Université de Montréal

Empirical Study and Multi-task Learning Exploration for Neural Sequence Labeling Models

par

Peng Lu

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en informatique

April, 2019

Université de Montréal

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Ce mémoire intitulé

Empirical Study and Multi-task Learning Exploration for Neural Sequence Labeling Models

Présenté par

Peng Lu

A été évalué par un jury composé des personnes suivantes

Jian-Yun Nie
Président-rapporteur

Philippe Langlais
Directeur de recherche

Pascal Vincent
Membre du jury

Sommaire

Les modèles de réseau de neuronaux ont attiré une attention considérable pour l'étiquetage de séquence. Comparés aux modèles traditionnels, les modèles neuronaux offrent de meilleures performances avec moins ou pas d'ingénierie de traits caractéristiques. Cependant, en raison de la sensibilité du cadre expérimental, il est toujours difficile de reproduire et de comparer l'efficacité de différents modèles dans des conditions identiques. Et même si ces modèles peuvent être appliqués à différentes tâches d'étiquetage de séquence, telles que la reconnaissance d'entités nommées (NER), la segmentation de texte et l'étiquetage morphosyntaxique (POS), les travaux antérieurs ne donnent pas de meilleures performances quand ils sont placés dans un processus d'apprentissage multitâche (MTL) que d'apprendre chaque tâche individuellement.

Nous étudions les principaux facteurs d'influence sur la performance des systèmes d'étiquetage de séquences neuronaux en réimplémentant douze modèles d'étiquetage des séquences, qui incluent la plupart de systèmes état de l'art, en effectuant une comparaison systématique sur trois tâches (NER, segmentation de texte et POS). Grâce à la comparaison et à l'analyse empirique, nous obtenons plusieurs conclusions pratiques dans chacune des tâches.

Ensuite, nous essayons de construire un système capable d'apprendre trois tâches d'étiquetage séquentiel et d'améliorer la précision de chaque tâche. Nous proposons donc un réseau de mémoire partagée, Shared Cell Long-Short Term Memory network (SCLSTM), pour l'étiquetage multi-tâche de séquences et comparons notre modèle avec deux modèles partagés d'étiquetage.

En détenant un ensemble de paramètres partagés, l'état de la cellule de notre modèle SCLSTM peut être supervisé à partir de trois tâches, tandis qu' il comporte un composant indépendant spécifique à la tâche pour apprendre les informations privées de chaque tâche.

Les résultats expérimentaux sur trois ensembles de données d'étiquetage de séquence de référence montrent l'efficacité de notre modèle SCLSTM pour les tâches de NER, de segmentation de texte (*text chunking*) et l'étiquetage morphosyntaxique (*POS tagging*).

Summary

Neural-based models have attracted considerable attention for automatic sequence labeling. Compared to the traditional models, neural-based models achieve better performance with less or no hand-craft feature engineering. Due to the sensitivity of the experimental setting, it is always hard to reproduce and compare the effectiveness of different models in an identical condition. Moreover, even these models can be applied on different sequence labeling tasks, like Name Entity Recognition (NER), Text Chunking, and Part of Speech tagging (POS), previous works fail to give better performance under a multi-task learning (MTL) setting than when learning each task individually.

We study the main factors affecting the performance of neural sequence labeling systems, by re-implementing sequence labeling models based on different neural architectures, which include most of the state-of-the-art methods, and run a systematic model comparison on three benchmarks (NER, Chunking, and POS tagging). Through the empirical comparison and analysis, we get several practical conclusions in such sequence labeling tasks.

Then we attempt to build a system that can learn three sequential tagging tasks at the same time improve the accuracy of each task. We propose a Shared Cell Long-Short Term Memory network (SCLSTM) for multi-task sequence labeling and compare our model with two shared-encoder sequential tagging models.

By holding a set of shared parameters, the cell state of our SCLSTM can get supervision from three tasks, while our SCLSTM model has independent task-specific components to learn private information of each task. Experimental results on three benchmark sequence labeling datasets demonstrate the effectiveness of our SCLSTM for NER, text chunking, and POS tagging tasks.

Contents

List of Tables

List of Figures

List of Abbreviations

NLP	*Natural Language Processing*
NER	*Name Entity Recognition*
POS	*Part of Speech*
MTL	*Multi-Task Learning*
RNN	*Recurrent Neural Network*
CNN	*Convolutional Neural Network*
LSTM	*Long Short-Term Memory*
Bi-LSTM	*Bi-directional Long Short-Term Memory*
MEMM	*Maximum Entropy Markov Model*
CRF	*Conditional Random Fields*
SC-LSTM	*Shared-Cell Long Short-Term Memory*
SGD	*Stochastic Gradient Descent*

Acknowledgements

First, my greatest thanks go to my advisor Philippe Langlais. He is a kind, caring and supportive advisor. He always behaves patiently and gives insightful views about my research, which helped me get results of better quality. I am also grateful to the members of my committee for their patience and support in overcoming numerous obstacles I have been facing through my research.

I would like to thank my friends for accepting nothing less than excellence from me. Last but not least, I would like to thank my family: my parents, my uncle and aunt for supporting me spiritually throughout writing this thesis and in my life in general.

INTRODUCTION

Motivation

Sequence labeling tasks, such as named entity recognition (NER), text chunking and part of speech tagging (POS) are all fundamental natural language processing (NLP) tasks that have attracted much attention for years. Sequence labeling is one of the first stages in language understanding for text mining problems, such as question answering, summarization, knowledge graph construction, and information retrieval. Its importance has been well recognized in the NLP community. In addition to the traditional sequence labeling models, e.g. Maximum Entropy Markov Model (MEMM) [37] and Conditional Random Fields (CRF) [27], in recent years, neural network based models have achieved impressive results in sequence labeling tasks with less or no hand-crafted features. With advances in deep learning, neural-based models have achieved many state-of-the-art results on different sequence labeling tasks.

On the other hand, Convolution Neural Network (CNN) has also been used to achieve the same purpose with the advantage of being efficient in parallel computation. Despite the impressive results in this research literature, reproducing reported results of neural models sometimes can be difficult, even if the codes are open source, and the datasets are under public licenses. The comparison among various neural-based models is challenging, as it is hard to deal with the sensitivity in different experimental settings. In order to conduct a systematic comparison between different models, it is advisable to re-implement these models and compare the performance under identical experiment setting.

Besides, most of the models in previous studies are learned separately based on single-task supervised objectives, which ignores the relation in different tasks of sequence labeling [22, 54, 23, 35]. The strongly related semantics information would benefit each other tasks of sequence labeling by being trained in a multi-task scenario [56, 20]. For example, the basic technique used for entity recognition is chunking, which segments and labels multi-token sequences, and the components of multi-token sequences in chunking are related to word-level tokens learned from part of speech tagging. Such mutual influence in different tasks of sequence labeling would benefit each other if being utilized in a unified joint learning model. In many such studies, though, neural network models are trained toward a single task in a supervised way by making use of relatively small annotated training material. Jointly learning multiple tasks can reduce the risk of over-fitting to one task, and many attempts have been made at doing so for sequence labeling tasks [7, 11, 12]. Results so far are not conclusive.

