)

Part-of-Speech (POS) tag defines the class of a word. Some examples of POS tag are noun, verb, adverb, and adjective. POS tags are useful because of the large amount of information they give about a word and its neighbors (Jurafsky and James, 2016). For instance, knowing whether a word is a noun tells us about likely neighboring words since nouns are usually preceded by determiners or adjectives. It also tells us about the syntactic structure around the word as nouns are generally part of noun phrases. Table 2.1 shows an example of a sentence and the POS tags for each word. PRP, VBP, DET, and NN refer to personal pronoun, present verb, determiner, and singular noun, respectively.

Table 2.1: An example of a sentence and the POS tags for each word

Sentence	I	eat	a	chicken	rice
POS tag	PRP	VBP	DET	NN	NN

The computational methods for assigning POS tags to words is called POS tagging (Jurafsky and James, 2016). POS tagging can be trained using supervised

approaches such as Hidden Markov Model and the Maximum Entropy Markov Model (Jurafsky and James, 2016).

2.1.2 Word Embedding

Word representation is an important feature when one wants to build deep learning model for NLP tasks. The idea is to convert words into vectors. There are two approaches for this vector representation, which are traditional and word embedding approach. Traditional approach uses one-hot vectors for the representation, meanwhile word embedding approach uses real values vectors that contain information about the words.

In the traditional approach, the vectors are retrieved based on the index of the word found in the dictionary. The dictionary consists of the word and its index. Suppose that we have four words: *I, eat, chicken, you*. Each of these words has their own index, with *I*:0, *eat*:1, *chicken*:2, *you*:3. These indices will represent the one-hot vectors for the words. For instance, word with index 0 has a one-hot vector [1, 0, 0, 0], word with index 1 has a one-hot vector [0, 1, 0, 0], and so on. The length of the vector is determined by the size of our dictionary. In this case, the size of our dictionary is 4, hence the length of the vector is also 4.

This approach, however, has a drawback since the vector representation is sparse. As we just give the index to the all the words based on the dictionary, it does not really represent an important information from the words. For instance, the word *chicken* and *beef*, though have similarity as they are eatable, can be represented by two far indices, say 1 and 100. This representation therefore does not capture the similarity between those words.

To address this issue, there is another better word representation approach: the so-called word embedding. Word embedding is designed to represent word with a dense, low dimension, real-values vector. For example, the word *chicken* is mapped into a vector [0.28, 0.31, -0.17, ..., 0.89]. With this representation, word embedding transforms similar words to similar vectors. From the previous example, the word *chicken* and *beef* will most likely have vectors that are close to each other.

There has been a lot of research on word embedding, such as Word2Vec (Mikolov et al., 2014) and Glove (Pennington et al., 2014). In this section, we only explain Word2Vec since we will use this later in our research. Word2Vec uses unsupervised approach so that we only need a lot of unlabeled data for building word embedding model (Mikolov et al., 2013). Basically, Word2Vec uses neural networks architecture to train the unlabeled data. Word2Vec is able to learn and classify words based on their similarity and represent it in the vector

space.

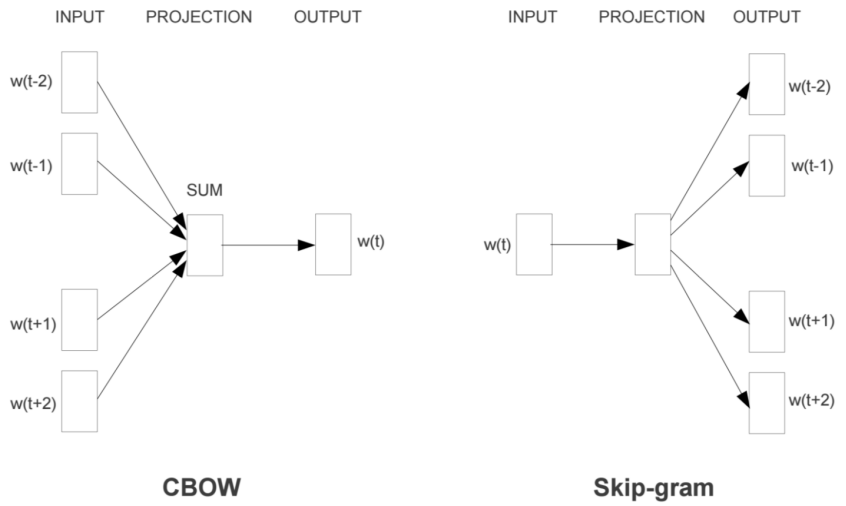

Figure 2.1: CBOW and Skip-gram architectures (Mikolov et al., 2013)

Word2Vec has two architectures, namely Continuous Bags of Words (CBOW) and Skip-gram (Mikolov et al., 2013), as illustrated in Figure 2.1. In CBOW, the model learns to predict a word based on its neighboring words. Therefore, the input layer is represented with the bag-of-words. In contrast, Skip-gram architecture aims to predict the neighboring words based on a given word. The advantage of CBOW is that it can be used for training a huge amount of data, while Skipgram is best for capturing the average co-occurance from two words from the data.

Both architectures mainly aims to build language model, however, one does not need the whole trained model for having the word embedding representation. Instead, we only need to extract the weight matrix used for converting words into vectors from the model. This weight matrix is the word embedding model that we use for transforming the words into the dense vectors.

2.2 Deep Learning

Deep learning is a branch of machine learning that has multiple layers inside the model. Deep learning is able to extract implicit features in a high, abstract level (LeCun et al., 2015). Deep learning model has proved to produce robust performance in a variety of reserach, including computer vision and natural language processing.

Deep learning is basically a neural networks model with deeper hidden units. Neural networks model is based on how the neuron works inside human brain.

Neurons with deep hidden units are then able to extract features in a abstract level (Bengio et al., 2007). The deep learning structure consists of input layer, hidden layer, and output layer. The input layer is where the data being fed into the model, while output layer is the result of the model. The important layer here is the hidden layer in which a linear and/or non-linear functions are approximated in order to get the best predicted outputs.

Deep learning model has proved to give outstanding performances in supervised learning (Goodfellow et al., 2016). A model with deeper layer will learn more implicit features out of the training data. There are a lot of deep learning models that have been proposed, some of which are Recurrent Neural Networks (Elman, 1990) and Convolutional Neural Networks. Each of the deep learning models is designed to fulfill specific computation needs.

2.2.1 Recurrent Neural Networks

Recurrent Neural Networks, shortened as RNN, is one of deep learning models designed for processing sequential data. There are some varieties of RNN, including the one proposed by Elman (1990) and Jordan (1986). Since it is designed for processing sequential data, it has a nature advantage for modeling the sequence labeling problem. Suppose that we have sequence of inputs, RNN will take each input in a time step t to process it in a function. Figure 2.2 shows a general RNN.

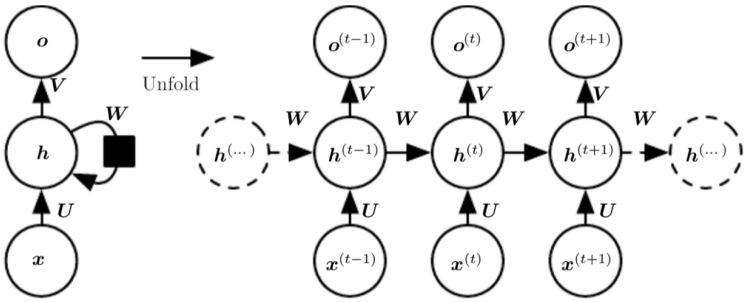

Figure 2.2: A simple Recurrent Neural Networks (RNN) (Goodfellow et al., 2016). (left) folded RNN. (right) unfolded RNN

The left picture illustrates the folded RNN model applied to all time steps. Note that the black rectangle represents one time step delay, meaning that that input is coming from the output of the previous time step.

The right picture shows the unfolded RNN that is more intuitive since it visualizes the time steps. There are three layers in every time step t, which are input, hidden, and output layers. The input layer is for the input representations. In the hidden layer, it contains information from the input layer as well as those coming

from hidden layers in the previous time steps. The output layer consists of the output of the model. These three layers are in a form of vectors. In every time step t, RNN has an input layer $x(\vec{t}) \in \mathbb{R}^{A}$, hidden layer $h(\vec{t}) \in \mathbb{R}^{H}$, and output layer $o(\vec{t}) \in \mathbb{R}^{B}$. The values of A, H, and B represent the length of the input vector, the number of unit in a hidden layer, and the length of the output vector, respectively. There are three parameters that will be trained, which are U, V, and W. These parameters are the weight matrices for connecting two layers. $U \in \mathbb{R}^{H \times A}$ connects input layer with hidden layer (input-hidden), $W \in \mathbb{R}^{H \times H}$ connects hidden layer with the previous hidden layer (hidden-hidden), and $V \in \mathbb{R}^{B \times H}$ connects hidden layer with output layer (hidden-ouput). These parameters are time-distributed, meaning that they are shared across time steps.

Every input layer $x(\vec{t})$ is mapped into output layer $o(\vec{t})$ in every time step t. In the middle of the process, the model calculates the hidden layer $h(\vec{t})$ from two layers, $x(\vec{t})$ and $h(\vec{t-1})$. The output layer $o(\vec{t})$ then is retrieved by performing a function to the hidden layer h(t). The general equations for RNN are presented as follows:

$$o(\vec{t}) = f2(V \cdot h(\vec{t}) + \vec{c}) \tag{2.1}$$

$$h(\vec{t}) = f1(U \cdot x(\vec{t}) + W \cdot h(\vec{t-1}) + \vec{b}) \tag{2.2}$$

Where $h(\vec{0}) = f1(U \cdot x(\vec{0}))$.

Note that there are two additional parameters to train, which are the bias vectors $\vec{b}$ and $\vec{c}$. In Equation 2.2, the input $x(\vec{t})$ and $h(\vec{t-1})$ are weighted by matrices U and W respectively, added by a bias vector $\vec{b}$. The result is then inserted to an activation function $f1$ in order to produce hidden layer $h(\vec{t})$. In the Equation 2.1, $h(\vec{t})$ is multiplied by the weight matrix V and added by a bias vector $\vec{c}$ before being processed by the activation function $f2$ to produce $o(\vec{t})$. The examples of activation function $f1$ and $f2$ are tanh and softmax.

Based on this illustration, there are two main characteristics of RNN:

1. It has a cycle in the graph for every time step. Hidden layer $h(\vec{t-1})$ will be one of the inputs for forming $h(\vec{t})$.

2. It has shared parameters across time steps.

Figure 2.3 illustrates a more complete RNN model on how it is being trained.

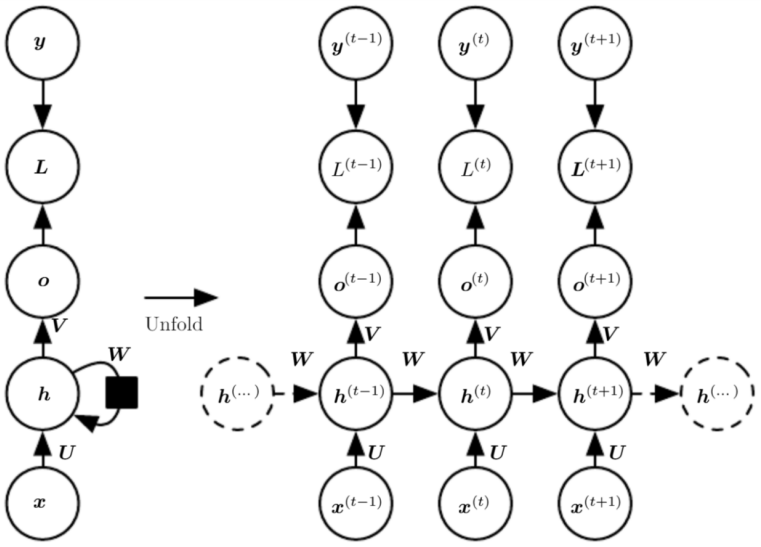

Figure 2.3: A Complete RNN on how it is being trained (Goodfellow et al., 2016). (left) folded RNN. (right) unfolded RNN

The goal of training the model is to find the estimated values of parameters W, U, V, $\vec{b}$, and $\vec{c}$ which produce output $o(\vec{t})$ as close as the expected output $y(\vec{t})$ in the training data.

The loss function L measures the difference between the predicted output $o(\vec{t})$ and the expected output $y(\vec{t})$ in every time step t. The smaller the difference, the better the model. The machine thus has to minimize the result of loss function as small as possible. The parameters W, U, V, $\vec{b}$, and $\vec{c}$ are unknown in the beginning. At first, these parameters are initiated randomly. For every iteration, called epoch, the machine aims to learn the best values for each parameter.

The way to do so is by computing the gradient for each iteration. The idea behind computing the gradient values is to show us which parameter setting that brings us into smaller loss function result. By having this information, the machine then sets the better values for each parameter in the next iteration in order to reduce the loss function. From one iteration into another, the machine will find better parameter values to minimize the loss function. The learning method based on the gradient information is called optimization algorithm. Some optimization algorithms available are Stochastic Gradient Descent (), Adam (Kingma and Ba, 2014), and RMSProp (Hinton, 2012).

2.2.2 Long Short-Term Memories

Regular RNN has an issue called vanishing and exploding gradient problem. The RNN architecture repeatedly uses the same parameters for each time steps. Suppose

that we use W as the parameter for each time step between the hidden units. After t time steps, the matrix would be multiplied t times, hence it is the same as multiplying the hidden units with Wt. Assuming that W has an eigen-decomposition $W = X \cdot diag(\lambda) \cdot X^{-1}$, W^t is equal to:

$$W^t = (X \cdot diag(\lambda) \cdot X^{-1})^t = (X \cdot diag(\lambda)^t \cdot X^t) \tag{2.3}$$

The eigenvalues λ in $diag(\lambda)$ will either vanish if they are less than 1 in magnitude or explode if they are greater than 1 in magnitude. The gradient counted in each time step is aligned with the eigenvalues. Hence, the gradient may also vanish or explode. This is what we called as vanishing and exploding gradient problem. When the gradient vanishes, it is hard for the machine to find the direction to reduce the cost function. In the case of exploding gradient, the learning algorithm will become unstable.

To address this issue, there are solutions proposed such as leaky units (Mozer, 1992), simulated annealing and discrete error propagation (Bengio et al., 1994), time delays (Lang et al., 1990), and hierarchical sequence compression (Schmidhuber et al., 2007). Among these approach, one of the most robust solutions is the Long Short Term Memories (LSTM) (Hochreiter et. al., 1997).

The modification added in LSTM to address the issue is by using gates. It is basically RNN, but the nonlinear units in the hidden layer is replaced by the memory blocks. One nonlinear unit *tanh* in RNN is replaced by more complex memory blocks in LSTM. Besides the hidden layer $\vec{h(t)}$, LSTM also has $\vec{m(t)}$ which is called memory cells. The idea of LSTM is to learn when to forget or remember the memory from previous time steps through multiplicative gates. It thus prevents the vanishing and exploding gradient problem. For example, if the input gate is closed, then the memory will be unchanged.

Figure 2.4 illustrates a one block memory in LSTM. There are three main gates, which are forget gate, input gate, and output gate. These gates are responsible to determine whether an information is added, kept, or deleted in a cell. Each gate has sigmoid layer and element-wise operations. The sigmoid layer converts the input into a probability between 0 and 1. This probability describes the gate behavior towards the input, whether to accept it (probability close to 1) or not (probability close to 0).

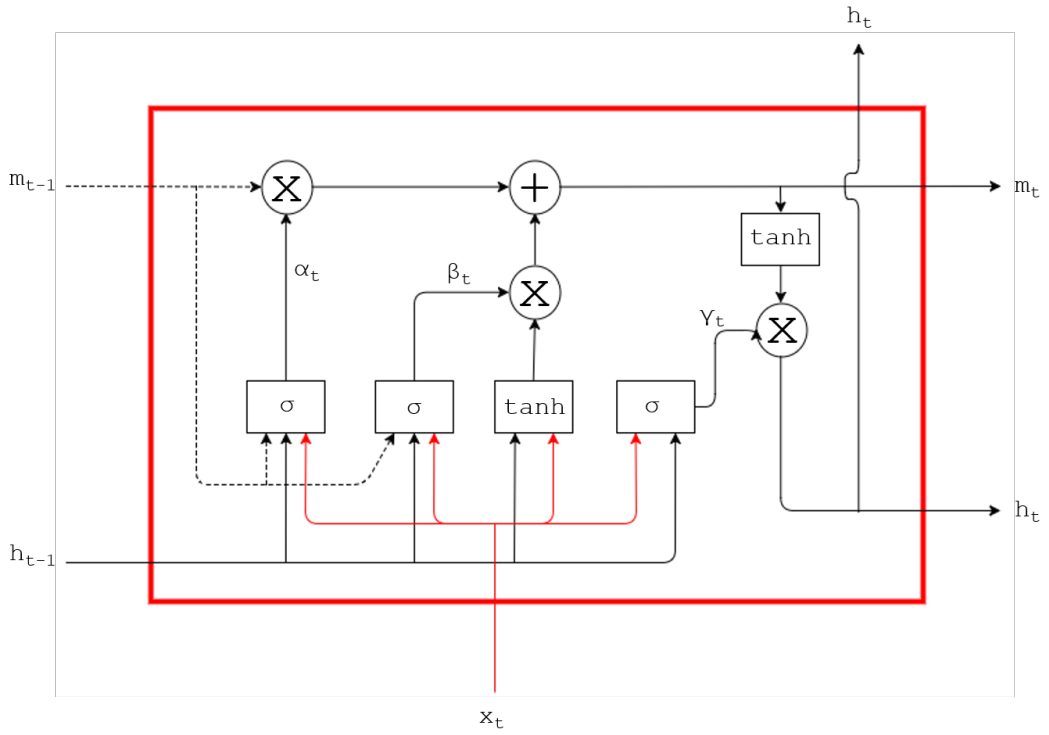

Figure 2.4: One memory block in LSTM (Rohman, 2017)

The equations of the sigmoid layers for each of the gates are explained as follows:

1. *Forget Gate*
   This gate is responsible to determine how much the information from the past should be kept in the memory. The equation of the forget gate is given as follows:
   $$\alpha_t = \sigma(W_{x\alpha} \cdot x_t + W_{h\alpha} \cdot h_{t-1} + W_{m\alpha} \cdot m_{t-1}) \tag{2.4}$$

2. *Input Gate*
   This gate is responsible to determine how much the current information x(t) should be kept in the memory. The equation of the input gate is given as follows:
   $$\beta_t = \sigma(W_{x\beta} \cdot x_t + W_{h\beta} \cdot h_{t-1} + W_{m\beta} \cdot m_{t-1}) \tag{2.5}$$

3. *Output Gate*
   This gate is responsible to determine the output of a time step based on current cell state. The equation of the output gate is given as follows:
   $$\gamma_t = \sigma(W_{x\gamma} \cdot x_t + W_{h\gamma} \cdot h_{t-1} + W_{m\gamma} \cdot m_{t-1}) \tag{2.6}$$

In every time step t, the equations for computing cell state $m(t)$ and hidden layer $h(t)$ are presented as follows:

$$m_t = \alpha_t(\times)m_{t-1} + \beta_t(\times)tanh(W_{xm} \cdot x_t + W_{hm} \cdot h_{t-1})) \tag{2.7}$$

$$h_t = \gamma_t(\times)tanh(m_t) \tag{2.8}$$

2.3 Semantic Role Labeling

Semantic role labeling (SRL) is a task in Natural Language Processing to assign semantic roles for each argument for each predicate in given input sentence. In this section, the definition of semantic roles and the most commonly used annotation corpus for SRL are explained. In the end, the details on the semantic role labeling task are described.

2.3.1 Semantic Roles

Semantic roles are the representations that express the abstract role of the arguments of a predicate can take in the event (Jurafsky and James, 2016). When it comes to understanding natural language, one would want to understand the events and their participants of a given input sentence. In this case, the events refer to the predicate and the participants refer to the argument. Table 2.2 illustrates the connection between a predicate and its arguments.

Table 2.2: An example predicate and its arguments

Andy	eats	fried chicken
ARGUMENT	PREDICATE	ARGUMENT

In this example, "eat" is the predicate with "Andy" and "fried chicken" as its argument. With this point of view, the predicate can be seen as the center of the sentence, followed by the arguments that depend on it.

Knowing the predicate and its arguments is not enough to understand the sentence since the roles of the arguments towards the predicate are unknown. In the previous example, it will be more meaningful to differentiate that "Andy" is the EATER and fried chicken is the EATENTHING. EATER and EATENTHING are the examples of semantic roles for the predicate "eat". These semantic roles could be used to identify the roles of the arguments regardless its position in the sentence. The previous example could be represented in two ways, as presented in Table 2.3

Table 2.3: Examples of a predicate and its deep roles

Andy	eats	fried chicken
EATER	PREDICATE	EATENTHING
The fried chicken	is eaten	by Andy
EATENTHING	PREDICATE	EATER

Both sentences represent the role of "Andy" and "fried chicken" as EATER and EATENTHING respectively, regardless of their position in the sentence as a subject or object.

There are many ways to define such semantic roles. From the examples above, the semantic roles are very specific for its predicate, known as deep roles (Jurafsky and James, 2016). EATER and EATENTHING are semantic roles for the predicate "eat", KICKER and KICKEDTHING are semantic roles for the predicate "kick", and so on. In order to further knowing more about the semantics of these arguments, the semantic roles could be generalized into more abstract roles. EATER and KICKER have something in common: they are volitional actors having direct causal responsibility for the predicate. For this reason, thematic roles are introduced as a set of semantic roles designed to capture semantic commonality between EATER and KICKER (Jurafsky and James, 2016). With this in mind, EATER and KICKER can be represented as AGENT, which represents the abstract concept that is a volitional causer of an event (or predicate). On the other hand, EATENTHING and KICKEDTHING both represent the direct objects that are affected by the event. The semantic role for EATENTHING and KICKEDTHING is THEME.

Table 2.4 describes the thematic roles often used across computational papers (Jurafsky and James, 2016)

Table 2.4: Examples of thematic roles (Jurafsky and James, 2016)

Thematic Role	Definition	Example
AGENT	The volitional causer of an event	The waiter spilled the soup
EXPERIENCER	The experiencer of an event	John has a headache
FORCE	The non-volitional causer of the event	The wind blows debris
THEME	The participant directly affected by an event	Benjamin Franklin broke the ice
RESULT	The end product of an event	The city built a requlation-size baseball diamond
CONTENT	The content of a propositional event	Mona asked "Did you met Mary Ann?"
INSTRUMENT	An instrument used in an event	He stunned catfish with a shocking device
BENEFICIARY	The beneficiary of an event	Ann Callahan makes hotel reservations for her boss
SOURCE	The origin of the object of a transfer event	I flew in from Boston
GOAL	The destination of an object of a transfer event	I drove to Portland

AGENT and EXPERIENCER represent the volitonal causer of an event and the experiencer of an event, respectively. While AGENT is volitional, the non-

volitional causer of an event is called FORCE. Furthermore, THEME and RESULT are the participant directly affected by an event and the end product of an event, respectively. BENEFICIARY represents the beneficiary of an event. CONTENT describes the content of a propositional event. An instrument used in an event is called INSTRUMENT. For location, there are SOURCE and GOAL which represent the origin and the destination of an object of a transfer event.

2.3.2 Annotation Corpus

There are available annotated corpus for SRL consists of sentences labeled with semantic roles. Researchers are using these annotated corpus for building supervised machine learning model for SRL. The two most commonly used annotation corpus for SRL are Proposition Bank and FrameNet.

2.3.2.1 Proposition Bank

Proposition Bank (Kingsbury and Palmer, 2002), shortened as PropBank, is a corpus in which sentences are annotated with semantic roles. PropBank corpus is available for many languages, such as English, Chinese, Hindi, Arabic, Finnish, and Portuguese. The main approach used for its semantic roles grouping is based on proto-roles and verb-specific semantic roles. Every verb sense has its set of semantic roles with argument numbers rather than names, for example: Arg0, Arg1. Arg2, etc. Generally, Arg0 represents PROTO-AGENT while Arg1 represents PROTO-PATIENT. Other argument number representations may vary based on each verb sense.

The PropBank entries are called frame files. One example of the frame files for one sense of verb eat is presented as follows.

Frame File:
Eat.01
Arg0: Eater
Arg1: Things Eaten
Arg2: Instrument used

Example:
Ex1: [Arg0 Andy] eats [Arg1 fried chicken] [Arg2 with spoon]
Ex2: [Arg1 That fried chicken] is eaten by [Arg0 Andy] [Arg2 with spoon]

For verb sense Eat.01, Arg0 acts as the Eater (PROTO-AGENT), and Arg1 represents the Things Eaten (PROTO-PATIENT). As we can see from the example

above, we can infer the commonality between examples Ex1 and Ex2 regardless its structure, be it in a passive or active voice. In both examples, Andy is the Eater and fried chicken is the Things Eaten. In this frame file, there is also another argument, Arg2, that represents the instrument used by the Eater. In example Ex1 and Ex2, the instrument is spoon.

Other non-numbered arguments are available in PropBank, the so-called ArgMs, representing modifiers that could be used across frame files. Some examples of ArgMS include:

TMP: When? LOC: Where? DIR: Where to/from?

The next annotation corpus is called FrameNet which has different approach on how to group the set of semantic roles. Instead of using verb-specific, it uses frame-specific grouping.

2.3.2.2 FrameNet

FrameNet (Baker et al., 1998) is an annotation corpus for semantic roles that are specific to a frame. In PropBank, the semantic roles are defined based on each sense of a verb. In contrast, a frame in FrameNet could include more than one predicate (verbs or nouns) that have the same background context. Each frame consists of two elements: 1.) A set of semantic roles related to this frame, and 2.) A set of predicates using the respective semantic roles.

One example is a frame called **change_position_on_a_scale** defined as:

`This frame consists of words that indicate the change of an Item's position on a scale (the Attribute) from a starting point (Initial value) to an end point (Final value).`

The set of semantic roles for a frame is divided into two roles: Core roles and Non-Core Roles. Core Roles are specific to a frame while Non-Core Roles are more general across frames (like ArgMs in PropBank). The set of semantic roles of the frame **change_position_on_a_scale** is explained as bellow:

Core Roles
ITEM: The entity that has a position on the scale.
ATTRIBUTE: The ATTRIBUTE is a scalar property that the ITEM possesses
DIFFERENCE: The distance by which an ITEM changes its position on the scale.
FINAL STATE: A description that presents the ITEM's state after the change in the ATTRIBUTE's value as an independent predication.
FINAL VALUE: The position on the scale where the ITEM ends up.
INITIAL STATE: A description that presents the ITEM's state before the change in the ATTRIBUTE's value as an independent predication.
INITIAL VALUE:The initial position on the scale from which the ITEM moves away.
VALUE RANGE: A portion of the scale, typically identified by its end points, along which the values of the ATTRIBUTE fluctuate.

Non-Core Roles
DURATION SPEED GROUP
The length of time over which the change takes place.
The rate of change of the VALUE.

For instance, the possible predicates of the frame change position on a scale are: *rose, increase, fell*.

The example of semantic roles of the frame change position on a scale can be seen as follows:

[ITEM Oil] rose [ATTRIBUTE in price] [DIFFERENCE by 2%].

[ITEM It] has increased [FINAL STATE to having them 1 day a month].

[ITEM Microsoft shares] fell [FINAL VALUE to 7 5/8].

[ITEM Colon cancer incidence] fell [DIFFERENCE by 50%] [GROUP among men]

a steady increase [INITIAL VALUE from 9.5] [FINAL VALUE to 14.3] [ITEM in dividends]

a [DIFFERENCE 5%] [ITEM dividend] increase...

As we can see from the examples above, *rose*, *fell*, and *increase* have the same set of semantic roles under the frame change position on a scale. Instead of defining the semantic roles for each verb sense one by one, FrameNet groups predicates (not limited to verbs) that have same semantic roles as one frame.

2.3.3 Common Features for SRL

The first set of features for SRL is proposed by Gildea and Jurafsky (2000). They are the first ones who used supervised machine learning approach to solve SRL. Over the years, many research proposed new set of features to improve the result, but they still used the basic features proposed by Gildea and Jurafsky (2000). The common features used for solving SRL task are:

1. The predicate.
   Usually in a form of verb.
2. The phrase type of the constituent.
   NP, PP, etc
3. The headword of the constituent.
   The black bird. Headword: bird.
4. The headword part of speech of the constituent.
   Example: NNP.
5. The path of the parse tree from constituent to predicate.
   This is to represent the grammatical relationships between the constituent and the predicate. Example: NP S VP VBD
6. The voice of the clause, active or passive.
   Example: I eat chicken rice (active), Chicken rice is eaten by me (passive).
7. The binary linear position of the constituent from the predicate.
   Could be before or after the predicate.
8. The subcategorization of the predicate
   Set of arguments that appear in the verb phrase VP. Example: NP and PP in 'VP -> VBD NP PP'
9. The named entity type of the constituent
   Example: Organization, Person, Location
10. The first and last words of the constituent.

There are also other additional features that could be used for SRL, such as sets of n-grams inside the constituent. Another variation is to use dependency parser instead of syntactic parser for extracting features.

2.3.4 Historical Perspectives

SRL can be seen as either a classification or sequence labeling problem. The earlier research on SRL was conducted with the classification approach, meaning that each argument is being predicted independently from the others. Those research focused on how to extract meaningful features out of syntactic parsers (Gildea and Jurafsky, 2002; Gildea and Palmer, 2002; Pradhan et al., 2005), such as the path to predicate and constituent type. This syntactic information plays a pivotal role in solving SRL problem (Punyakanok et al., 2008) as it addresses SLR's long distance dependency (Zhou and Xu, 2015). Thus, traditional SRL system heavily depends on the quality of the parsers. The analysis done by Pradhan et al. shows that most errors of the SRL system were caused by the parser's error (Pradhan et al., 2005). In addition, those parsers are costly to build, since it needs linguistic experts to annotate the data. If we want to create an SRL system on another language, one should build a new parser all over again for it (Zhou and Xu, 2015).

In order to minimize the number of hand-crafted features, Collobert et al. utilized deep learning for solving NLP tasks including Part-of-Speech Tagging (POS), Chunking (CHUNK), Named Entity Recognition (NER), and Semantic Role Labeling (SRL) with classification approach (Collobert et al., 2011). The research aims to prevent using any task-specific feature in order to achieve state-of-the-art performance. The word embedding is used as the main feature across tasks, combined with Convolutional Neural Networks (CNN) architecture to train the model. They achieve promising results for the POS Tagging and Chunking, while for SRL features from the parsers are still needed to achieve competitive results.

Different from the previous works, Zhou et al. view SRL as a sequence labeling problem in which the arguments are labeled sequentially instead of independently (Zhou and Xu, 2015). They proposed an end-to-end learning of SRL using Deep Bi-Directional Long Short-Term Memories (DB-LSTM), with word embedding as the main feature. Their analysis suggests that the DB-LSTM model implicitly extracts the syntactic information over the sentences and thus, syntactic parser is not needed. The research result outperforms the previous state-of-the-art traditional SLR systems as it achieves F1 score of 81,07%. The research also shows that the performance of the sequence labeling approach using DB-LSTM is better than the classification approach using CNN, since the DB-LSTM can extract syntactic information implicitly.

CHAPTER 3
METHODOLOGY

In this chapter, we describe the methodology used in this research. It consists of data gathering, data pre-processing, data annotation, experiment, and evaluation. Before we describe each part in details, we explain the big picture of the methodology in a form of pipeline.

3.1 Pipeline

The goal of this research is to build a model that predicts the semantic roles of each Indonesian sentence. In other languages such as English, there are already semantic role corpus from which the SRL system is built. Unfortunately, that is not the case for Indonesian, since there is no annotated corpus available yet. We therefore create our own with the annotation guideline crafted for Indonesian conversational language. In this research, we focus on building SRL system for the conversational Indonesian language.