Many works have focussed on jointly learning two tasks, often with one being considered as the main task, the other being the auxiliary one [56, 6, 3]. For instance, chunking, combinatory categorical grammar supertagging, NER, super senses (SemCor), or multiword expression + supersense will be taken as the main task, while POS is the auxiliary task in [56]. Exceptions to this line of work include Collobert et al. [12] that evaluates four tasks: POS, chunking, NER and semantic role labeling; Kiperwasser and Ballesteros [26] consider a machine translation task with POS and dependency parsing; Niehues and Cho [40] consider machine translation with POS and NER tasks; Zhang and Weiss [67] show that jointly learning a POS tagger and a dependency parser is effective. Miwa and Bansal [38] jointly train models for entity detection and relation extraction in the field of relation extraction. Other works are trying to leverage language models to improve the performance of sequence labeling tasks. In [30], they propose a model which uses a neural language model to learn character-level knowledge, and conducts sequence labeling to guide the language model towards specific tasks. In works [45, 46, 14], they use neural language models pre-trained on a large unlabeled corpus to learn context-sensitive representations of words, and leverage this representation into the sequence labeling model.

More related to the present work are studies that analyze the effectiveness of different combinations of sequence labeling tasks in a multi-task learning. In particular, Changpinyo, Hu, and Sha [8] conduct an investigation on 11 sequence labeling tasks, while Alonso and Plank [3] evaluate 5 tasks but report signifiant gains for only one task.

Jointly learning multiple tasks in a way that benefit all of them simultaneously can increase the utility of multi-task learning (MTL). In order to do so, we propose a new LSTM cell which contains both shared parameters that can learn from all tasks, and task-specific parameters that can learn task specific information.

Contribution

Our contributions in this thesis are as follows:

- We re-implement 12 neural sequence labeling models and compare the performance under identical experiment settings. We also analyze several important factors by running experiments under different random seeds.
- We also propose a novel multi-task learning model: Shared Cell based LSTM (SC-LSTM), which can jointly learn the interplay information of different tasks by a shared cell in sequence labeling problems. Experimental results on three sequence labeling benchmark demonstrate the effectiveness of our SC-LSTM for the NER, text chunking, and POS tagging tasks. This work has been published in [33].

Organization

Chapter 1 provides an overview of three sequence labeling tasks, i.e. name entity recognition (NER), chunking and part of speech (POS) and introduce the basics of neural networks, including recurrent neural network (RNN) and its variant Long-Short Term Memory (LSTM), convolutional neural network (CNN), and several optimization algorithms etc. In the last part of this chapter, we will introduce briefly the multi-task learning methods.

Chapter 2 introduces methods that encode word-level sequence and character-level sequence and two inference methods. Then we present two kinds of shared-encoder models for Multi-task sequence labeling and analyze their advantages and disadvantages. Finally we introduce our method for tackling issues in these MTL approaches. We propose a shared cell LSTM, to leverage the supervision of different tasks into the cell state of LSTM. This method enables the model to learn shared and task-specific representations at the same time. Finally, we introduce the training algorithm and training objective used for multi-task learning.

Chapter 3 describes our experimental settings and results for analyzing effect of different components of neural tagger system in single task learning and gives a series of analysis of those results. We will investigate the different sequence encoding methods, and then evaluating the influence of different external factors, i.e. pre-trained embeddings, optimization methods and tagging schemes.

Chapter 4 provides experiments and discussions for multi-task learning methods. In this chapter, we will first compare our methods with two widely used shared-encoder methods. And then we will compare our methods with different results of current state-of-the-art models. Finally we will further do an analysis for the effect of our method by exploring the convergence and influence of various task groups.

Chapter 5 concludes the work briefly.

Chapter 1

BASICS

In this chapter, we first introduce the used three sequence labeling tasks, Named Entity Recognition (NER), Chunking, and Part-of-Speech tagging (POS) briefly, and give a short explanation of the evaluation metrics[1]. Then we introduce several basics of neural networks, including RNN, LSTM and CNN. We will also introduce some important aspects for training neural networks like optimization algorithms (i.e. SGD, Adam and Nadam) and regularization approaches (i.e. Dropout, Weight Decay). In the last part of this chapter, we will also introduce the basics of multi-task learning briefly.

1.1. Benchmark Tasks

The goal of sequence labeling is to assign tags to words, or more generally, to give discrete labels to discrete elements in a sequence. In this work, we focus on three sequence tagging tasks: Named Entity Recognition (NER), Chunking, and Part-of-Speech tagging (POS).

1.1.1. Named Entity Recognition (NER)

Named Entity Recognition (NER) is a subtask of information extraction that aims at labeling named entity mentions in unstructured texts, like person names, locations, organization names, time expressions, monetary values. As the first example shown in table 1.1, each token of the input sentence is labelled with its corresponding category. Since a named entity may consist of several tokens, the model should recover the spans of tokens, such as

[1]We will present how to compute typically considered evaluation metrics (F1-score, recall, precision and token accuracy) later in this section.

Task	Input/Output
NER	*Tunbridge Wells : Nottinghamshire 214 (P. Johnson 84 ; M. McCague 4-55) , Kent 108-3 .* *B-LOC I-LOC O B-ORG O O B-PER I-PER O O B-PER I-PER O O O B-ORG O O*
Chunking	*Those activities generated $ 26.1 million in operating profit last year .* *B-NP I-NP B-VP B-NP I-NP I-NP B-PP B-NP I-NP B-NP I-NP O*
POS	*The US troops fired into the hostile crowd , killing 4 .* *DET PROPN NOUN VERB ADP DET ADJ NOUN PUNCT VERB NUM* PUNCT

Table 1.1. Examples of three Tasks (NER, Chunking and POS)

Tunbridge Wells. We accomplish this by using a tagging scheme, for instance, a BIO tagging scheme. Each token at the beginning of a name span is labeld with a B-prefix; each token within a name span is labeled with an I-prefix. The entity type follows the prefix, e.g. *B-PER* for the start of a person name, and *I-LOC* for the inside of a location. Tokens not belonging to any parts of spans will be labeled as *O*. Using different tagging schemes has different influence on the accuracy of model. We will investigate this factor by comparing BIO notation and BIOES[2] [48] notation schemes in chapter 3.

In this work, we choose the CoNLL2003 benchmark[3] to evaluate the performance of our models on the NER task. We follow the CoNLL2003 setup [61], whose sentences are from the Reuters Corpus [29]. This dataset has been annotated for the CoNLL2003 shared task. It contains training, development and test data. In this dataset, named entities are categorized in the four categories (person names, locations, organizations and miscellaneous). The performance measurement adopts the macro F1 score which is the harmonic mean of the precision and recall for all named entities; see section 1.1.4 for details.