We view SRL as a sequence labeling problem. Suppose that we have an input of n words $w = (w_1, w_2, ..., w_n)$, the goal is to find the best label sequence $y = (y_1, y_2, ..., y_n)$, with y_i representing the semantic roles. The probabilities of the label in each time step i is described as follows.

$$P(y_i | w_{i-l}, ..., w_{i+l}, y_{i-l}, ..., y_{i+l}) \tag{3.1}$$

whereby l is a small number.

In this section, we explain our research methodology including the data annotation, features used, and the proposed model architecture. Figure 3.1 shows the pipeline of this research.

This research uses the data from one of Kata.ai's chat bots. Firstly, the data is pre-processed before going into the next steps. After that, the data is then annotated with semantic roles based on the annotation guideline proposed by us for Indonesian conversational language.

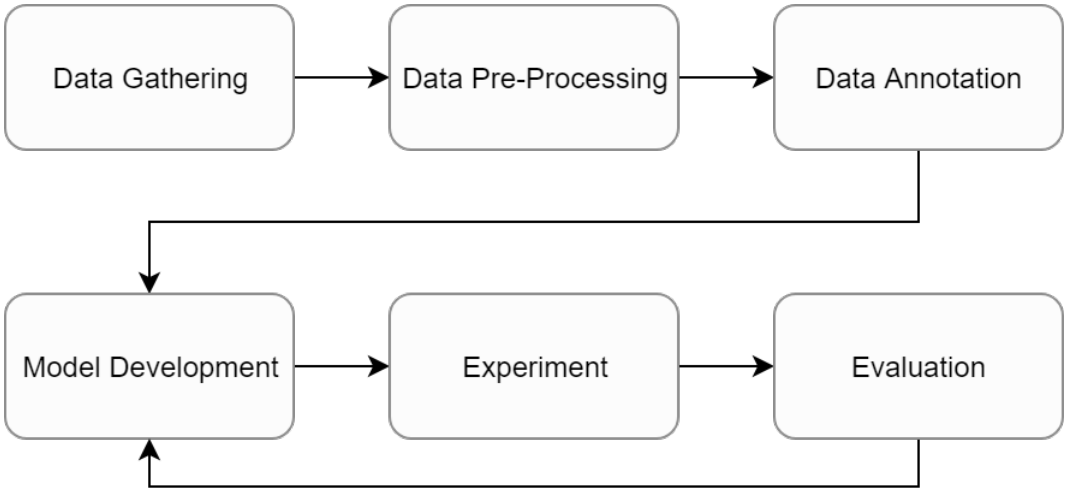

Figure 3.1: Methodology Pipeline

The model development step consists of feature extraction and model architecture. The features are Word Embeddings, POS Tag, and Neighboring Word Embeddings. The model architecture consists of main and additional layer. The main layer options include vanilla Long Short-Term Memories (LSTM), Bi-Directional LSTM (BLSTM), Deep BLSTM (DBLSTM), DBLSTM-Zhou, and DBLSTM Highway.

There are 2 main scenarios for the experiment. The first scenario aims for finding the best set of features that outputs the best performance. The goal of the second scenario is to find which deep learning model architecture has the best result.

We use 5-fold cross validation for every experiment. Each experiment is evaluated based on precision, recall, and F1 of each semantic roles with partial match approach (Seki and Mostafa, 2003). The performance of a model is retrieved by averaging precision, recall, and F1 of every semantic role. We then analyze and explain the results of each experiment scenario.

3.2 Data Gathering

In this research we use real-world data from one of Kata.ai chat bots. We firstly retrieved data consisting of 40.000 instances of text chats. We then manually deleted junk chats which contain, for example, only laugh or greeting. After that, we run a script to delete duplicate chats. The deletion process outputs a clean data with 30.000 instances in total. Finally, we randomly selected 9.000 out of 30.000 instances as the data to be annotated. This data set will be the one which is trained and tested to build the SRL system.

It is worth to note that conversational language has unique characteristics. First,

they use slangs and abbreviations. For example, one might use *"u"* instead of *"you"* in *"I brought u a present"*. The grammars are often unstructured and thus, one cannot rely on syntactic parsers to build SRL system for conversational language. The sentences are also filled with interjections such as *"haha"* and *"lol"*. Lastly, since conversational sentences are really short, averaging around 5-7 words per sentence, it sometimes contains incomplete information. These are the interesting challenges the SRL system should learn and tackle.

3.3 Data Pre-Processing

After the data has been gathered, the next step is to pre-process the data so that it could be fed into the machine learning model later. In this step, each sentence is going through a process called tokenization. Tokenization splits sentence into its individual tokens. Traditionally, one can split sentence by *space*, however, conversational language often contains a concatenated word with symbol such as "makan yuk!!!". It thus needs further tokenization technique, that is, splitting the alphabet with symbol tokens. This way, "makan yuk!!!" will be tokenized as "makan", "yuk", and "!!!". In addition to tokenization by *space*, the rules are listed as follows:

1. <alphabet><symbol> to be <alphabet><space><symbol>
2. <symbol><alphabet> to be <symbol><space> <alphabet>

3.4 Data Annotation

In this work, we create a new set of semantic roles mainly crafted for Indonesian conversational language. The summary of semantic roles proposed with its examples are presented in Table 3.1.

Table 3.1: Set of Semantic Roles for Conversational Language

Semantic Roles	*Example*
AGENT	Aku beliin kamu kado
PATIENT	Aku beliin kamu kado
BENEFICIARY	Aku beliin kamu kado
GREET	Hai Budi! Aku beliin kamu kado
MODAL	Aku bisa makan di rumah besok
LOCATION	Aku bisa makan di rumah besok
TIME	Aku bisa makan di rumah besok

These semantic roles are mainly inspired by the work of Saeed (1997), except that AGENT and PATIENT are adapted from PROTO-AGENT and PROTO-PATIENT as explained by Dowty (1991). The main difference of this set of semantic roles compared to the previous ones is GREET.

The center of all semantic roles is the PREDICATE. As in English, PREDICATE in Indonesian is usually in a form of *verb*, as illustrated by the examples below:

- "Kemarin aku makan di rumah"
- "Aku ada ujian nih hari Senin"

In Indonesian, however, predicate can also be an adjective. Some examples are presented as follows:

- "Kamu cantik deh"
- "Aku lagi sedih nih"

In this section, we briefly explain each semantic roles with their respective examples.

1. AGENT
   AGENTis described as initiator of action or capable of volition. It can also be the one which perceives action but not in control. Usually, an AGENTis the subject of a verb in active voice. The examples of AGENT in a sentence are given as follows:
   - "Aku makan ayam dulu ya"
   - "Kamu gak tidur?"
   - "Kamu mau beliin aku pulsa?"
2. PATIENT
   PATIENT is described as an entity affected by action or undergoes change of state. It can also be an entity being located. PATIENT is usually the direct object of a verb in active voice. If the predicate is in a form of adjective, the subject of it is labeled as PATIENT. The examples of PATIENT in a sentence are given as follows:
   - "Aku makan ayam dulu ya"
   - "Kamu mau beliin aku pulsa?"

- "Aku lagi sedih nih.."

3. BENEFICIARY
   BENEFICIARY is an entity which gets benefit of the predicate. It is usually in a form of indirect object of a predicate (can be a ditransitive verb or an adjective). The examples of BENEFICIARY in a sentence are given as follows:

   - "Kamu mau beliin aku pulsa?"
   - "Aku pengen ngobrol sama kamu"

4. GREET
   GREET is the main difference of this set of semantic roles for conversational language. GREET refers to an animate object, usually a person, which is being greeted in a chat. In conversational language, one often calls the name of person it is talking to. This information is useful, for instance, we can derive that *"you"* refers to *"Budi"* in *"Halo Budi! Aku beliin kamu kado loh"*. The examples of GREET in a sentence are given as follows:

   - "Hai rizky! kamu udah makan belum?"
   - "aku ga bisa tidur nih Val"

5. MODAL
   MODAL refers to *modal verb* of a predicate. The MODAL examples are *"boleh, harus, pernah, sudah, udah, mesti, perlu, akan, lagi, bisa, mau, ingin, pengen, pingin"*. The examples of MODAL in a sentence are given as follows:

   - "Aku mau makan dulu ya!"
   - "Kamu udah tidur belum?"

6. LOCATION
   LOCATION refers to the location of a predicate. The examples of LOCATION in a sentence are given as follows:

   - "Aku mau makan di rumah ya!"
   - "Kamu gak pergi ke sekolah?"

7. TIME
   TIME refers to the time of a predicate. The examples of TIME in a sentence are given as follows:

   - "Kemarin aku makan di rumah"

- "Aku ada ujian nih hari Senin"

Following Collobert et al., all the labels are tagged using BIO (Begin Inside Outside) tagging Collobert et al. (2011). Suppose that a label PATIENTconsists of more than one word, such as *"ayam goreng"* in "Aku makan *ayam goreng*", "ayam" and "goreng" are tagged as "B-Patient" and "I-Patient", respectively. If the label has only one word, than it is tagged as "B-Patient". Word that does not have any label is thus tagged as "O" which means "Others".

After the data has been labeled, the labels need to be encoded in a way the deep learning model understands. To do so, the labels are then transformed into *one-hot-vector*. Each label is mapped into a unique one-hot-vector, hence the relation is 1-to-1.

Table 3.2: Set of Semantic Roles for Conversational Language

Sentence	Aku	pengen	makan	ayam
BIO-Label	B-AGEN	B-MD	B-PRED	B-PATIENT
One-Hot-Vector	[1, 0, 0, .., 0]	[0, 1, 0, .., 0]	[0, 0, 1, .., 0]	[0, .., 1, 0, 0]

Table 3.2 shows an example of how sentence is labeled with the one-hot vectors representations of BIO format.

3.5 Feature Extraction

In this step, we extract features from the data that has been annotated with semantic roles. We propose three features which will be combined later to find the best feature selection that outputs the best result. Those features are word embedding, POS-Tag, and neighboring word embeddings.

3.5.1 Word Embedding

Word embedding represents word as a vector. Word embedding has proved to be one of the most contributing features by a lot of deep learning research, such as for SRL system proposed by Zhou and Xu (2015) and Collobert et al. (2011). The interesting characteristic of word embedding is that similar words have proved to have similar vectors. This is very important when dealing with conversational language which has a lot of slang words. For instance, pronoun "Aku" will have similar vector with its slang form, "Gue". We believe that this feature will contribute greatly to the model performance. Therefore, we utilized our embedding as one of our feature candidates.

In order to utilize word embedding as our features, we conduct three steps which consist of: 1.) data gathering for building word embedding model, 2.) training the word embedding model, and 3.) converting words into vectors with the trained word embedding model.

1. Data gathering for building word embedding model
   We first gather an unlabeled dataset in order to build the word embedding model. The dataset is retrieved from 1.300.000 sentences of chats from Kata.ai.

2. Training the word embedding model
   The word embedding model is trained using the afore-mentioned unlabeled data set. This model is used to transform words into their respective vector representations.

3. Converting words into vectors with the trained word embedding model
   The trained word embedding model is then used to convert words into vectors. The example of this conversion can be seen in Table 3.3

Table 3.3: An example of word embedding vector representation with dimension of 3

Sentence	Aku	pengen	makan	ayam
Word Vector	[0.2, -0.4, 0.9]	[0.7, 0.1, 0.2]	[0.1, 0.6, -0.5]	[0.9, 0.1, 0.8]

In Table 3.3 provides an example assuming that the vector dimension is 3. This means that every word is mapped into a vector with a length of 3.

3.5.2 Part-of-Speech Tag (POS Tag)

POS Tag is a feature to represent the class of each word. In this research, the POS tags used are: Verb (V), Noun (NN), Adjective (ADJ), Adverb (ADV), Coordinative Conjunction (CC), Subordinative Conjunction (SC), Interjection (INTJ), Question (WH), Preposition (PREP), and Negation (NEG). Before feeding the feature to the deep learning model, we encode the POS tag as a one-hot-vector. Each one-hot-vector represents each POS tag uniquely.

The example of POS Tag features is presented in Table 3.4

Table 3.4: An example of POS Tag feature and its respective one-hot-vector

Sentence	Aku	pengen	makan	ayam
POS Tag	NN	ADV	V	NN
One-Hot-Vector	[1, 0, 0, .., 0]	[0, 1, 0, .., 0]	[0, 0, 1, .., 0]	[0, .., 1, 0, 0]

While most of the deep learning research aims for not using such feature, we argue that POS tag is still important for building a robust model in our case. This is because of the fact that size of our corpus is relatively small compared to the huge English-based SRL corpus such as ConLL 2015 by Carreras and Màrquez (2005). We argue that having Verb as one of our POS Tag will help to determine which one is the predicate, since predicate is usually in a form of verb, though some can also be adjectives. As the arguments having semantic role are mostly in a form of Noun, POS Tag Noun will be a helpful information as well. Other POS tags will help to determine which word that obviously does not have any semantic role.

In this work, we use gold-standard POS tag on our data as the features to prevent propagation errors from the POS tag model. This way, we can focus on analyzing errors resulting from the SRL model later in chapter 5.

3.5.3 Neighboring Word Embeddings

Neighboring word embeddings are the vector representations of words located before and after the word being processed. We use specifically one word before and after the word being processed. Suppose that we are processing the word w_t at time step t, the neighboring words embeddings are the vector representations of the word w_{t-1} and w_{t+1}. We argue that this feature can be useful for capturing the context of the word by looking at the surrounding words. Suppose that the machine is processing the word *"rumah"* in *"aku tidur di rumah"*. By looking at the previous word, which is the preposition *"di"*, it gives a hint that *"rumah"* might have the semantic role LOCATION.

Table 3.5: An example of neighboring word embedding vectors of every time step

Sentence	Aku	pengen	makan	ayam
Word Vector	[0.2, -0.4, 0.9]	[0.7, 0.1, 0.2]	[0.1, 0.6, -0.5]	[0.9, 0.1, 0.8]
Neighbor Vector	[0.0, 0.0, 0.0]	[0.2, -0.4, 0.9]	[0.7, 0.1, 0.2]	[0.1, 0.6, -0.5]
	[0.7, 0.1, 0.2]	[0.1, 0.6, -0.5]	[0.9, 0.1, 0.8]	[0.0, 0.0, 0.0]

Table 3.5 illustrates the neighboring word embeddings of every time step. The table shows that in every time step, it adds the word vector information from the left and right. Since the first word does not have any previous word, its left neighbor is a vector $\vec{0}$. This is also the case for the last word which does not have any subsequent word. Hence, the right neighbor of the last word is also a vector $\vec{0}$.

3.6 Model Architecture

Recurrent Neural Networks (RNN) has a nature advantage for solving sequence labeling problem Zhou and Xu (2015). Hochreiter and Schmidhuber (1997) proposed Long Short-Term Memories (LSTM) as the specific version of RNN designed to overcome vanishing and exploding gradient problem. In this research, we experiment on various LSTM architectures, namely vanilla LSTM, Bi-Directional LSTM (BLSTM), Deep BLSTM (DBLSTM), DBLSTM-Zhou, and DBLSTM-Highway. All these LSTM architectures are considered as the main layers. Moreover, there are two additional layers: Convolutional Neural Networks (CNN) and attention. CNN layer is located underneath the LSTM layer while the attention is placed on top of the LSTM. The attention layer is the one we proposed in this research. These additional layers can be used for any afore-mentioned LSTM architectures.

3.6.1 Main Layers

The main layers consist of Vanilla LSTM (LSTM), Bi-Directional LSTM (BLSTM), Deep BLSTM (DBLSTM), DBLSTM-Zhou, and DBLSTM-Highway.

3.6.1.1 Vanilla LSTM (LSTM)

Vanilla LSTM is the basic, one-directional LSTM networks designed to overcome vanishing and exploding gradient problem found in RNN. To do so, it has forget gates, a gate to open or close incoming information from the previous time steps. We firstly explain the LSTM networks as the big picture for solving sequence labeling problem, followed by the details of every LSTM unit.

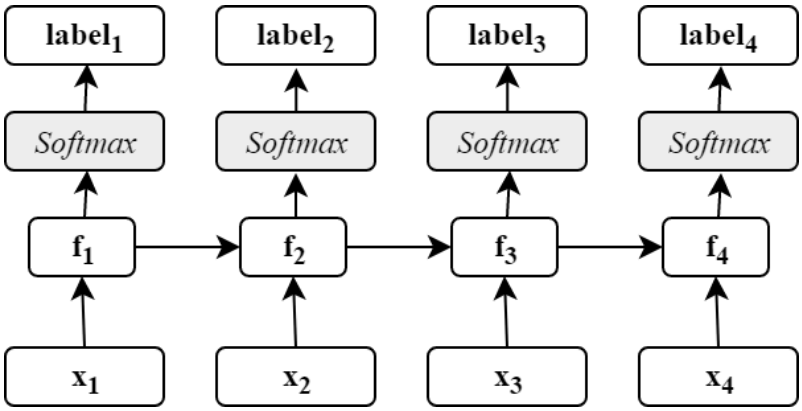

Figure 3.2: An architecture of vanilla LSTM with total time step of 4

Figure 3.2 illustrates the big picture of the vanilla LSTM networks used in this research. Suppose that we have sequence of input tensors $[\mathbf{x_1}; \mathbf{x_2}; ...; \mathbf{x_n}]$, with n denotes the number of time steps. Each input tensor $\mathbf{x_t}$ represents features of the word in time step t. Suppose that we have more than one feature (for example: Word Embedding and POS Tag), the vectors of these features are concatenated. For every time step t, each input tensor $\mathbf{x_t}$ is fed into the LSTM layer, resulting tensor $\mathbf{f_t}$, as shown in Equation 3.2.

$$\mathbf{f_t} = LSTM(\mathbf{x_t}, \mathbf{f_{t-1}}) \tag{3.2}$$

Equation 3.3 shows that each LSTM output from every time step is then processed by the time-distributed softmax layer to produce the probabilities of every possible label.

$$label_t = Softmax(\mathbf{f_t}) \tag{3.3}$$

In each time step, label with highest probability among others will be the final output. This way, we have all the predicted labels for every time step.

After explaining the big picture of LSTM networks, we describe the details of every LSTM unit in time step t shown in Equation 3.2. That is, given an input tensor $\mathbf{x_t}$, each LSTM unit will output tensor $\mathbf{f_t}$. Figure 3.3 illustrates the architecture of one LSTM unit in time step t.

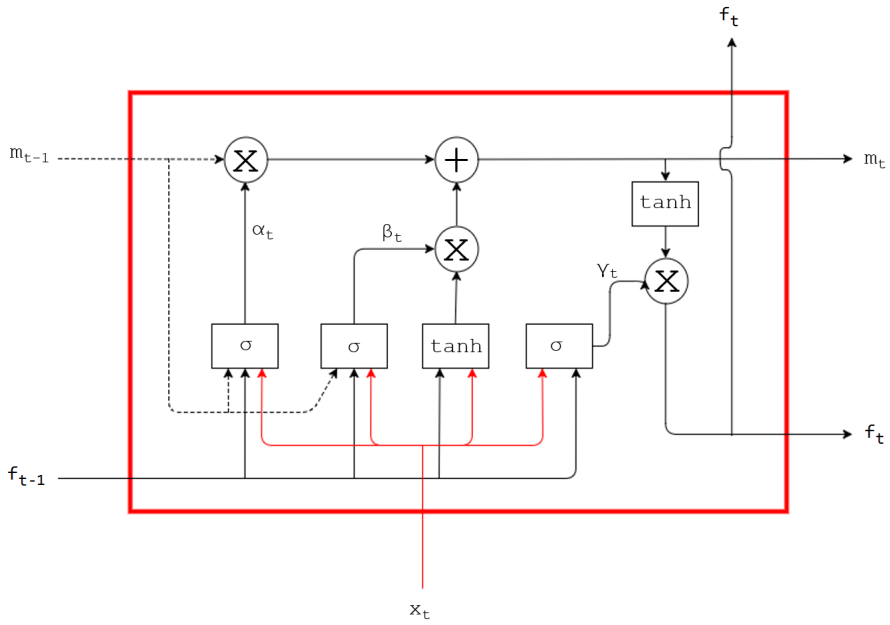

Figure 3.3: An LSTM unit in time step t. Adapted from (Rohman, 2017).

The LSTM unit in time step t requires an input tensor a_t. As explained in subchapter X, the equations to produce output tensor $\mathbf{f_t}$ are presented as follows:

$$\mathbf{m_t} = \alpha_t \cdot \mathbf{m_{t-1}} + \beta_t \cdot tanh(W_{xm} \cdot \mathbf{x_t} + W_{fm} \cdot \mathbf{f_{t-1}}) \tag{3.4}$$

$$\mathbf{f_t} = \gamma_t \cdot tanh(\mathbf{m_t}) \tag{3.5}$$

where α_t, β_t, and γ_t are the gates:

1. *Forget gates*: $\alpha_t = \sigma(W_{x\alpha} \cdot \mathbf{x_t} + W_{f\alpha} \cdot \mathbf{f_{t-1}} + W_{m\alpha} \cdot \mathbf{m_{t-1}})$
2. *Input gates*: $\beta_t = \sigma(W_{x\beta} \cdot \mathbf{x_t} + W_{f\beta} \cdot \mathbf{f_{t-1}} + W_{m\beta} \cdot \mathbf{m_{t-1}})$
3. *Output gates*: $\gamma_t = \sigma(W_{x\gamma} \cdot \mathbf{x_t} + W_{f\gamma} \cdot \mathbf{f_{t-1}} + W_{m\gamma} \cdot \mathbf{m_{t-1}})$

It is worth noting that the LSTM layer is recursive, meaning that one of the inputs comes from the output of previous time step. This way, the result of each time step also depends on the previous ones.

3.6.1.2 Bi-Directional LSTM (BLSTM)

Bi-Directional LSTM (BLSTM) is a modification of LSTM networks. It was firstly introduced by Schuster and Paliwal (1997) for the original recurrent neural networks. While vanilla LSTM only goes on one direction, BLSTM goes both ways in order to capture context information from the past and future, as explained by Zhou and Xu (2015). The idea is to have two LSTM layers, one for going forward and another for going backward, as illustrated in Figure 3.4.

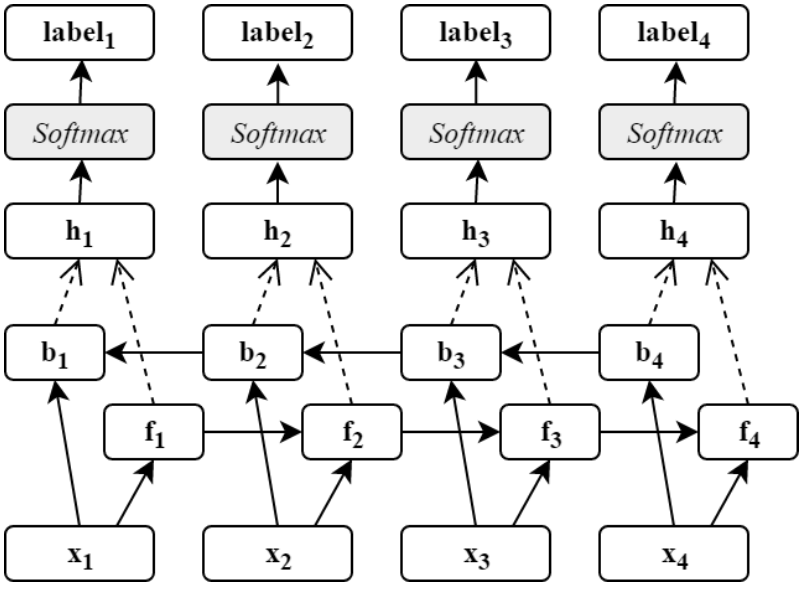

Figure 3.4: An architecture of Bi-Directional LSTM (BLSTM) with total time step of 4

Figure 3.4 shows that the input $\mathbf{x_t}$ in every time step t is fed into two LSTM layers, the first one for going forward and the second one for going backward. These are illustrated in Equation 3.6 and Equation 3.7

$$\mathbf{f_t} = ForwardLSTM(\mathbf{x_t}, \mathbf{f_{t-1}}) \tag{3.6}$$

$$\mathbf{b_t} = BackwardLSTM(\mathbf{x_t}, \mathbf{b_{t+1}}) \tag{3.7}$$

In each time step, the result tensors $\mathbf{f_t}$ and $\mathbf{b_t}$ are then concatenated to be one tensor output $\mathbf{h_t}$, as shown in Equation 3.8

$$\mathbf{h_t} = Concatenate(\mathbf{f_t}, \mathbf{b_t}) \tag{3.8}$$

Likewise in vanilla LSTM architecture, the output tensor $\mathbf{h_t}$ is then fed into softmax layer, as shown in Equation 3.9

$$label_t = Softmax(\mathbf{h_t}) \tag{3.9}$$

After which, the output of the softmax layer will determine the final label for each time step.

3.6.1.3 Deep BLSTM (DBLSTM)

Deep BLSTM (DBLSTM) is basically formed by stacking BLSTM layers. This concept was introduced by Zhou and Xu (2015). However, they proposed a different way of building the BLSTM layer, which will be explained in the next section. This section explains the stacked version of the original BLSTM layers as explained in the previous section. In this work, we stack two BLSTM layers. Figure 3.5 illustrates the architecture of DBLSTM.

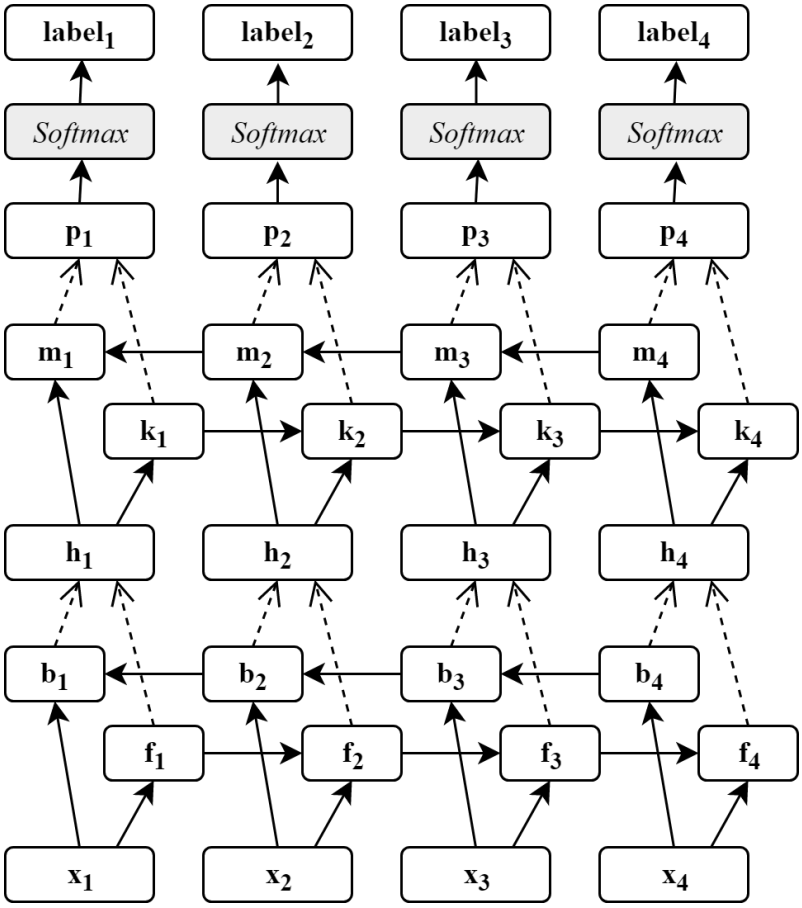

Figure 3.5: An architecture of Deep BLSTM (DBLSTM) with total time step of 4

Initially, the input tensors $\mathbf{x_t}$ are processed by the first BLSTM layer, resulting the output tensors $\mathbf{h_t}$ as explained in Equation 3.6, 3.7, and 3.8. The output tensors $\mathbf{h_t}$ are then processed by the second BLSTM layer. These are illustrated in Equation 3.10 and Equation 3.11.

$$\mathbf{k_t} = ForwardLSTM2(\mathbf{h_t}, \mathbf{k_{t-1}}) \tag{3.10}$$

$$\mathbf{m_t} = BackwardLSTM2(\mathbf{h_t}, \mathbf{m_{t+1}}) \tag{3.11}$$

In each time step, the result tensors $\mathbf{k_t}$ and $\mathbf{m_t}$ are then concatenated to be one tensor output $\mathbf{p_t}$, as shown in Equation 3.12

$$\mathbf{p_t} = Concatenate(\mathbf{k_t}, \mathbf{m_t}) \tag{3.12}$$

The output tensor $\mathbf{h_t}$ is finally fed into the last softmax layer to output the final label of each time step t.

3.6.1.4 Deep BLSTM-Zhou (DBLSTM-Zhou)

Zhou and Xu (2015) proposed another way of constructing the BLSTM layer. First, an LSTM layer processes the input sequence in forward direction. The output of this layer is then fed into the next LSTM layer as input, processed in backward direction. At this point, one BLSTM-Zhou layer is built. Zhou and Xu (2015) then stacked the BLSTM layers and named it as the deep bi-directional LSTM. In this work, we stack two BLSTM-Zhou layers for constructing the DBLSTM-Zhou as presented in Figure 3.6.

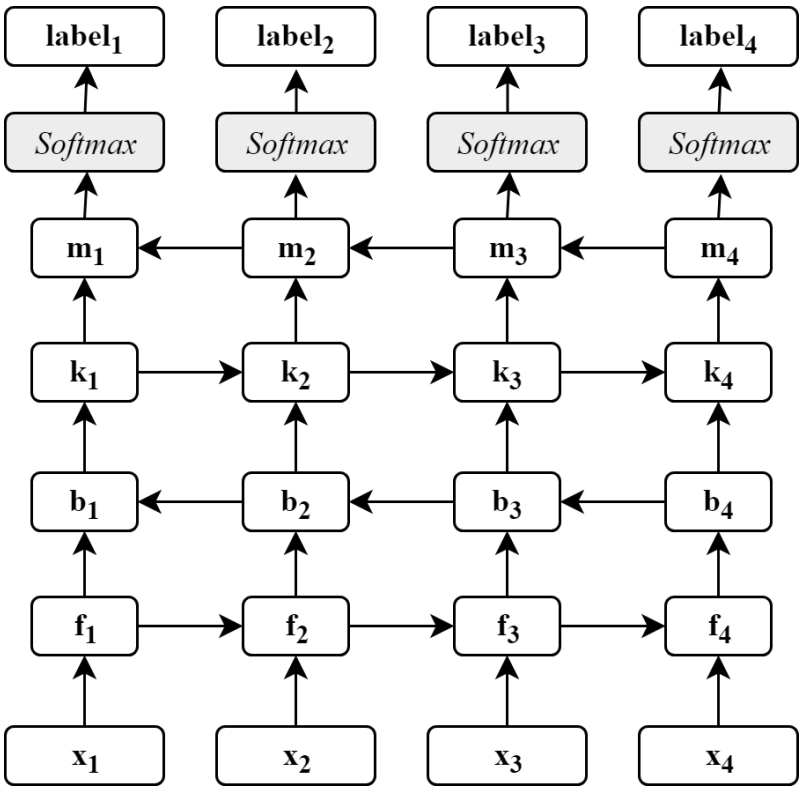

Figure 3.6: An architecture of DBLSTM-Zhou with total time step of 4

First, the input tensor $\mathbf{x_t}$ is processed by the first forward LSTM layer, resulting output tensor $\mathbf{f_t}$. This output is then fed into the backward LSTM layer to produce the output tensor $\mathbf{b_t}$. These processes are presented in Equation 3.13 and 3.14.

$$\mathbf{f_t} = ForwardLSTM1(\mathbf{x_t}, \mathbf{f_{t-1}}) \tag{3.13}$$

$$\mathbf{b_t} = BackwardLSTM1(\mathbf{f_t}, \mathbf{b_{t+1}}) \tag{3.14}$$

The output tensor $\mathbf{b_t}$ is then fed into the next forward LSTM layer to produce the next output tensor $\mathbf{k_t}$ (Equation 3.15). Finally, $\mathbf{k_t}$ is processed by the second

backward LSTM layer to produce the output tensor $\mathbf{m_t}$ (Equation 3.16).

$$\mathbf{k_t} = ForwardLSTM2(\mathbf{b_t}, \mathbf{k_{t-1}}) \tag{3.15}$$

$$\mathbf{m_t} = BackwardLSTM2(\mathbf{k_t}, \mathbf{m_{t+1}}) \tag{3.16}$$

The output tensor $\mathbf{m_t}$ is then processed by the softmax layer to produce the final label of every time step t.