Current state-of-the-art performance under this setup is 93.09% F1-score [2]; Model of [14, 46] achieves 92.8% and 92.22%, respectively. All three works leverage different pre-trained contextualized representations into their models to boost performance. These pre-trained representations are contextualized by their surrounding text, meaning that the same word can

[2]The BIOES tag scheme has S tag for number of token span equels to 1 and E for the last token in a multiple token span. For instance, in the sentence: *Tom likes shopping*, Tom will be labeled as *S-Person* instead of *B-Person*; in sentence: *Michael Joseph Jackson was an American singer*, Jackson will be labeled as *E-Person* instead of *I-Person*

[3]The statistic details of dataset will be present in chapter 3.

have different embeddings depending on its context. These powerful pre-training embeddings have shown powerful ability in various natural language processing tasks. The model of [10] achieves 92.61% by proposing a Cross-View training method. Their model designs extra auxiliary prediction modules having a restricted view of input to match the predictions of the full model seeing the whole input. By joint training a language model, Peters et al. [45], Liu et al. [30] achieves 91.93% and 91.24% respectively. Ghaddar and Langlais [15] achieves 91.73% by designing robust lexical features for NER.

1.1.2. Chunking

Chunking seeks to tag segments of a sentence with syntactic constituents like noun or verbal phrases. As in the NER task, each word is assigned a single tag, so a tagging scheme is also used to denote the beginning and the end of phrasal segments. We evaluate our model using the CoNLL2000 shared task3 [60]. In this setup, sections 15-18 of the Wall Street Journal are taken as training set, section 20 as the testing set. We randomly choose 1000 sentences from training set as development set, this follows setups in [30, 45]. The performance measurement is computed using the F1-score as well.

The current state-of-the art performance under this setup is 96.72% F1-score [2]. By introducing a joint many-task model and designing a strategy for successively growing depth of model to solve increasingly complex tasks, Hashimoto et al. [20] achieves 95.77%. By placing different tasks into different layers, Søgaard and Goldberg [56] achieves 95.57%. Suzuki and Isozaki [59] achieves 95.15% by proposing a semi-supervised discriminative model for handling large scale unlabeled data.

1.1.3. Part-of-Speech tagging (POS)

Part-of-Speech tagging seeks to tag each token with a label indicating its syntactic role in the sentence, i.e. noun, verb, adj. and so on. We choose universal dependency (UD) POS dataset 1.3 version to evaluate our model on POS task. Sentences in the UD dataset are selected from the English Web Treebank [5]. This dataset is annotated by the Universal Dependency Project [41]. For the POS task, as each token in the sentence has it own syntactic meaning, we do not need a tagging scheme for it.

There are not many reported results on UD dataset version 1.3. The best reported result is 95.67% in [6]. We also find some results on UD POS dataset version 1.2. But these works all make use of extra techniques to improve performance. The current state-of-the-art performance is 96.73% F1-score [64] by utilizing adversarial training. By introducing an auxiliary loss function which accounts for rare words into a Bi-LSTM model, this method achieves 96.40% [47]. By joint learning POS Tagging and graph-based dependency parsing, this model achieves 95.55% [39].

Most works report their results on CoNLL2003 NER [61] and PTB POS [36] datasets [12, 23, 35], while others give their results on the NER dataset only [9, 28, 58]. Besides, most works use development set for selecting hyper-parameters [28, 35], while some papers combine training set and development set for training [9, 45]. There are other setups like [23], where the authers use a different data split for POS dataset. In other words, these results are somehow difficult to compare, and it is necessary to reproduce their models and compare corresponding results in an unified setting, which we will do in the chapter 3.

1.1.4. Evaluation Metrics

As a multi-class classification problem, the F1-score is used as the evaluation metrics for NER and chunking; Token accuracy is used to evaluate the performance of POS taggers. For NER and Chunking tasks, we first compute the percentage of detected entities or phrases that are correct (precision); and the percentage of entities or phrases in the data that were found by the model (recall); then the F1-score based on these two values. For POS, it contains no "O" label, all kinds of label are taken into account, so we use token accuracy for its evaluation. The precision, recall, F1-score and accuracy are calculated as follow:

$$\begin{aligned}
precision &= \frac{TP}{TP+FP}, \\
recall &= \frac{TP}{TP+FN}, \\
F1-score &= 2*\frac{precision*recall}{precision+recall}, \\
accuracy &= \frac{TP+TN}{TP+TN+FP+FN}
\end{aligned} \tag{1.1.1}$$

where the term true positives (TP) is the number of examples being correctly labeled except for "O" label; false positives (FP) is the number of detected examples being incorrectly labeled; false negative (FN) is the number of data with gold label being incorrectly labeled; True negatives (TN) is numbers of data with "O" label being correctly labeled.

We give a simplified example for calculating precision and recall for a multi-classification problem with labels A, B, C and O, where A, B and C are labels our models aim to detect while O stands for the out of span label we do not count. First, we generate a confusion matrix shown in Table 1.2. Second, we compute each value based on it.

	GoldLabel A	GoldLabel B	GoldLabel C	GoldLabel O	TotalPredicted #
Predicted A	$\underline{30}$	20	10	30	90
Predicted B	40	$\underline{20}$	10	10	80
Predicted C	20	10	$\underline{60}$	20	110
Predicted O	10	50	20	40	120
TotalGold	100	100	100	100	

Table 1.2. This is an example confusion matrix for 4 labels: A,B,C and O.

For this example, the TP is the sum of values of predicted A, B and C in the diagonal ($30 + 20 + 60 = 110$); The FP is equal to the sum of the rest of prediced values of A, B and C ($60 + 60 + 50 = 170$); The FN is equal to the sum of the rest of GoldLabel values of A, B and C ($70 + 80 + 40 = 190$). So the recall is $\frac{110}{110+1900} \approx 0.37$ and precision is $\frac{110}{110+170} \approx 0.39$.

1.2. Neural Network Basics

In this section, we will introduce all basic components of neural network based sequence labeling models, including recurrent neural network (RNN) and its variations (Long short-term memory (LSTM) and bidirectional LSTM), as well as convolutional neural networks (CNN). RNN and CNN are commonly used in neural-based sequence tagging models, both of them serve the same purpose: transforming a sequence of continuous vector representation $x_1, ..., x_n \in R^d$, into a sequence of latent vector representation $s_1, ..., s_n \in R^h$ containing sequence dependency information. Besides, we will describe several optimization and regularization methods typically used in the training of such networks. The last part of this section introduces the two kinds of inference approaches for sequence labeling, the softmax classifier and conditional random field classifier (CRF) .

1.2.1. Fully Connected Neural Networks

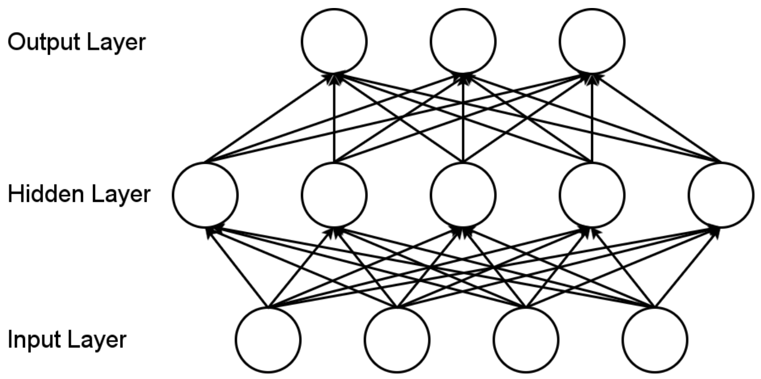


Figure 1.1. Illustration of a one hidden layer fully connected neural network.