3.6.1.5 Deep BLSTM-Highway (DBLSTM-Highway)

The DBLSTM-Highway architecture is adapted from the work of He et al. (2017). They used the same BLSTM architecture as Zhou and Xu (2015) but added the so-called highway connections (Srivastava et al., 2015) for their DBLSTM architecture. Figure 3.7 illustrates the DBLSTM-Highway architecture.

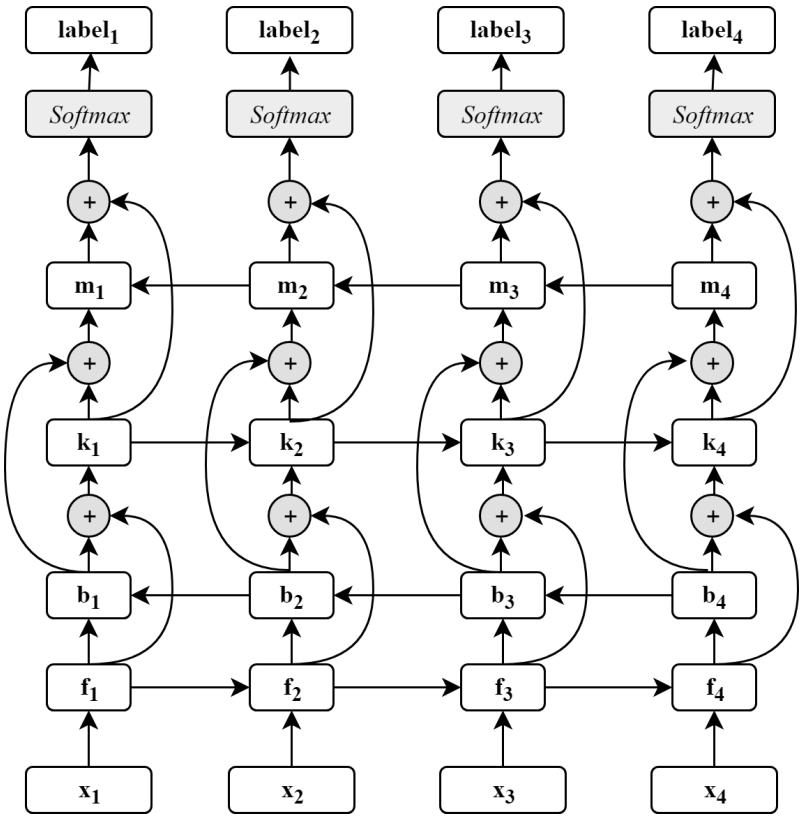

Figure 3.7: An architecture of DBLSTM-Highway with total time step of 4

First, the input tensor $\mathbf{x_t}$ is processed by the first forward LSTM layer which produces output tensor $\mathbf{f_t}$.

$$\mathbf{f_t} = ForwardLSTM1(\mathbf{x_t}, \mathbf{f_{t-1}}) \tag{3.17}$$

The tensor $\mathbf{f_t}$ is then fed into the first backward LSTM layer which outputs tensor $\mathbf{b_t}$. The tensors $\mathbf{b_t}$ and $\mathbf{f_t}$ are then concatenated before being processed by the sigmoid function which outputs the weight $s_{1,t}$. After that, $\mathbf{b_t}$ is retrieved by summing the multiplication results of $s_{1,t}$ with $\mathbf{b_t}$ and $(1-s_{1,t})$ with $\mathbf{f_t}$

$$\mathbf{b_t} = BackwardLSTM1(\mathbf{f_t}, \mathbf{b_{t+1}}) \tag{3.18}$$

$$s_{1,t} = \sigma(W_1 \cdot [\mathbf{b_t}; \mathbf{f_t}] + d_1) \tag{3.19}$$

$$\mathbf{b'_t} = s_{1,t} \cdot \mathbf{b_t} + (1 - s_{1,t}) \cdot \mathbf{f_t} \tag{3.20}$$

$\mathbf{k'_t}$ is retrieved by the same way as $\mathbf{b'_t}$, which we define as follows:

$$\mathbf{k_t} = ForwardLSTM2(\mathbf{b'_t}, \mathbf{k_{t-1}}) \tag{3.21}$$

$$s_{2,t} = \sigma(W_2 \cdot [\mathbf{k_t}; \mathbf{b_t}] + d_2) \tag{3.22}$$

$$\mathbf{k'_t} = s_{2,t} \cdot \mathbf{k_t} + (1 - s_{2,t}) \cdot \mathbf{b_t} \tag{3.23}$$

Lastly, $\mathbf{m'_t}$ is retrieved by the same way as $\mathbf{k'_t}$, which we define as follows:

$$\mathbf{m_t} = BackwardLSTM2(\mathbf{k'_t}, \mathbf{m_{t+1}}) \tag{3.24}$$

$$s_{3,t} = \sigma(W_3 \cdot [\mathbf{m_t}; \mathbf{k_t}] + d_3) \tag{3.25}$$

$$\mathbf{m'_t} = s_{3,t} \cdot \mathbf{m_t} + (1 - s_{3,t}) \cdot \mathbf{k_t} \tag{3.26}$$

The output tensor $\mathbf{m'_t}$ is then processed by the softmax layer to produce the final label of every time step t.

3.6.2 Additional Layers

The additional layers consist of Convolutional Neural Networks (CNN) and attention mechanism. These additional layers can be used in addition to any of the main layers.

3.6.2.1 Convolutional Neural Networks (CNN)

In addition to the main layer architecture, we also experiment on adding Convolutional Neural Networks (CNN) layer underneath the main layer. The rationale is to capture raw context from the neighboring input tensors. This way, CNN can implicitly extract meaningful context information. Figure 3.8 illustrates the big picture of adding CNN underneath the main layer. The main layer is illustrated by

the shaded architecture inside the rectangle. The main layer can be changed into any LSTM variants explained in the previous sections. In this illustration, we use DBLSTM-Highway as the main layer.

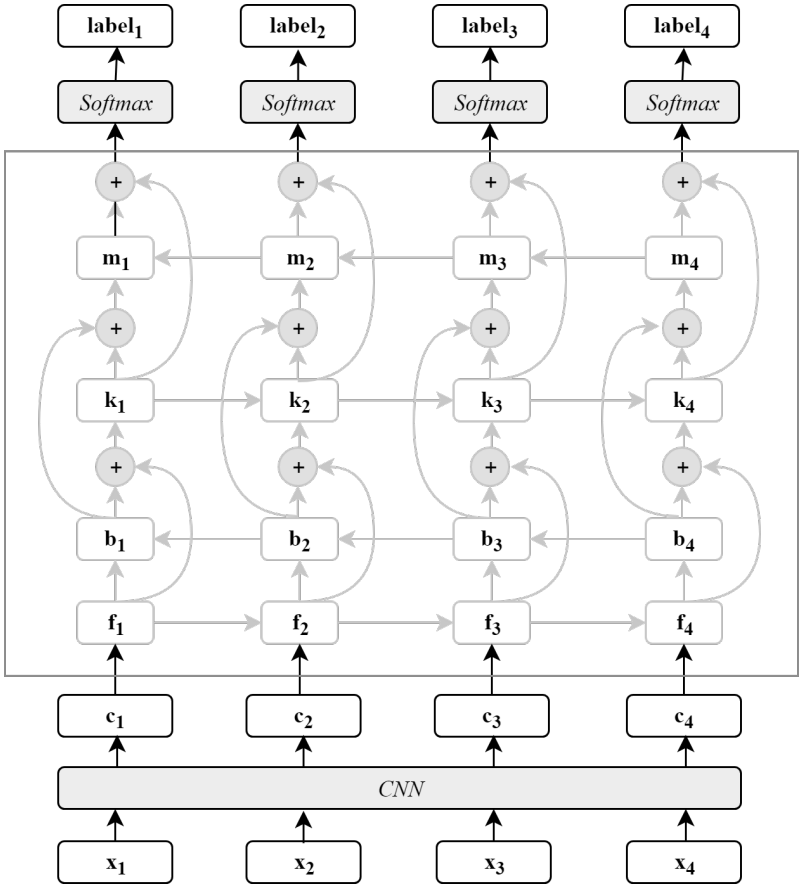

Figure 3.8: An architecture of adding CNN underneath the main layer with total time step of 4. The main layer is illustrated by the shaded architecture inside the rectangle. The main layer can be changed into any LSTM variants.

Instead of feeding the main layer with raw input tensor $\mathbf{x_t}$, this architecture firstly processes it through CNN layer, resulting output tensor $\mathbf{c_t}$, as shown in Equation 3.27.

$$[\mathbf{c_1};\mathbf{c_2};...;\mathbf{c_n}] = CNN([\mathbf{x_1};\mathbf{x_2};...;\mathbf{x_n}]) \tag{3.27}$$

The result tensor $\mathbf{c_t}$ is then fed into the main layer that can be any of LSTM variants. The output of the LSTM layer is then processed by the time-distributed softmax layer to determine the final output label.

3.6.2.2 Attention Mechanism

In this work, we propose an additional attention mechanism that can be used on top the main layer. The rationale is to add a dense yet useful high-level information

containing a sentence context to every time step in order to help the machine to decide semantic roles better. With this in mind, we design an attention mechanism on top of the main layer which can be any of the LSTM variant explained in the previous sections. Figure 3.9 illustrates the architecture in which attention mechanism is added on top of the main layer. For this illustration, the DBLSTM-Highway is used as the main layer.

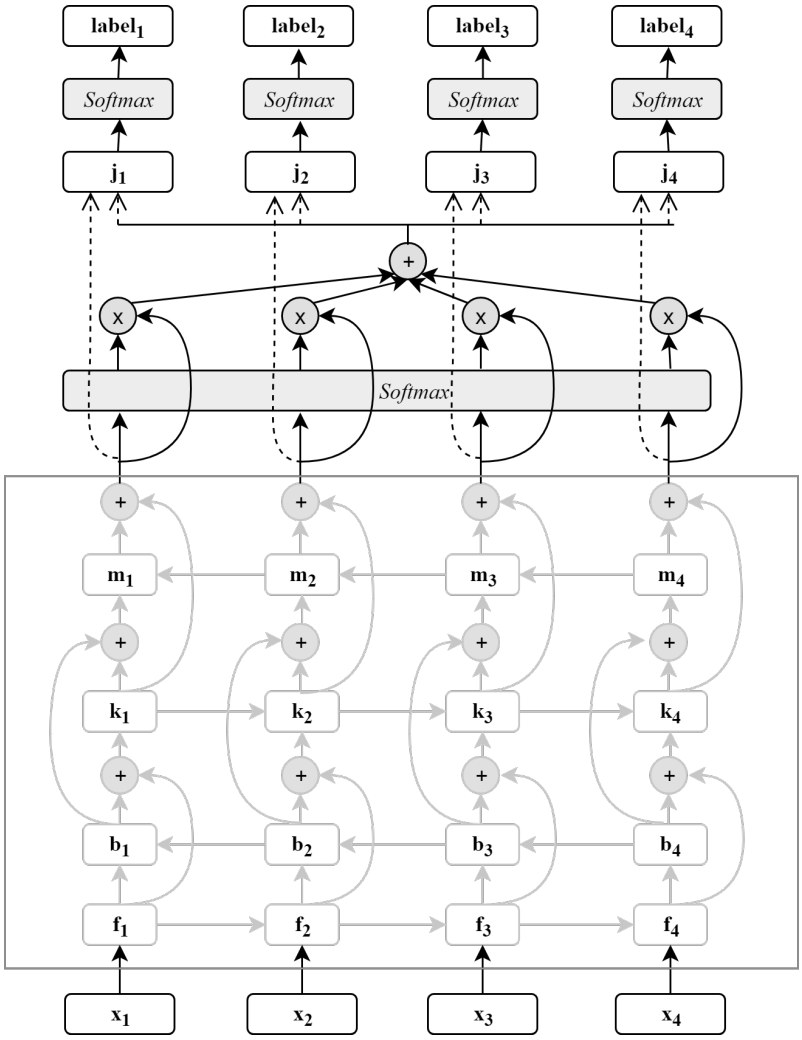

Figure 3.9: An architecture of adding attention mechanism on top of the main layer with total time step of 4. The main layer is illustrated by the shaded architecture inside the rectangle. The main layer can be changed into any LSTM variant.

The attention mechanism firstly collects the context information by multiplying trainable weights with all the vectors from every time step of the last LSTM output. We sum each element for each weighted vectors to reduce the dimension. The results are then fed into a hidden softmax layer which outputs weights with a total of 1. The original output vectors of the last LSTM output are multiplied by these distributed weights respectively. We then sum all the multiplication results as the

final context information. The original LSTM outputs are concatenated with this context information before going to the last softmax layer to predict the semantic roles.

Suppose that we have $\mathbf{m_i}$ as the last LSTM output, it is then fed into a differentiable neural networks function $g(\mathbf{m_i})$ in which it is multiplied by the time-distributed matrix $\mathbf{W} \in \mathbb{R}^{H \times K}$ and all the elements in it are summed. H is the vector dimension of $\mathbf{m_i}$, meanwhile K is the dimension size that we want as an output when we multiply W with $\mathbf{m_i}$.

$$g(\mathbf{m_i}) = Sum(\mathbf{W}.\mathbf{m_i}) \tag{3.28}$$

Once we have all the values of $[g(\mathbf{m_1}); g(\mathbf{m_2}); ...; g(\mathbf{m_n})]$, we make it as an input for the softmax layer, resulting weights $[\alpha_1; \alpha_2; ...; \alpha_n]$. All the original LSTM outputs $[\mathbf{m_1}; \mathbf{m_2}; ...; \mathbf{m_n}]$ are multiplied by these weights, with the results of $[\mathbf{r_1}; \mathbf{r_2}; ...; \mathbf{r_n}]$

$$[\alpha_1, \alpha_2, ..., \alpha_n] = Softmax([g(\mathbf{m_1}); g(\mathbf{m_2}); ..., g(\mathbf{m_n})]) \tag{3.29}$$

$$\mathbf{r_i} = \alpha_i.\mathbf{m_i} \tag{3.30}$$

We then sum all these vectors element-wise to have a context tensor $\mathbf{z}$. All the original LSTM outputs are thus concatenated with tensor $\mathbf{z}$ as the additional information to predict the semantic roles.

$$\mathbf{z} = \mathbf{r_1} + \mathbf{r_2} + ... + \mathbf{r_n} \tag{3.31}$$

$$\mathbf{j_i} = Concatenate(\mathbf{m\_i}, z) \tag{3.32}$$

Lastly, the time-distributed softmax layer produces the final semantic roles label.

3.7 Experiment

We use 5-fold cross validation for out experiments and thus, the data set is firstly split into 5 sets. These 5 sets are divided into training and testing sets with the ratio of 4:1. After that, we train the model using the training set and evaluate it using the testing set. This process is done 5 times until every set of data is tested.

3.8 Evaluation

In each scenario, we evaluate the trained model in order to see how good it predicts the semantic roles as expected. The metrics for our evaluation are precision, recall, and F1. These metrics is applied to all semantic role labels. We then average each metrics from all semantic role labels to get the average precision, recall, and F1 of a model. The evaluation approach used is partial match in which a set of predicted labels is counted right if there is an intersection with the gold-standard (Seki and Mostafa, 2003).