Neural Networks receive an input (a single vector) and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer are entirely independent and do not share any connections. The last fully-connected layer is called the "output layer", and in classification settings it represents the class scores. Formally, a one hidden-layer can be represented as a function $f : R^D \Rightarrow R^L$, where D is the dimension of input vector x and L is the dimension of output vector $f(x)$. The hidden representation $h(x) \in R^H$ is computed as:

$$\mathbf{h}(\mathbf{x}) = \sigma(\mathbf{W^{(1)}}\mathbf{x} + \mathbf{b^{(1)}}) \tag{1.2.1}$$

where $\mathbf{W^{(1)}} \in R^{H \times D}$ is the weight matrix, $\mathbf{b^{(1)}} \in R^H$ is the bias vector, σ is a logistic sigmoid function:

$$sigmoid(x) = \frac{1}{1 + exp(-x)} \tag{1.2.2}$$

which performs non-linear transformation to the hidden state. There are many kinds of non-linear functions, typical choices including tangent function :

$$tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{1.2.3}$$

or the Rectified Linear Unit (ReLU) function [17] :

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x <= 0 \end{cases} \quad (1.2.4)$$

.

1.2.2. Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data [18]. RNNs take a sequence of inputs and compute a hidden state vector based on the current input and the previous hidden state at each time step. The hidden representation $\mathbf{h_t} \in R^H$ is computed recursively as the following equation:

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t) \quad (1.2.5)$$

where $\mathbf{x_t} \in R^D$ is the vectorized representation of input at time step t, $\mathbf{h}_t$ is the hidden state at time step t, $\mathbf{W} \in R^{H \times D}$ and $\mathbf{U} \in R^{H \times D}$ are weight matrices, and σ is a nonlinear activation function such as ReLU or tanh. For sequence labeling task, neural models take the hidden state as the representation of current input token. Vanilla RNN easily suffers gradient vanishing problem. To solve this issue, we can use gated recurrent neural networks. In particular, Long Short-Term Memory networks [22] and its variant bidirectional Long Short-Term Memory (Bi-LSTM) [54] are widely used in sequence labeling tasks [12, 23, 35, 55, 30, 68, 31]. Here, we mainly revisit the structure of Bi-LSTM.

Each unit in LSTM is made up of four functional gate units: a forget gate controlling what information to remove from the candidate cell of the last time step, an input gate controlling what information to add to the current candidate cell, an output gate controlling what information to release from current candidate cell, and a candidate gate to calculate the current candidate state. All computation are calculated as follows:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t), \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t), \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t), \\
\hat{\mathbf{c}} &= \tanh(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t), \\
\mathbf{c}_t &= \mathbf{i}_t * \hat{\mathbf{c}} + \mathbf{f}_t * \mathbf{c}_{t-1}, \\
\mathbf{h}_t &= \mathbf{o}_t * \tanh(\mathbf{c}_t),
\end{aligned} \tag{1.2.6}$$

where $\mathbf{x}_t$ is the input vector[4] at time step t and , $\mathbf{h}_{t-1}$ is the hidden vector at time step $t-1$, and $\mathbf{h}_{t-1}$ is the output vector of time step $t-1$. $\mathbf{f}_t$ is the forget gate, $\mathbf{i}_t$ is the input gate, $\mathbf{o}_t$ is the output gate, σ is the sigmoid function, $\hat{\mathbf{c}}$ is the new candidate state, $\mathbf{c}_t$ is the updated candidate cell, which encodes information of current input and history information, and $*$ indicates the element-wise product. $\mathbf{W}_f, \mathbf{U}_f, \mathbf{W}_i, \mathbf{U}_i, \mathbf{W}_o, \mathbf{U}_o, \mathbf{W}_c, \mathbf{U}_c$ are weight matrices.

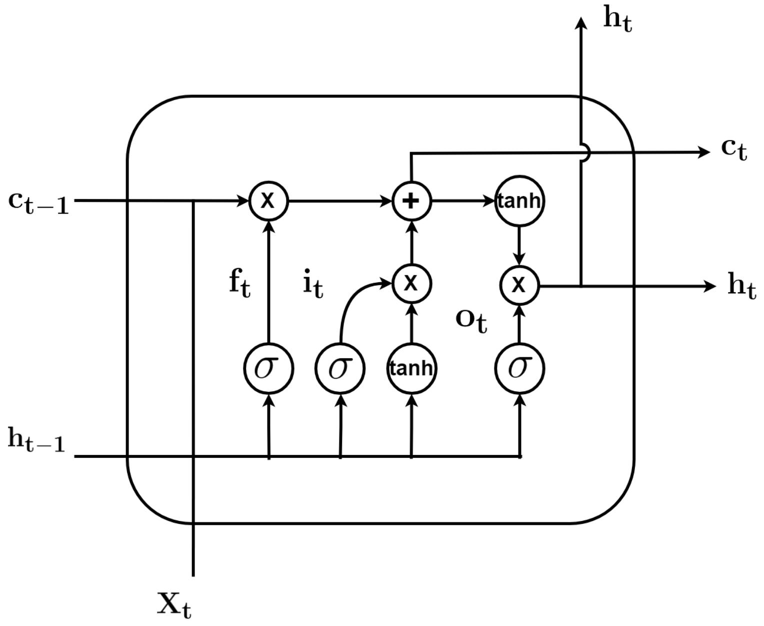


Figure 1.2. The structure of a LSTM.

Bi-LSTM is a combination of two LSTM with their own parameters: one encodes the input sequence in the forward direction, the other encodes the input sequence in the backward

[4]The input vector is from a look-up table that maps raw texts to vector representations like word embeddings

direction. It returns two hidden vectors $\overleftarrow{\mathbf{h}}_t$ and $\overrightarrow{\mathbf{h}}_t$ at time step t. We represent the hidden states $\mathbf{h}_t$ of Bi-LSTM by making a concatenation of two hidden states, which captures both previous and future information in the sequence.

$$\mathbf{h}_t = \overleftarrow{\mathbf{h}}_t \oplus \overrightarrow{\mathbf{h}}_t, \tag{1.2.7}$$

where $\oplus$ is the concatenation operation.

1.2.3. Convolutional Neural Networks (CNNs)

Convolutional networks (CNNs) are a family of neural networks dedicated to processing data with grid topology. For example, time series data, which can be considered as a one-dimensional grid sampled at regular intervals, as well as image data, can be considered as a two-dimensional pixel grid. In CNN, each filter f is replicated across the entire input field. These shared filters use the same set of parameters (weight matrix and bias). In this way, the shared filters allow local features to be identified regardless of their position in the input. Furthermore, making weight shared can increases the computation efficiency by reducing the number of independent parameters being learned [18]. In this work, we use CNN to process sequential data, i.e. word-level sequence and character-level sequence to get the hidden representation of each token.The CNN achieves this purpose by using f kernels and sliding them over the input sequence. Each kernel performs local convolution on the sub-sequences of the input to obtain a set of feature maps (scalars), then a global max-pooling-over-time is performed to obtain a scalar. These scalars from the f filters are then concatenated into the sequence representation vector [18]. This process is shown in figure 1.3.

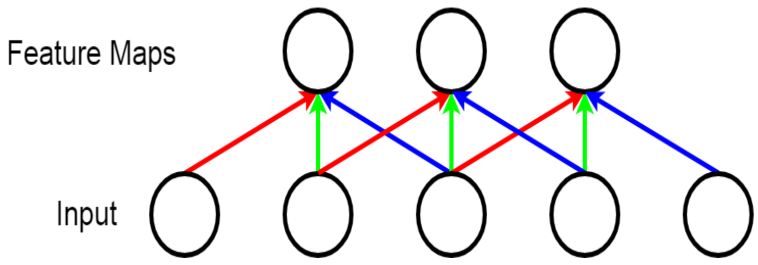


Figure 1.3. The illustration of convolution calculation. Weights of the same colour are identical. The computation can be performed in parallel by directly replicating weights and calculate across the sub-region of the entire input simultaneously.

We have described different neural-based architectures. Neural models always involve a huge amount of parameters, which make it is difficult to train these models efficiently. As this issue is so crucial and so expensive, a specialized set of optimization techniques have been developed for solving it. Next, we will present some optimization methods for neural network training.

1.2.4. Optimization for Training Neural Models

Typically, the goal of a machine learning algorithm is to find the optimal set of parameters of a model by reducing the expected generalization error given by the equation 1.2.8.

$$J^*(\theta) = E_{(x,y)\sim p_{data}} L(f(x;\theta), y) \tag{1.2.8}$$

where $L(f(x;\theta), y)$ is the loss function designed based on our tasks. $f(x;\theta)$ is the neural model parameterized by θ, p_{data} is the data distribution.

The most used optimization algorithm to estimate θ parameters which minimize a cost function $J(\theta)$ is the stochastic gradient descent algorithm. The SGD algorithm is an iterative optimization method that modifies the set of parameters until the cost function on the training data converges to one optimal solution [18]. At each time step, we move one step in the opposite direction towards the gradient of the cost function so that J moves to a local minimal. When a neural model contains huge amount of parameters, vanilla SGD algorithm tends to find local minimal points with bad quality and becomes time consuming. Since the rise of deep learning, several SGD variants have been developed to accelerate or improve the learning process. Our objective is not to make a full summary of these optimization algorithms, so we present only the one we used in our experiments to investigate its influence on the models. A thorough comparison of these learning methods is made in [51], using adaptive momentum based optimizors can accelerate the training greatly. Here, we introduce two widely used adaptive methods Adam and Nadam.

Adam: Adaptive Moment Estimation (Adam) [25] is a method that computes adaptive learning rates for each parameter. It stores an exponentially decaying average of previous squared gradients v_t, and it also keeps an exponentially decaying average of previous gradients m_t, which is similar to momentum. m_t and v_t estimate the first momentum and second momentum of the gradients respectively. The parameters of models are updated as follows:

$$\begin{aligned} g_t &= \bigtriangledown J(\theta_{t-1}), \qquad (1.2.9)\\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t,\\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2,\\ \hat{m_t} &= \frac{m_t}{1-\beta_1^t},\\ \hat{v_t} &= \frac{v_t}{1-\beta_2^t},\\ \alpha_t &= \alpha\frac{\sqrt{(1-\beta_2^t)}}{1-\beta_1^t},\\ \theta_t &= \theta_{t-1} - \alpha_t \frac{\hat{m}_t}{\sqrt[*]{\hat{v}_t + \epsilon}}, \end{aligned}$$

where α is the initial learning rate, β_1 and β_2 are hyper-parameters that control the exponential decaying rate of m_t and v_t. The authors recommend initializing hyper-parameters to the following values; $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and ϵ= 10e-8.

Nadam: Nesterov-accelerated Adaptive Moment Estimation (Nadam) incorporates Nesterov accelerated gradient (NAG) into Adam optimizer. NAG aims to update parameters with momentum step before computing the gradient. In practice, using NAG in gradient calculation only needs to modify the momentum m_{t-1} of time $t-1$ to the momentum m_t of time t.

The parameters of a model will be updated as follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}) \qquad (1.2.10)$$

The neural models are powerful to memorize the whole training data [66], leading to over-fitting on training set, which is not what we want. In order to make the neural model to generalize better, there are many regularizing techniques to help avoiding over-fitting. Next, we will introduce three commonly used regularization approaches.

1.2.5. Regularization for Training Neural Models

In machine learning settings, regularization can prevent the model from over-fitting the training data by adding some penalty.

Dropout: Dropout [57] is one of the powerful regularizing methods. Practically, during the training process of neural models, individual units of the model will be either dropped out of the system with probability $1-p$ or activated with probability p, while during testing, the net will activate all units, but reduces the weight of them by a factor p to offset the missing activations during training.

Weight-decay: One popular regularization method in deep learning is to include a weight decay term during updating. In practice, the weight decay is equal to add an L_2 regularization term to the loss function. Then the optimization target becomes:

$$w* = \arg min_w(L + \lambda\|w\|_2^2) \tag{1.2.11}$$

where the w stands for the parameters of the model, L is the original loss function (e.g. cross entropy), and λ is a hyper-parameter that controls how much to regularize.

Gradient Clipping: Gradient clipping is mostly used in recurrent neural networks. When training recurrent neural networks with the full gradient, the derivatives will become excessively large as the input time steps increase. This problem known as "Exploding Gradients" can cause unstable training process.[19] To solve such issue, we utilize a technique called "gradient clipping" that rescales the loss derivative before propagating it backward through the network. This method uses normalized gradients to update the parameters of model to prevent underflow or overflow weights.[18]

We also explore to boost the performance of the neural models on sequence labeling tasks by using Multi-task learning. Here, we introduce some basics of multi-task learning.

1.3. Multi-Task Learning

Multi-Task Learning (MTL) has been used successfully across many applications of machine learning field, from computer vision [16] and speech recognition [13] to natural language processing [11]. MTL can get implicit data augmentation which effectively increases the size of training samples for our model. Besides, some useful features K might be hard to learn in task A, while easy to learn from task B. Thereby joint learning task A and task B can allow the model to learn these kind of features. Furthermore, MTL make the model to learn representations that other tasks also prefer [51]. This helps the model to generalize better to new tasks in the future. Finally, MTL can be considered as a way of regularization as it reduces the risk of over-fitting to a single task during training.

In a general MTL setup, the model will be trained jointly on multiple tasks. The purpose is to achieve a better generalization across different tasks on unseen data. In the next chapter, this work will introduce two popular MTL frameworks for sequence labeling tasks. Motivated by the success of multi-task learning [7, 11, 12, 32], in which multiple related tasks can simultaneously improve the performance of a single task. We design a novel unit with a shared cell in Bi-LSTM to leverage the mutual beneficial information in other tasks for the target task prediction, and meanwhile, utilize the task-specific cell to maintain the specific information in the target task.