Suppose that we are evaluating the semantic role PATIENT. The rules for evaluation using partial match for semantic role PATIENTare explained as follows.

1. Counting *True Positive* (TP)
   For every gold standard label that has intersection with the predicted label, the value of True Positive (TP) is added by 1.

Gold-standard: Aku pengen makan <Patient>**ayam goreng**</Patient> deh
Predicted.1: Aku pengen makan <Patient>**ayam goreng**</Patient> deh
Predicted.2: Aku pengen makan <Patient>**ayam**</Patient> goreng deh
Predicted.3: Aku pengen makan <Patient>**ayam goreng deh** </Patient>
Predicted.4: Aku pengen makan ayam goreng deh

   The examples above illustrate four scenarios of possible predicted results, denoted as Predicted.1, Predicted.2, Predicted.3, and Predicted.4, given a gold-standard called Expected which has "ayam goreng" as the PATIENT.

   Predicted.1 predicts exactly the same as the Expected, hence the value of TP is added by 1. The result of Predicted.2 has an intersection with the gold-standard, which is "ayam", the value of TP is then added by 1 as well. Although Predicted.3 predicts too much as it includes "deh" as part of PATIENT, it also has an intersection with the gold-standard, which is "ayam goreng". The Predicted.3 therefore also adds the value of TP by 1. Meanwhile, Predicted.4 does not predict anything. In this case, the value of TP is not added at all.

2. Counting *False Positive* (FP)
   For every predicted label that should not be predicted according to gold-

standard, the value of False Positive (FP) is added by 1.

Gold-standard: Aku pengen makan <Patient>**ayam goreng**</Patient> deh Predicted.1: <Patient>Aku</Patient> pengen makan ayam goreng deh

From the example above, "Aku" is predicted as PATIENT, while it should not be predicted as PATIENTaccording to the gold-standard. This will add the value of FP by 1.

3. Counting *False Negative* (FN)
   For every gold-standard label that is either not predicted or predicted with wrong label, the value of False Negative (FN) is added by 1.

Gold-standard: Aku pengen makan <Patient>**ayam goreng**</Patient> deh Predicted.1: Aku pengen makan <Agent>**ayam goreng**</Agent> deh Predicted.2: Aku pengen makan ayam goreng deh

From the example above, Predicted.1 predicts "ayam goreng" as AGENT, while it should be predicted as PATIENT. In this case, the value of FP is added by 1. Predicted.2 illustrates an example which does not predict "ayam goreng" with any label, while it should be predicted as PATIENT. In this case, the value of FP is added by 1 as well.

After we have the value of TP, FP, and FN for the semantic role PATIENT, we then count the precision, recall, and F1 with following equations:

$$Precision = \frac{TP}{TP+FP} \tag{3.33}$$

$$Recall = \frac{TP}{TP+FN} \tag{3.34}$$

$$F1 = 2 \cdot \frac{Precission \cdot Recall}{Precission + Recall} \tag{3.35}$$

Other semantic roles, such as AGENTand BENEFICIARY, are also evaluated by following these rules. After we have the value of precision, recall, and F1 for every semantic role, we average them to get the average precision, recall, and F1 for the model being evaluated.

CHAPTER 4
IMPLEMENTATION

This chapter explains the implementations of the methodology explained in chapter 3. It includes the implementation of data annotation and pre-processing, model development, experiment, as well as the evaluation.

4.1 Computer Specification

Every experiment uses GPU-based virtual server provided by Kata.ai. The specifications of the server are explained as follows.

Table 4.1: Server Specifications

GPU	Nvidia Tesla M60
Number of Cores	6
RAM	56.00 GiB
Operating System	Ubuntu 16.04.2

The server uses GPU Nvidia Tesla M60 with 6 cores. The size of the RAM is 56.00 GiB. The operating system used is Ubuntu 16.04.2.

4.2 Data Annotation and Pre-processing

For data annotation, we use an in-house tool provided by Kata.ai, named *kata-annotator*. The total amount of data to be annotated was 9000 sentences. The data was annotated by three linguists with each of them annotating different set of data containing 3000 sentences for 8 weeks. In order to align the annotation understanding between them, the three linguists annotated the same trial set consisting of 100 sentences before starting to annotate the real one. The annotation differences found are then discussed in order to align the understanding between them.

After 8 weeks of annotation, the total amount of data that has been annotated is 8000 sentences. The other 1000 sentences missing is due to one annotator that could not complete the annotation on time. After finish annotating, the tool outputs the tokenized annotation result as JSON in BIO format. An example is given as follows:

```
[{
"data": ["Aku", "pengen", "makan", "ayam", "goreng", "dong"],
"label": ["B-AGENT", "B-MD", "B-PRED", "B-PATIENT", "I-PATIENT", "O"]
},
{
"data": ["Kamu", "gak", "tidur", ",", "Andi", "?"],
"label": ["B-AGENT", "O", "B-PRED", "O", "B-GREET", "O"]
}s]
```

It turns out that there are only 5036 sentences which contain predicate in it. These 5036 sentences are the main data set to be trained and tested.

Pseudocode 4.1: A pseudocode for converting labels of a sentence into one-hot-vectors

```
1 Function convertLabelToOneHotVector(arrLabel) is
      Input  : array of labels of a sentence
      Output: array of one hot vectors
2     oneHotVectorLabel = [];
3     foreach label in arrLabel do
4         oneHotVectorLabel.append(label.convertToOneHotVector())
5     return oneHotVectorLabel;
```

The label is then converted into a one-hot-vector representation which is presented in Pseudocode 4.1. The function *convertLabelToOneHotVector* takes an array of labels from one sentence as an input. Each label in the array is then converted into its one-hot-vector representation. All the converted results are appended into a new array called *oneHotVectorLabel*. The function finally returns the array of one-hot-vectors.

4.3 Feature Extraction

There are three features to be extracted including Word Embedding, POS tag, and Neighboring Word Embeddings. This section explains the implementation of each feature extraction. All codes are implemented using Python.

4.3.1 Word Embedding

We use Gensim's Word2Vec (Rehurek, 2013) as the library for training the word embedding model as well as converting words into vectors. Pseudocode 4.2

explains on how to train the word embedding model.

Pseudocode 4.2: A pseudocode to train word embedding model using Word2Vec

```
1 Function trainWordEmbeddingModel(corpus, contextWindow,
    vectorDimension) is
      Input  : training corpus, context window, vector dimension
      Output: Word2Vec model
2     model = Word2Vec.createModel(corpus, contextWindow,
        vectorDimension)
3     return model;
```

There are two parameters, which are context window and vector dimension. Context window determines the area of interest in building the word embedding model. Vector dimension represents the length of the output vector. In this work, the context window and vector dimension used are 5 and 64, respectively. The output is the trained word embedding model.

Pseudocode 4.3: A pseudocode to transform words into vectors by word embedding model

```
1 Function wordToVector(model, arrWord) is
      Input  : trained word embedding model, array of words of a sentence
      Output: array of word vectors
2     arrVector = [];
3     foreach word in arrWord do
4         arrVector.append(model.getVector(word))
5     return arrVector;
```

Pseudocode 4.3 shows on how to use the trained word embedding model to convert words into vectors. The function initially takes the trained model and an array of words from a sentence. Each word is then converted into the vector using the trained model. The converted words are appended in a new array called *arrVector*. The function finally returns this new array.

4.3.2 POS Tag

For POS Tag feature, we use the gold-standard POS tag annotated by the three linguists. The annotation tool *kata-annotator* is also used for annotating the POS tag. The output example of the POS tag from the tool in a form of JSON is given as follows:

```
[
{
"data": ["Aku", "pengen", "makan", "ayam", "goreng", "dong"],
"label": ["NN", "ADV", "V", "NN", "V", "INTJ"]
},
{
"data": ["Kamu", "gak", "tidur", ",", "Andi", "?"],
"label": ["NN", "NEG", "V", "O", "NN", "O"]
}
]
```

The JSON file consists of an array of sentences. Each word in a sentence is labeled with the POS tag accordingly.

Pseudocode 4.4: A pseudocode for converting POS tags of a sentence into one hot vectors

```
1 Function convertPOSToOneHotVector(arrPOS) is
      Input  : array of POS tags of a sentence
      Output: array of one hot vector
2     posTagFeature = [];
3     foreach pos in arrPOS do
4         posTagFeature.append(pos.convertToOneHotVector())
5     return posTagFeature;
```

Each of the POS tag in a sentence is then converted into one-hot-vector. The implementation is presented in Pseudocode 4.4. The function *convertPOSToOneHotVector* initially takes array of POS tags from a sentence as the input. Each POS tag is converted into its respective one-hot-vector. The results are appended into a new array called *posTagFeature*. The function finally returns this new array as the one-hot-vector representations of POS tags in a sentence.

4.3.3 Neighboring Word Embeddings

For neighboring word embeddings, we extract one vector on the left and one on the right of the word being processed. Pseudocode 4.5 shows the implementation of extracting neighboring word embeddings.

In this function, the parameter that can be set is the *window*. Since we only extract one word on the left and another on the right of the word being processed, the value of parameter *window* is 1.

Pseudocode 4.5: A pseudocode to extract neighboring word embeddings

```
1  Function extractNeighboringWordEmbedding(sentenceVector) is
       Input  : array of word embedding vectors of a sentence
       Output: array of neighboring word embedding vectors
2      window = 1;
3      vectorDimension = getVectorDimension(sentenceVector);
4      padded = window * [vectorDimension*[0.]] + sequence + window *
         [vectorDimension*[0.]];
5      neighboringVectors = [];
6      for i in 0...sentenceVector.length - 1 do
7          left = [item for sublist in padded[i:(i + window)] for item in sublist];
8          right = [item for sublist in padded[(i+ window + 1):(i + (2*window +
             1))] for item in sublist];
9          concate = left + right;
10         neighboringVectors.append(concate);
11     end
12     return neighboringVectors;
13 end
```

4.4 Model Architecture

As explained in chapter 3, the model architecture consists of main layer and additional layer. The main layer choices include vanilla LSTM (LSTM), Bi-Directional LSTM (BLSTM), Deep BLSTM (BLSTM), DBLSTM-Zhou, and DBLSTM-Highway. On the other hand, the additional layer options include Convolutional Neural Networks (CNN) and attention mechanism.

We use Keras 2.0 (Chollet, 2015) as our deep learning library with Tensorflow 1.1.0 backend for all the architectures. For constructing the model, we use the Functional API of Keras. The model in Keras only receives an input data with a fixed number of time steps for all sentences. Suppose that the maximum number of time steps in our data is l. Thus, all sentences in our data whose number of time steps is lower than l have to be padded with vector $\vec{0}$ in order to get a fixed number of time steps of l. To do the padding, we use *padsequences* function available from Keras. All codes are implemented using Python.

4.4.1 Main Layers

The main layer choices include vanilla LSTM (LSTM), Bi-Directional LSTM (BLSTM), Deep BLSTM (BLSTM), DBLSTM-Zhou, and DBLSTM-Highway. This section explains the implementation of each architecture.

4.4.1.1 Vanilla LSTM (LSTM)

Vanilla LSTM (LSTM) consists of only one layer of forward LSTM. Pseudocode **??** shows the implementation of the LSTM architecture.

Pseudocode 4.6: A pseudocode for building and training vanilla LSTM architecture

```
1  Function lstm(xTrain, yTrain, timesteps, features, xTest, yTest) is
       Input  : x train, y train, number of time steps, number of features, x test,
                y test
       Output: trained model, testing prediction result
2      input = Input(shape=(timesteps, features));
3      forward = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(input);
4      dropout = Dropout(0.2)(forward);
5      output = TimeDistributed(Dense(units=17,
         activation='softmax'))(dropout);
6      model = Model(inputs=[input], outputs=[output]);
7      model.compile(loss='categoricalCrossentropy', optimizer='adam');
8      model.fit(xTrain, yTrain, epochs=50, batchSize=150);
9      prediction = model.predict(xTest);
10     return model, prediction;
```

The Pseudocode 4.6 takes x train, y train, number of time steps, number of features, x test and y test as the inputs. The model starts with the defining the input layer, with the input shape of (timesteps, features). The input layer is then connected to the LSTM layer thas has 128 hidden units. These hidden units are the output of the LSTM layer. We use recurrent dropout in LSTM, as recommended by He et al. (2017). The recurrent dropout used is 0.2. We also use dropout layer on top of the LSTM layer by the value of 0.2. The output of the dropout layer is connected to the time-distributed dense layer with softmax activation function. These last layer produces the labels of semantic roles. The model is trained with categorical crossentropy loss function and Adam optimizer. The number of epochs and batch size used are both 50. After the model has been trained, we use it to predict the semantic roles the x test data which later will be evaluated. The function returns the trained model as well as the prediction result of the test data.

4.4.1.2 Bi-Directional LSTM (BLSTM)

Bi-Directional LSTM (BLSTM) consists of two layers of LSTM. The first layer is moving forward and the other is moving backward. Pseudocode 4.7 shows the

implementation of the BLSTM architecture.

Pseudocode 4.7: A pseudocode for building and training BLSTM architecture

```
1  Function blstm(xTrain, yTrain, timesteps, features, xTest, yTest) is
       Input  : x train, y train, number of time steps, number of features, x test,
                y test
       Output: trained model, testing prediction result
2      input = Input(shape=(timesteps, features));
3      forward = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(input);
4      backward = LSTM(units=128, returnSequences=True,
         goBackwards=True, recurrentDropout=0.2)(input);
5      merged = Concatenate([forward, backward]);
6      dropout = Dropout(0.2)(merged);
7      output = TimeDistributed(Dense(units=17,
         activation='softmax'))(dropout);
8      model = Model(inputs=[input], outputs=[output]);
9      model.compile(loss='categoricalCrossentropy', optimizer='adam');
10     model.fit(xTrain, yTrain, epochs=50, batchSize=150);
11     prediction = model.predict(xTest);
12     return model, prediction;
```

The Pseudocode 4.7 takes x train, y train, number of time steps, number of features, x test and y test as the inputs. The model starts with the defining the input layer, with the input shape of (timesteps, features). The input layer is then connected to two LSTM layers: forward layer and backward layer. Both forward and backward LSTM layers have 128 hidden units. The recurrent dropout used is 0.2. The output of both forward and backward layers are concatenated, resulting a vector whose length is 256. We also use dropout layer on top of the concatenated layer by the value of 0.2. The output of the dropout layer is connected to the time-distributed dense layer with softmax activation function. This last layer produces the labels of semantic roles. The model is trained with categorical crossentropy loss function and Adam optimizer. The number of epochs and batch size used are 50 and 150, respectively. After the model has been trained, we use it to predict the semantic roles the x test data which later will be evaluated. The function returns the trained model as well as the prediction result of the test data.

4.4.1.3 Deep BLSTM (DBLSTM)

Deep BLSTM (BLSTM) consists of two pairs of BLSTM. Pseudocode 4.8 shows the implementation of the BLSTM architecture.

Pseudocode 4.8: A pseudocode for building and training DBLSTM architecture

```
1  Function blstm(xTrain, yTrain, timesteps, features, xTest, yTest) is
       Input  : x train, y train, number of time steps, number of features, x test,
                y test
       Output: trained model, testing prediction result
2      input = Input(shape=(timesteps, features));
3      forward1 = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(input);
4      backward1 = LSTM(units=128, returnSequences=True,
         goBackwards=True, recurrentDropout=0.2)(input);
5      merged1 = Concatenate([forward, backward]);
6      dropout1 = Dropout(0.2)(merged);
7      forward2 = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(dropout1);
8      backward2 = LSTM(units=128, returnSequences=True,
         goBackwards=True, recurrentDropout=0.2)(dropout1);
9      merged2 = Concatenate([forward2, backward2]);
10     dropout2 = Dropout(0.2)(merged2);
11     output = TimeDistributed(Dense(units=17,
         activation='softmax'))(dropout2);
12     model = Model(inputs=[input], outputs=[output]);
13     model.compile(loss='categoricalCrossentropy', optimizer='adam');
14     model.fit(xTrain, yTrain, epochs=50, batchSize=150);
15     prediction = model.predict(xTest);
16     return model, prediction;
```

The input layer is first connected to two LSTM layers, the first one for going forward and another for going backward. Both forward and backward LSTM layers have 128 hidden units. The recurrent dropout used is 0.2. The output of both forward and backward layers are concatenated, resulting a vector whose length is 256. We also use dropout layer on top of the concatenated layer by the value of 0.2. The output of the dropout layer is then processed by the next two LSTM layers using the same way as the previous steps. The output of both layers are also concatenated and processed by the dropout layer. The output of the final dropout layer is connected to the time-distributed dense layer with softmax

activation function. This last layer produces the labels of semantic roles. Likewise in the previous architectures, the function returns the trained model as well as the prediction result of the test data.