Chapter 2

MODELS

In this chapter, we first present the neural tagger system and how to implement its components in detail. We introduce and compare two methods for encoding word-level inputs, two methods for encoding character-level inputs and two inference methods. Next, we introduce two shared-encoder frameworks for sequence learning tasks with multi-task learning, and analyze the advantage and disadvantage of these two approaches. Then we propose a SC-LSTM for MTL and SC-LSTM based system for sequence labeling.

2.1. Neural Tagger Systems

A neural tagger system is a neural network based model which takes a sentence as input and outputs the corresponding target label of each token in the sentence. Neural tagger systems consist of three components, a look-up table which maps raw text input to its corresponding vector representation (i.e. word embeddings), an encoder which converts the input word to a hidden representation and an inference layer (Softmax or CRF) that labels the current input based on the hidden representation.

There are many works on the encoder part [23, 35, 28, 12, 6, 58]. First, the most popular method is using LSTM to extract word-level information from the input sequence, as recurrent networks can handle sequential dependences. The bidirectional LSTM (Bi-LSTM) can encode forward and backward dependences in the sequence. Leveraging character-level information into the input often improves the richness of hidden representations. By considering words as a character sequences, like *Panda* as *P*, *a*, *n*, *d*, *a*. Utilizing RNN or CNN to extract char-level features is helpful for the accuracy. Third, most of the recent works found that contextualized features can significantly boost many supervised tasks, so there are many attempts to combine language models into sequence labeling problems. Finally, there are several works exploring transfer learning or multi-task learning methods to improve the performance, but some of them fail to improve all used tasks.

2.1.1. Word-level Representations

We can model word sequences through LSTM or CNN structures. Both of two methods are widely used to encode sequential information: Lample et al. [28], Ma and Hovy [35], Chiu and Nichols [9] and Huang et al. [23] make use of the LSTM cells to encode the word-level information and achieve good performance. Others' work [12, 58, 53, 52] choose CNN as the word-level encoder due to the fact that convolution calculations can be computed on the input sequence in parallel.

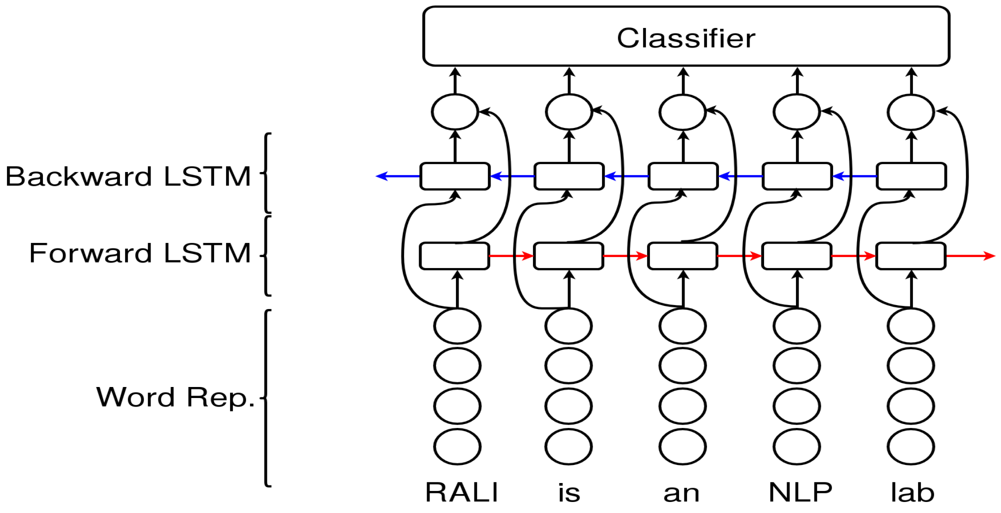
Word RNN As figure 2.1(a) shows word representations are fed into a forward and backward LSTM respectively, and then the hidden states of the forward and backward LSTMs are concatenated at each word to capture the global information of the whole sequence.

Word CNN As figure 2.1(b) shows the structure of word-level CNN extractor. For each convolution layer, a kernel of size three slides along the sequence of vector representations to extract local features on the input words and apply a ReLU function. In our experiment, we use a four-layer CNN as [58].

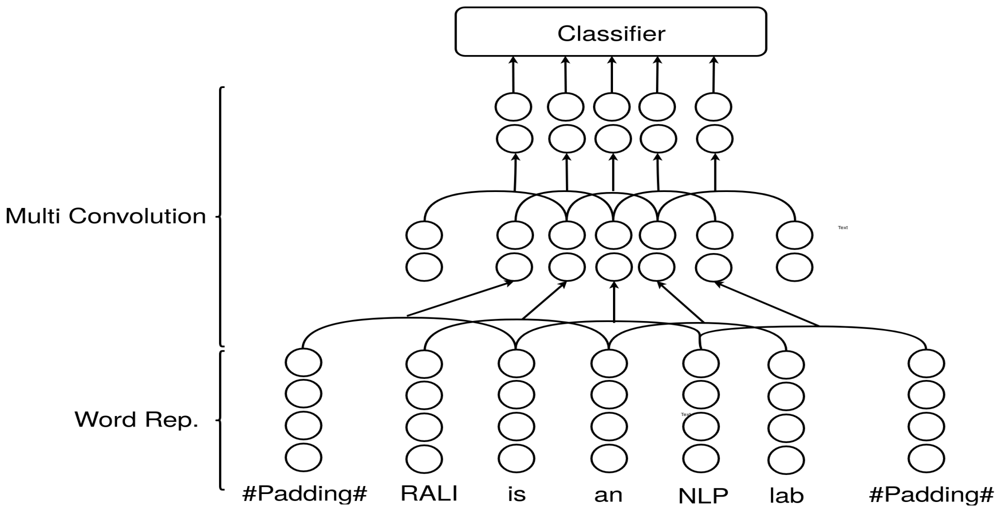
2.1.2. Character-level Representations

As many natural language processing tasks, rare and unknown words can influence the final performance: if a word is not in the training data, then we can not find obvious choice for its word embedding. Besides, character-level features like prefix, suffix and capitalization play an important role in sequence tagging task. These morphological properties were considered in feature-based models. In neural-based models, these properties can be represented by embedding methods through a human-defined look-up table, or by using neural networks without hand-craft features. In this work, we concentrate on two neural character sequence methods without hand-engineered information. There were many works utilizing RNN or CNN to encoding character-level sequential information. In our experiments, we choose the same structures as [28] and [35].

Character RNN: As figure 2.2(a) shows, to encode the whole sequential information of a word "PANDA", we make use of a bi-directional LSTM on the character sequence of each



(a) Word LSTM model

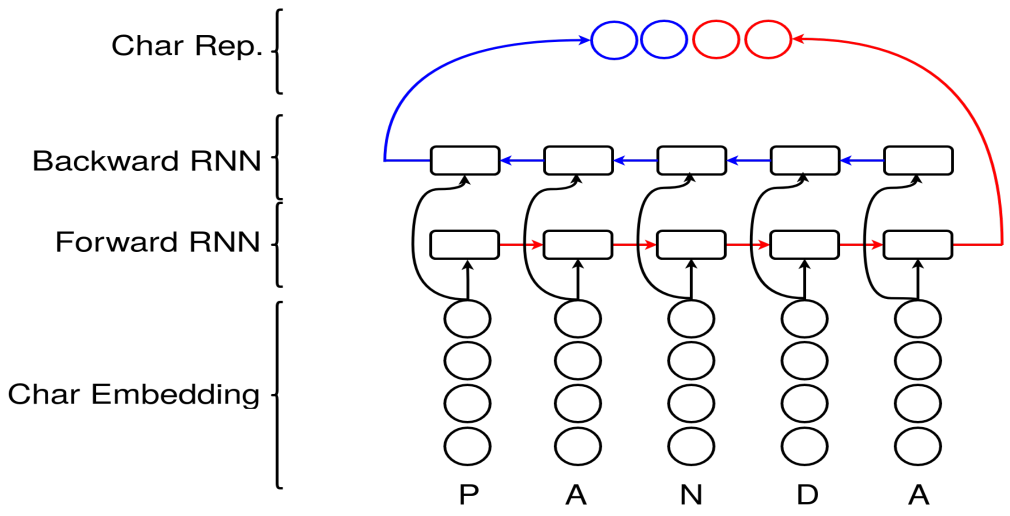


(b) Word CNN model

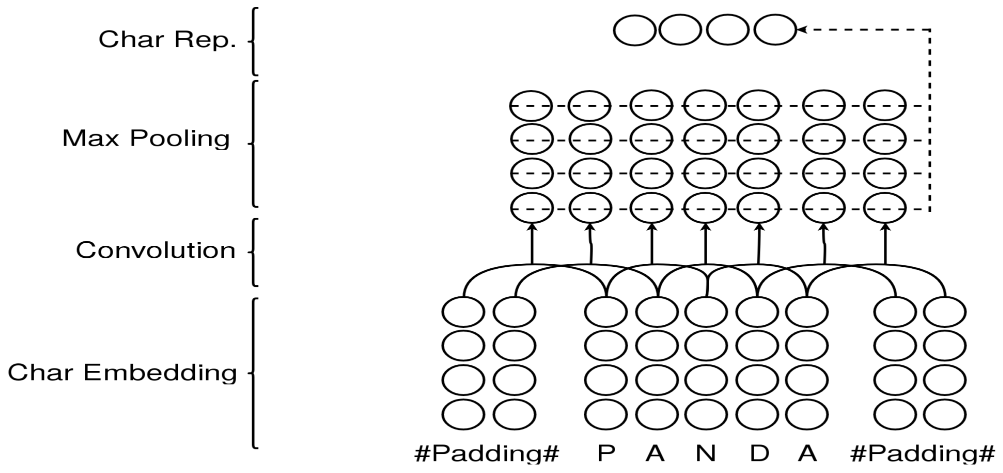
Figure 2.1. Word level representation models. The input of both models can be a concatenation of word embedding , character-level representation and contextualized representation of the token. The classifier of both models can be a softmax layer or a CRF layer

word, then concatenating the forward final hidden state $\overrightarrow{\mathbf{h}}_t$ and the backward final hidden state $\overleftarrow{\mathbf{h}}_t$ as the final representation of character sequence. This method takes all characters of the token into account and is position sensitive. Therefore, it can distinguish the different characters at the beginning, in the middle, or at the end of a token.

Character CNN: Figure 2.2(b) shows, in order to encode the word "PANDA", we utilize one layer CNN to extract local features and then feed these features to a max-pooling



(a) Char RNN model



(b) Char CNN model

Figure 2.2. Character level representation models. The input of both models are a series of character embeddings of the tokens. The dashed line in (b) means maxpooling operation.

layer to capture the character-level representations. To create a vector representation with suitable dimension, we need to add padding tokens at the beginning and end of the character sequence before feeding the input to the embedding layer. In our experiment, we choose to use one layer CNN with kernel size of three[1]. This approach takes trigrams into account and is position independent. Therefore, it will not be able to tell the difference between trigrams at the start and at the end of the word.

[1]This is achieved by doing a grid search from 1 to 5 to find the best filter size

So the learned character-level representation and the word embedding are concatenated and are fed to word-level encoder which outputs the word-level representation. The learned word-level representation will be used as the input for an inference layer.

2.1.3. Inference Layers

In this work, we evaluate two kinds of inference layers for sequence labeling tasks. The softmax function has widely been used in multi-class classification problems, it takes a vector of arbitrary real-valued scores $\mathbf{z} \in R^N$ and squashes it to a vector of values between zero and one which sums to one. It is computed as follow:

$$softmax(\mathbf{z}) = \frac{exp(z_j)}{\sum_{i=1}^{N} exp(z_i)},\ for\ j = 1, ..., N \tag{2.1.1}$$

By using the softmax inference layer, the loss function will be cross-entropy. In sequence tagging, tags can depend on their neighbours, for instance in POS, a verb is likely to appear after a noun, while not appear after an article. CRF (conditional random field) can consider the dependence of tags, hence it is commonly utilized in sequence labeling tasks. By using CRF inference layer, we will optimize the conditional probability:

$$p(y_1, ..., y_m | x_1, ..., x_m) = \frac{exp(\sum_{j=1}^{m} w \cdot \phi(x_1, ..., x_m, s_{j-1}, s_j, j)}{\sum_{s_1 ... s_m} exp(\sum_{j=1}^{m} w \cdot \phi(x_1, ..., x_m, s_{j-1}, s_j, j)} \tag{2.1.2}$$

where, y_j is the tag, x_j is the input token and s_j is the possible state. $\phi(x, s)$ is a potential function that can be defined in various ways.

There are many neural tagger systems based on above components. These previous works shown in Table 2.1 can be classified based on different combinations of three components : (a) word-level inputs encoder; (b) character-level inputs encoder; (c) inference layer.

2.2. Multi-task Learning for Sequence Labeling

In a multi-task learning (MTL) setup, the model is trained jointly for more than a single task. Multi-task learning for neural networks has a long history[7]. The goal of MTL is to help model learn parameters or representations that generalize well over various tasks and to achieve better generalization on unseen data. In this work, we concentrate on shared encoder methods for multi-task learning.

	No Char	**Char-LSTM**	**Char-CNN**
Word-LSTM	[35] [65] [47]	[28]	[35]
Word-CNN	[58]	-	-
Word-LSTM + CRF	[23]	[28] [49] [64]	[35] [9] [45]
Word-CNN + CRF	[12]	-	[6]

Table 2.1. Neural Tagger Systems, models without CRF are using Softmax classifiers

In this section, we will introduce two popular shared-encoder methods for sequence labeling with Multi-task Learning, and analyze the advantage and disadvantage of these two approaches, then propose a novel LSTM unit, called Shared Cell LSTM (SC-LSTM) and SC-LSTM based system for sequence labeling. This SC-LSTM contains shared parameters that can learn from multiple tasks, and parameters that can learn from task-specific knowledge.