4.4.1.4 DBLSTM-Zhou

Deep BLSTM-Zhou (DBLSTM-Zhou) consists of two pairs of BLSTM-Zhou. Pseudocode 4.9 shows the implementation of the DBLSTM-Zhou architecture.

Pseudocode 4.9: A pseudocode for building and training DBLSTM-Zhou architecture

```
1  Function blstm(xTrain, yTrain, timesteps, features, xTest, yTest) is
       Input  : x train, y train, number of time steps, number of features, x test,
                y test
       Output: trained model, testing prediction result
2      input = Input(shape=(timesteps, features));
3      forward1 = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(input);
4      dropout1f = Dropout(0.2)(forward1);
5      backward1 = LSTM(units=128, returnSequences=True,
         goBackwards=True, recurrentDropout=0.2)(dropout1f);
6      dropout1b = Dropout(0.2)(backward1);
7      forward2 = LSTM(units=128, returnSequences=True,
         recurrentDropout=0.2)(input);
8      dropout2f = Dropout(0.2)(forward2);
9      backward2 = LSTM(units=128, returnSequences=True,
         goBackwards=True, recurrentDropout=0.2)(dropout2f);
10     dropout2b = Dropout(0.2)(backward2);
11     output = TimeDistributed(Dense(units=17,
         activation='softmax'))(dropout2b);
12     model = Model(inputs=[input], outputs=[output]);
13     model.compile(loss='categoricalCrossentropy', optimizer='adam');
14     model.fit(xTrain, yTrain, epochs=50, batchSize=150);
15     prediction = model.predict(xTest);
16     return model, prediction;
```

The input layer is first processed by the first forward LSTM layer. The output of this layer is then connected to the first dropout layer. After that, the dropout output is processed by the first backward LSTM layer, followed by the second dropout layer. At this point, one pair of BLSTM-Zhou is built. These processes are done one more time to build the second stack of BLSTM-Zhou. The output of the last

layer is then connected to the time-distributed dense layer with softmax activation function. This last layer produces the labels of semantic roles. As in the previous functions, this function also returns the trained model as well as the prediction result of the test data.

4.4.1.5 DBLSTM-Highway

Deep BLSTM-Zhou (DBLSTM-Highway) consists of two pairs of BLSTM-Highway. To implement the DBLSTM-Highway, line 3 to 10 in Pseudocode 4.9 are changed into the codes presented in Pseudocode 4.10.

Pseudocode 4.10: A pseudocode for building and training DBLSTM-Highway architecture

```
1  forward1 = LSTM(units=128, returnSequences=True,
     recurrentDropout=0.2)(input);
2  dropout1f = Dropout(0.2)(forward1);
3  backward1 = LSTM(units=128, returnSequences=True, goBackwards=True,
     recurrentDropout=0.2)(dropout1f);
4  merged1b = Concatenate([backward1, dropout1f]);
5  alpha1b = TimeDistributed(Dense(1, activation='sigmoid'))(merged1b);
6  out1b = Lambda(timesAlphaSigmoid)([alpha1b, backward1, dropout1f]);
7  dropout1b = Dropout(0.2)(out1b);
8  forward2 = LSTM(units=128, returnSequences=True, goBackwards=True,
     recurrentDropout=0.2)(dropout1b);
9  merged2f = Concatenate([forward2, backward1]);
10 alpha2f = TimeDistributed(Dense(1, activation='sigmoid'))(merged2f);
11 out2f = Lambda(timesAlphaSigmoid)([alpha2f, forward2, backward1]);
12 dropout2f = Dropout(0.2)(out2f);
13 backward2 = LSTM(units=128, returnSequences=True, goBackwards=True,
     recurrentDropout=0.2)(dropout2f);
14 merged2b = Concatenate([backward2, forward2]);
15 alpha2b = TimeDistributed(Dense(1, activation='sigmoid'))(merged2b);
16 out2b = Lambda(timesAlphaSigmoid )([alpha2b, backward2, forward2]);
17 dropout2b = Dropout(0.2)(out2b);
```

The input layer is first processed by the first forward LSTM layer. The output of this layer is then connected to the first dropout layer. After that, the output is processed by the first backward LSTM layer. The output of the first backward layer (*backward1*) and the first dropout layer (*dropout1f*) are concatenated before being processed by the time-distributed dense layer with sigmoid function. The output of the dense layer is the weight *alpha2b*. In lambda function called *timesAlphaSigmoid*, *backward1* is multiplied by the weight *alpha2b* and *dropout1f* is

multiplied by the weight *1 - alpha2b*. The lambda function then sums both results. These steps are also done for the next forward and the backward LSTM layers. As in the previous architectures, the output of the last layer is then connected to the time-distributed dense layer with softmax activation function to predict the semantic roles.

4.4.2 Additional Layers

The additional layer options consist of Convolutional Neural Networks (CNN) and attention mechanism. These additional layers can be used in addition to any of the main layers.

4.4.2.1 Convolutional Neural Networks (CNN)

CNN-BLSTM adds a CNN layer underneath the BLSTM layers. Pseudocode 4.11 shows the implementation of the CNN-BLSTM architecture.

Pseudocode 4.11: A pseudocode for adding CNN layer underneath the main layer

```
1 inputLayer = Input(shape=(timesteps, features));
2 cnnLayer = Conv1D(filters=128, kernelSize=3, padding='same',
    activation='relu', strides=1)(inputLayer);
```

In this architecture, the CNN layer is added before the BLSTM layers. Thus, the input layer is connected with the CNN layer. The parameters of the CNN layer are filters, kernel size, and strides. Filters represent the number of output hidden layers. Kernel size defines the context window of the CNN layer. Strides define the amount of slide for every time step. The value of filters, kernel size, and strides parameters are 128, 3, and 1, respectively. The output the CNN layer is then connected to the any LSTM variant as explained in the previous sections.

4.4.2.2 Attention Mechanism

The attention mechanism can be added on top of the any main layer variant. Pseudocode 4.12 shows the implementation of the adding attention mechanism on top of the main layer architecture.

The output of the dropout layer from last LSTM layer is first fed into a time-distributed, raw dense layer. The output dimension of this dense layer is set to 512. The output of this layer is then summed for each time step. The result of this sum is then fed into the dense layer with softmax activation function. This dense layer

Pseudocode 4.12: A pseudocode for adding attention mechanism on top of the main layer architecture

```
1 outs1 = TimeDistributed(RawDense(outputDim=512))(dropout);
2 m = Lambda(sum)(outs1);
3 alpha = Dense(timesteps, activation="softmax")(m);
4 z = Lambda(timesAlphaSum)([alpha, dropout]);
5 dropoutZ = Dropout(0.2)(z);
6 repeatedZ = RepeatVector(timesteps)(dropoutZ);
7 outFinal = concatenate([dropout, repeatedZ]);
8 output = TimeDistributed(Dense(units=17, activation='softmax'))(outFinal);
```

outputs the weights alpha for each time step. Lambda function *timesAlphaSum* firstly multiplies the original dropout output of the last LSTM layer with the alpha values. The function then sums all the result element-wise in order to get a context vector z. Vector z is then duplicated by the number of time steps before it is concatenated with all the original dropout output of the last LSTM layer. These concatenated vectors are then fed into the last softmax layer to produce the output labels.

CHAPTER 5
EXPERIMENTS

This chapter focuses on the experiments that we have conducted. Firstly, the data statistics about the semantic role distribution are presented. The experiment scenarios are then explained, followed by the results and analyses of this work.

5.1 Data Statistics

As explained in Chapter 3, there are 8 labels to be predicted, which are: AGENT, PREDICATE, PATIENT, MODAL, BENEFICIARY, LOCATION, GREET, and TIME. The distribution of the labels in our data set is presented in Figure 5.1.

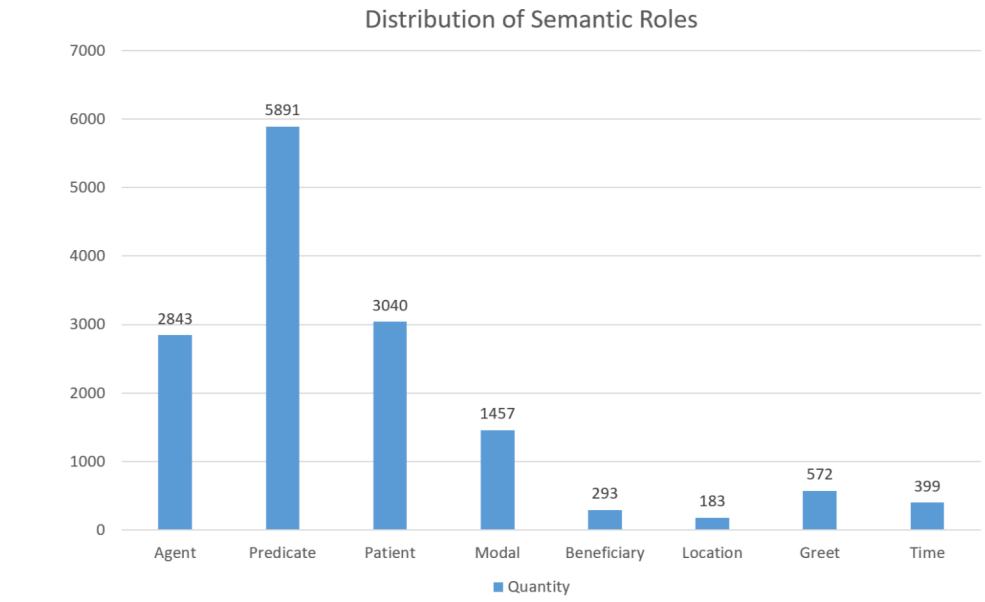

Figure 5.1: Label Distribution

5.2 Experiment Scenario

Our experiment consists of two scenario sets: feature selection and model selection. In feature selection, we experiment to find which feature combination outputs the best result. For model selection, the experiment focuses on finding the best model architecture to get the highest result.

The feature selection set consists of 4 combination scenarios, which are:

1. Word Embedding (WE)
2. Word Embedding + Neighboring Word Embeddings (WE + NW)
3. Word Embedding + POS Tag (WE + POS)
4. Word Embedding + POS Tag + Neighboring Word Embeddings (WE + POS + NW)

In this scenario set, we use vanilla LSTM for the base architecture. Out of 4 combinations, we pick the best two to be experimented in the model selection scenario set.

The model selection consists of 4 scenario sets. The first set aims to find which feature combination is better when the LSTM layers are added. The next set compares the performance between BLSTM, BLSTM Zhou, and BLSTM Highway. The third set aims to see the effect of using CNN layer. Lasltly, the fourth set aims to see the impact of our proposed attention layer towards the performance.

5.2.1 Scenario 1: Feature Selection

Table 5.1 shows the results of the afore-mentioned four feature combinations. The highest result is achieved with the combination of WE + POS + NW, followed by WE + POS, WE + NW, and WE, with F1 scores of 79.76%, 79.10%, 73.76%, and 73.12%, respectively.

Table 5.1: Results of Feature Selection Scenario, in percentage

Feature	Precision	Recall	F1
WE	75.20	71.16	73.12
WE + NW	76.07	71.58	73.76
WE + POS	80.09	78.14	79.10
WE + POS + NW	80.96	78.60	79.76

Our first experiment uses only word embedding (WE) as the feature. For a starting point, the results are promising with 75.20% precision, 71.16% recall and 73.12% F1. Table 5.2 shows the precision, recall, and F1 scores for each semantic role when only word embedding is used as the feature.

Table 5.2: Precision, Recall, F1 scores of each label for scenario WE

Label	Precision	Recall	F1
AGENT	74.53	86.74	80.17
PREDICATE	88.01	89.46	88.73
PATIENT	74.30	68.68	71.38
MODAL	78.14	76.74	77.43
BENEFICIARY	68.68	66.29	67.47
LOCATION	68.31	55.69	61.36
GREET	69.13	52.08	59.40
TIME	80.58	73.61	76.94
Average	75.21	71.16	73.12

As we can see from the table above, the top three semantic roles with the highest F1 scores are PREDICATE (88.72%), AGENT (80.16%), and MODAL (77.41%). On the other hand, the top three semantic roles with the lowest F1 scores are GREET (59.21%), LOCATION (61.26%), and BENEFICIARY (66.85%). We suggest that these results are partially caused by the unbalanced distribution of semantic roles as presented in Figure 5.1. PREDICATE, AGENT, and MODAL are on the top of the distribution while GREET, LOCATION, and BENEFICIARY are among the lowest. When we analyze the prediction results, some interesting findings are found. The model often mispredicts GREET as AGENT as presented in Table 5.3

Table 5.3: Misprediction example of GREETwhen using only WE as the feature

Sentence	jemma	kamu	kenal	dito	enggak
Expected	**B-GREET**	B-AGENT	B-PREDICATE	B-PATIENT	O
Predicted	**B-AGENT**	B-AGENT	B-PREDICATE	B-PATIENT	O

In this example, "jemma" should be labeled as GREET while "kamu" should be labeled as AGENT. This example illustrates a tricky situation in which the model has to decide which one is the AGENT or GREET since both are located side by side. The model from our first experiment (WE) cannot differentiate this and therefore, it predicts both "jemma" and "kamu" as AGENT. We thus propose to use Neighboring Word Embeddings (NW) as the additional feature. With this new feature, the machine will have the additional information from the surrounding words. In the previous example, the machine can determine that "jemma" should be labeled as GREET when it knows that the following word is "kamu", which most likely will be an AGENT or PATIENTin a sentence.

When we try to use Neighboring Word Embeddings (NW) as the additional feature (experiment WE + NW), the F1 result slightly increases from 73.12% to 73.76%. The most significant improvement is the label GREET whose F1 score increases from 59.21% to 71.41%. While in the previous experiment (WE), the model mispredicts GREET as AGENT, the model from this experiment (WE + N) predicts it correctly, as presented in Table 5.4:

Table 5.4: Correct prediction example of GREETwhen using WE + NW as the features

Sentence	jemma	kamu	kenal	dito	enggak
Expected	**B-GREET**	B-AGENT	B-PREDICATE	B-PATIENT	O
Predicted	**B-GREET**	B-AGENT	B-PREDICATE	B-PATIENT	O

This proves that the surrounding words can be a helpful information to distinguish when to predict a token as GREET or AGENT.

Another interesting finding that we found in the results of WE experiment is presented in Table 5.5.

Table 5.5: Misprediction example of TIMEwhen using WE as the feature

Sentence	skrg	lgi	main
Expected	**B-TIME**	B-MODAL	B-PREDICATE
Predicted	**O**	B-MODAL	B-PREDICATE

The example illustrates that the machine does not predict "skrg" (formal: "sekarang") as TIME, but rather labels it as "other". This case also happens to other semantic roles including AGENT, PATIENT, BENEFICIARY, LOCATION, and GREET, where the machine mispredicts them as "other". This may be caused by the wide variety of abbreviations in informal language that the training data may not contain. This is due to the fact that our data set is relatively small (5,035 instances) compared to other SRL data set for formal language, such as ConLL 2005 data set with 39,832 instances Carreras and Màrquez (2005). For instance, "sekarang" has a lot of informal forms such as "skrg", "skg", and "skrng". Our training data may not contain one of its informal forms, "skrg", and thus, the model mispredicts it as "other".

To address this issue, we argue that POS tags provide meaningful information to increase all semantic role results. AGENT, PATIENT, BENEFICIARY, LOCATION, GREET, and TIME are all *nouns*. Adding the POS tag *noun* information will give a hint to the model in order to determine which tokens are the candidates of these semantic roles. In the previous example, the machine may predict "skrg" as TIME if

it knows that the word's POS tag is *noun*. In addition, PREDICATE is often in a form of *verb*, though in some cases of Indonesian language, the PREDICATE is an *adjective*. Therefore, knowing that a token is a verb will help the machine to predict it as a PREDICATE. Lastly, MODAL is mostly in a form of adverb ("akan", "lagi", "harus"). Adding information that tells a word is an adverb will help the model to predict MODAL as well. Based on this rationale, we believe that POS tag will increase the performance of all semantic roles.

When we add POS tag as our additional feature (experiment WE + POS), the average F1 score dramatically increases from 73.12% (WE) to 79.10% (WE + POS) as presented in Table 5.1. The improvements of all semantic roles when we compare experiment WE and WE + POS are presented in Table 5.6.

Table 5.6: Precision, Recall, F1 scores (in %) of each label for scenario WE and WE + POS

	WE			WE + POS		
Label	Precision	Recall	F1	Precision	Recall	F1
AGENT	74.53	86.74	80.17	75.03	88.93	81.39
PREDICATE	88.01	89.46	88.73	96.36	97.07	96.71
PATIENT	74.30	68.68	71.38	78.48	75.57	77.00
MODAL	78.14	76.74	77.43	84.49	82.08	83.27
BENEFICIARY	68.68	66.29	67.47	73.91	71.01	72.43
LOCATION	68.31	55.69	61.36	72.35	67.68	69.93
GREET	69.13	52.08	59.4	71.28	56.01	62.73
TIME	80.58	73.61	76.94	88.87	86.78	87.81
Average	75.21	71.16	73.12	80.10	78.14	79.10

All the semantic roles with POS tag *noun* (AGENT, PATIENT, BENEFICIARY, LOCATION, GREET, and TIME) are all having improvements in terms of the F1 scores. The most notable improvements are TIME (76.85% to 87.72%), LOCATION (61.26% to 69.79%), and BENEFICIARY (66.85% to 72.02%). Referring to the previous example, the new model correctly predicts "skrg" as TIMErather than "other", as presented in Table 5.7. On the other hand, the PREDICATE improves from 88.72% to 96.71% while MODAL increases from 77.41% to 83.21%. These results prove that POS tag provides a significant contribution to increase the performance. While most of the deep learning research aims to not using hand-crafted features like POS tag, that is not the case in our work. Our experiment shows that such features are still important when the data set is relatively small. Nonetheless, building a huge data set is also costly, we therefore have to find a workaround in order to achieve competitive results by only using

small data set. In this case, the workaround is to add traditional features such as POS tag.

Table 5.7: Correct prediction example of TIME when using WE + POS as the features

Sentence	skrg	lgi	main
Expected	**B-TIME**	B-MODAL	B-PREDICATE
Predicted	**B-TIME**	B-MODAL	B-PREDICATE

Finally, we try to add Neighboring Words in addition to experiment WE + POS. Hence, the feature combination is WE + POS + N. The F1 score slightly increases from 79.10% (WE + POS) to 79.76% (WE + POS + N). In this improvement, the F1 score of GREETincreases from 62.65% (WE + POS) to 75.02% (WE + POS + N). This is the same case as the improvement that we have seen in experiment WE and WE + N.

In this feature selection scenarios, WE + POS and WE + POS + N are the top two feature combinations that output the highest results in terms of F1 score. Therefore, we use both feature combinations in the next scenario set: model selection.

5.2.2 Scenario 2: Model Selection

The model selection set consists of four scenario sets, which are: a.) LSTM, BLSTM, and Stacked BLSTM; b.) DBLSTM, DBLSTM-Zhou, and DBLSTM-Highway; c.) CNN layer; and d.) Attention layer.

5.2.2.1 Scenario 2a: LSTM, BLSTM and DBLSTM

For model selection set, we first experiment on comparing the top two feature combinations from the previous set (WE + POS and WE + POS + N) when using vanilla LSTM, BLSTM, and Stacked BLSTM (DBLSTM) architectures. Table 5.8 shows the results of this experiment.

Table 5.8: F1 scores (in %) of WE + POS and WE + POS + NW when using vanilla LSTM (LSTM), BLSTM, and stacked BLSTM (DBLSTM) architectures.

Feature	LSTM	BLSTM	DBLSTM
WE + POS	79.10	78.93	**82.30**
WE + POS + N	79.76	79.12	79.87

When using vanilla LSTM, the F1 scores of WE + POS and WE + POS + N are 79.10% and 79.76%, respectively. When bi-directional LSTM (BLSTM)

is used, the performances of both feature combinations slightly decrease: WE + POS decreases from 79.10% to 78.93%, and WE + POS + NW decreases from 79.76%. However, when two BLSTM layers are stacked (DBLSTM), the F1 scores of both feature combinations increase compared to the vanilla LSTM architecture. While WE + POS + NW slightly increases by 0.11% (from 79.76% to 79.87%), the feature combination WE + POS improves by 3.20% (from 79.10% to 82.30%). According to Zhou and Xu (2015), bi-directional LSTM architecture is able to implicitly capture the context information from the past (words on the left) and future (words on the right). Moreover, their research also shows that the stacked BLSTM architecture (DBLSTM) could enhance the model performance. In this case, we suggest that the BLSTM model, especially stacked BLSTM, does not need explicit context information such as neighboring word embeddings. Therefore, by only using WE + POS as the features, combined with stacked BLSTM, the model can achieve compelling result.

Since the WE + POS feature combination and stacked BLSTM (DBLSTM) architecture outputs the highest result in this case, it will be used for the next model selection scenarios.

5.2.2.2 Scenario 2b: DBLSTM, DBLSTM-Zhou, and DBLSTM-Highway

In this scenario set, we compare the stacked BLSTM (DBLSTM), stacked BLSTM-Zhou (DBLSTM-Zhou), and stacked BLSTM-Highway (DBLSTM-Highway). Table 5.9 shows the precision, recall, and F1 scores of each architecture.

Table 5.9: Precision, Recall, and F1 scores (in %) of Stacked BLSTM (DBLSTM), Stacked BLSTM-Zhou (DBLSTM-Zhou), and Stacked BLSTM-Highway (DBLSTM-Highway)

Model	Precision	Recall	F1
DBLSTM	**82.97**	**81.66**	**82.30**
DBLSTM-Zhou	81.52	81.15	81.33
DBLSTM-Highway	80.63	81.66	81.14

The highest F1 score is achieved by the Stacked BLSTM architecture (82.30%), followed by Stacked BLSTM-Zhou (81.33%) and Stacked BLSTM-Highway (81.14%). In their work, Zhou and Xu (2015) did not elaborate on why the pick their proposed architecture (Stacked BLSTM-Zhou) rather than the original Stacked BLSTM. They did not compare the performances between original Stacked BLSTM and their proposed architecture. Moreover, He et al. (2017) who proposed the modification of Zhou's architecture (Stacked BLSTM-Highway) also did not

explain the rationale of using Zhou's BLSTM rather than the original one. In our case, it turns out that the original Stacked BLSTM still outperforms the other BLSTM architectures. However, we are interested in using all these three architectures for the next model selection scenarios which aim to see the impact of using two additional layers: CNN and Attention layers.

5.2.2.3 Scenario 2c: CNN Layer

In this scenario set, we examine the impact of adding the additional CNN layer underneath the three afore-mentioned architectures: Stacked BLSTM, Stacked BLSTM-Zhou, and Stacked BLSTM-Highway. Table 5.9 shows the results of this scenario set.

Table 5.10: The precision, recall, and F1 scores of the stacked BLSTM architectures with and without CNN layer.

	Without CNN			With CNN		
Model	Precision	Recall	F1	Precision	Recall	F1
DBLSTM	82.97	81.66	82.30	81.09	81.98	81.53
DBLSTM-Zhou	81.52	81.15	81.33	75.58	78.17	76.85
DBLSTM-Highway	80.63	81.66	81.14	79.47	78.68	79.07

The F1 scores of all three architectures decrease when the CNN layer is added. The Stacked BLSTM model drops by 0.77% (from 82.30% to 81.53%), Stacked BLSTM-Zhou model significantly decreases by 4.48% (from 81.33% to 76.85%), and Stacked BLSTM-Highway model drops by 2.07% (from 81.14% to 79.07%). The rationale of using CNN layer is to capture context information from the features. However, from all these results, we suggest that the way CNN layer extracts context information may not work well with the way Stacked BLSTM architectures do. This analysis is supported by another experiment in which we found an improvement from 79.10% to 79.50% (F1 Score) when CNN layer is added underneath the vanilla LSTM. In this case, we believe that CNN layer works well with a simple, one layer LSTM.

5.2.2.4 Scenario 2d: Attention Layer

The last scenario set aims to see the impact of using our proposed attention layer on top of the three stacked BLSTM architectures. Table 5.11 shows the results of this scenario set.

Table 5.11: The precision, recall, and F1 scores of the stacked BLSTM architectures with and without attention layer.

	Without Attention			With Attention		
Model	Precision	Recall	F1	Precision	Recall	F1
DBLSTM	82.97	81.66	82.30	82.36	82.07	82.21
DBLSTM-Zhou	81.52	81.15	81.33	82.32	81.62	81.97
DBLSTM-Highway	80.63	81.66	81.14	**83.07**	**82.30**	**82.68**

As we can see from the table above, the original Stacked BLSTM (DBLSTM) model slightly decreases by 0.09% when using attention layer (from 82.30% to 82.21%). On the other hand, the performances of both DBLSTM-Zhou and DBLSTM-Highway models increase when attention layer is used. While DBLSTM-Zhou model increases by 0.64% (from 81.33% to 81.97%), the DBLSTM-Highway model significantly increases by 1.54% (from 81.14% to 82.68%). From these results, we suggest that the attention layer successfully contributes to the DBLSTM-Zhou and DBLSTM-Highway models, while the layer may not suitable to be used on top of original DBLSTM architecture.

CHAPTER 6
CONCLUSION AND FUTURE WORK

Semantic Role Labeling (SRL) is an integral part of understanding semantic information of a text. One of its applications is to make chat bots understand user's chat better and thus, it can provide more engaging answers. Even though the SRL on English formal language has been widely studied, only a few reports exist for informal conversational language, especially for language being used in the chatbot system. In Indonesian, both formal and conversational language are barely tapped for building SRL system. In this work, we focus on solving SRL for Indonesian conversational language. Our contributions are introducing a new set of semantic roles and proposing an attention mechanism on top of the Long Short-Term Memory Networks architecture.

We view SRL as a sequence labeling problem. One of the deep learning architectures that fits to solve sequence labeling problem is Long Short-Term Memories (LSTM), a modification of Recurrent Neural Networks (RNN). In this work, we experiment on a variety of LSTM networks, including vanilla LSTM, Bi-Directional LSTM (BLSTM), Deep BLSTM (DBLSTM), DBLSTM-Zhou, and DBLSTM-Highway. The features used are Word Embedding (WE), Part-of-Speech Tag (POS), and Neighboring Word Embeddings (NW).

We conducted two set of experiment scenarios: feature selection and model selection. Our experiments on feature selection show that when the size of training data is relatively small, one still needs to use traditional feature such as POS tag in addition to word embedding. The neighboring word embeddings feature can slightly improve the results, especially for the label GREET. By using vanilla LSTM, the top two best feature combinations are WE + POS + NW and WE + POS with the F1 scores of 79.76% and 79.10%, respectively.

In model selection set, our experiments show that DBLSTM architecture can outperform the vanilla LSTM and BLSTM. The results also show that when using DBLSTM, the feature combination WE + POS can outperform the WE + POS + NW combination. Furthermore, our results show that the original DBLSTM can outperform its variants: DBLSTM-Zhou and DBLSTM-Highway. For the additional layers, we experiment on adding CNN layer and our proposed attention mechanism to the three afore-mentioned DBLSTM architectures. It turns out that the CNN layer is not suitable for the DBLSTM networks. In

our experiment of adding the attention mechanism, the DBLSTM architecture's performance slightly decreases while the results of both DBLSTM-Zhou and DBLSTM-Highway improve. The highest result is achieved by using DBLSTM-Highway with the attention mechanism with F1 score of 81.26%.

For future works, there is a lot of possibilities to design the architectures. For example, instead of putting the attention layer on top of the main layer, one can add it right after the input layer. Another interesting architecture design would be combining the output of CNN layer and the original input before going to the main layer. Beside that, one can experiment on many features such as wordshape, prefix, suffix, and neighboring POS tags to enhance the performance. Once the SRL system is established, one can focus on building the Natural Language Generation (NLG) system for chat bots based on the semantic roles of the Indonesian conversational language. This way, one can create more intelligent chat bots which understand deeper on conversational language. Another interesting work would be integrating coreference resolution on the SRL system knowing that conversational language is usually in a form of dialogues.

REFERENCES

Baker, C. F., Fillmore, C. J., and Lowe, J. B. (1998). The berkeley framenet project. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*, pages 86–90. Association for Computational Linguistics.

Bengio, Y., LeCun, Y., et al. (2007). Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5).

Carreras, X. and Màrquez, L. (2005). Introduction to the conll-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*, pages 152–164. Association for Computational Linguistics.

Chollet, F. (2015). keras. https://github.com/fchollet/keras.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

Dowty, D. (1991). Thematic proto-roles and argument selection. *language*, pages 547–619.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Emanuele, B., Castellucci, G., Croce, D., and Basili, R. (2013). Textual inference and meaning representation in human robot interaction. In *Joint Symposium on Semantic Processing.*, page 65.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational linguistics*, 28(3):245–288.

Gildea, D. and Palmer, M. (2002). The necessity of parsing for predicate argument recognition. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 239–246. Association for Computational Linguistics.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

He, L., Lee, K., Lewis, M., and Zettlemoyer, L. (2017). Deep semantic role labeling: What works and whatâĂŹs next. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

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

Jordan, M. I. (1986). Attractor dynamics and parallellism in a connectionist sequential machine.

Jurafsky, D. and James, H. (2016). Speech and language processing (3rd ed. draft).

Kingsbury, P. and Palmer, M. (2002). From treebank to propbank. In *LREC*, pages 1989–1993. Citeseer.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Liu, D. and Gildea, D. (2010). Semantic role features for machine translation. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 716–724. Association for Computational Linguistics.

Lo, C.-k., Addanki, K., Saers, M., and Wu, D. (2013). Improving machine translation by training against an automatic semantic frame based evaluation metric. In *ACL (2)*, pages 375–381.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2014). word2vec.

Moschitti, A., Morarescu, P., and Harabagiu, S. M. (2003). Open domain information extraction via automatic semantic labeling. In *FLAIRS conference*, pages 397–401.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–43.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J. H., and Jurafsky, D. (2005). Semantic role labeling using different syntactic views. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 581–588. Association for Computational Linguistics.

Punyakanok, V., Roth, D., and Yih, W.-t. (2008). The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2):257–287.

Rehurek, R. (2013). gensim word2vec.

Rohman, W. N. (2017). Pengenalan entitas kesehatan pada forum kesehatan online dengan menggunakan recurrent neural networks. Bachelor's thesis, Universitas Indonesia, Kampus UI Depok.

Saeed, J. (1997). I. 2003. semantics. *GB: Blackwell Publishing*.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.

Seki, K. and Mostafa, J. (2003). A probabilistic model for identifying protein names and their name boundaries. In *Bioinformatics Conference, 2003. CSB 2003. Proceedings of the 2003 IEEE*, pages 251–258. IEEE.

Shen, D. and Lapata, M. (2007). Using semantic roles to improve question answering. In *EMNLP-CoNLL*, pages 12–21.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385.

Surdeanu, M., Harabagiu, S., Williams, J., and Aarseth, P. (2003). Using predicate-argument structures for information extraction. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 8–15. Association for Computational Linguistics.

Zhou, J. and Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. In *ACL (1)*, pages 1127–1137.

APPENDIX