2.2.1. Two Shared-encoder Methods

There are two kinds of neural-based MTL methods. The first one — LSTM-s hereafter — uses an identical representation for all tasks. This is illustrated in Figure 2.3 (left), where the three-layer of LSTMs are being stacked.[2] This three layers LSTM can be considered as an encoder. We take the hidden state $\mathbf{h}_t^{(3)}$ of the outermost layer as the representation of the current input $\mathbf{x}_t$. This hidden state $\mathbf{h}_t^{(3)}$ is passed to different downstream task classifiers. Within the encoder, the first layer computes $\mathbf{h}_t^{(1)}$ based on current input $\mathbf{x}_t$ and the last time step hidden state $\mathbf{h}_{t-1}^{(1)}$ of the 1st layer. Then $\mathbf{h}_t^{(1)}$ will be taken as input for the second layer and history information for 1st layer at next time step. The 2nd and 3rd layers will do similar computations, respectively. As all tasks share the same hidden state $\mathbf{h}_t^{(3)}$, the error derivative will be passed through it to update all parameters of the encoder during the training process. While the different tasks directly interact with all parameters of the model, this increases the risk of optimization conflicts when gold-standard labels from different tasks have no significant correlation.

The second class of multi-task architectures is depicted in the middle part of Figure 2.3, and is named LSTM-d hereafter. In this configuration, each LSTM layer feeds a task-specific

[2]This typically delivers better performance than having just one. In practice also, LSTM layers are replaced by bi-LSTM ones.

classifier and serves as input to the next stacked LSTM layer [56]. The 1st layer outputs hidden state $\mathbf{h}_t^{(1)}$ of time step t, and $\mathbf{h}_t^{(1)}$ will be taken as the representation of input $\mathbf{x}_t$ for task classifier 1; it also serves as the input of 2nd layer and the history information of 1st layer at next time step. Similar computations will be done respectively on the 2nd and 3rd layer to give $\mathbf{h}_t^{(2)}$ and $\mathbf{h}_t^{(3)}$. The underlying assumption is that tasks may be ordered in such a way that easier tasks are learned first, the target tasks being the latest one considered, thus benefiting the hidden state of the lower layers. One drawback, however, is that one must decide which task to consider first, a decision which may impact the overall performance. Furthermore, using the hidden state of lower layers increases the limitation of learning representation for that task.

We believe that one reason for the lack of consistent benefits of MTL in the labelling literature is that the proposed models share all or part of the parameters for extracting hidden states, which leads to optimization conflicts when different tasks require task-specific features. We believe it will be helpful if we give the model ability to learn a task-specific representation [4, 42, 26] at the same time. This observation led us to design a new LSTM cell called shared cell LSTM (SC-LSTM)[3] which allows at almost no additional computation cost to efficiently train a single RNN-based model, where task-specific labelers clearly outperform their singly-tasked counterparts. Actually, by training our model on NER, chunking and POS tagging, we report state-of-the-art (or highly competitive) results on each task, without using external knowledge (such as gazetteers that has been shown to be important for NER), or hand-picking tasks to combine. Our SC-LSTM based system can be understood as replace original LSTM cell at each layer with our SC-LSTM. At the outermost layer, SC-LSTM outputs three representation $\mathbf{s}_t^1$, $\mathbf{s}_t^2$ and $\mathbf{s}_t^3$ for three tasks respectively. Each of them is a combination of task-invariant and task-specific features learned by neural networks. We will present the details of SC-LSTM in the next section.

2.2.2. Shared Cell based LSTM (SC-LSTM)

The overall structure of our cell is depicted in Figure 2.4. On top of a standard LSTM cell, we add one cell per task with its own parameters. The standard LSTM cell is thus shared among the K task-specific cells, therefore the name we choose for this new cell, which

[3]This part concerned with SC-LSTM has been published as a conference paper at 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics.

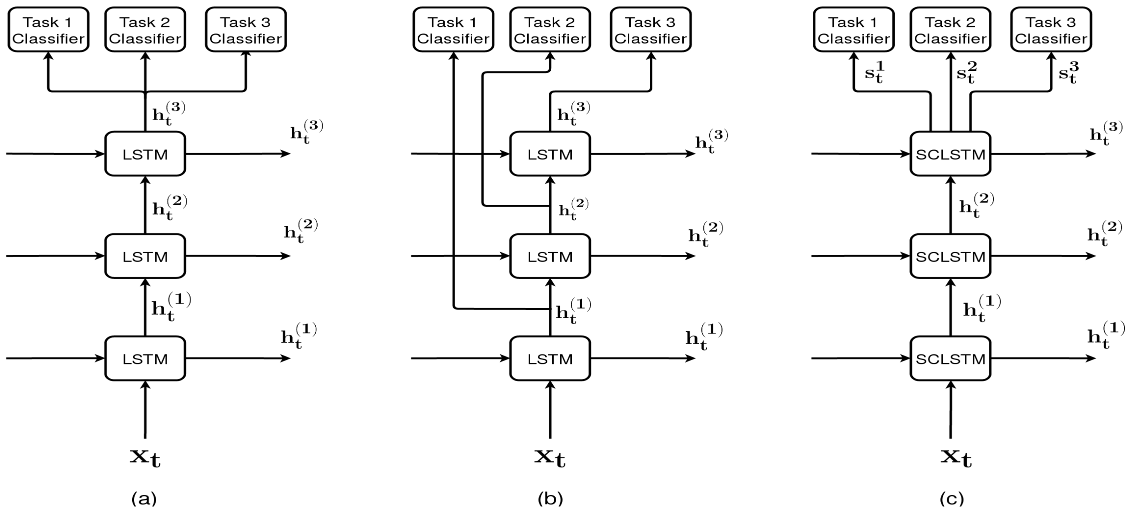


Figure 2.3. Overview of three shared encoder MTL tagging system: (a) LSTM-s (b) LSTM-d (c) our SC-LSTM based system.

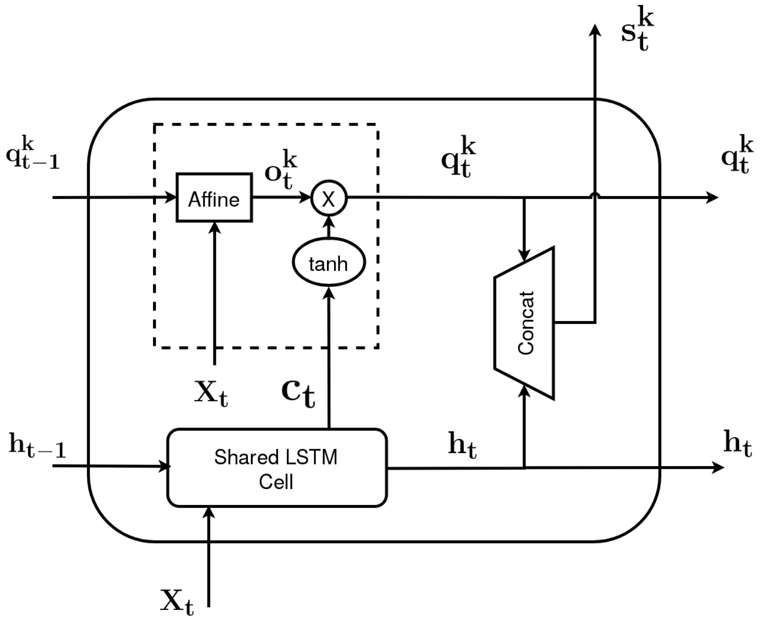


Figure 2.4. Structure of an SC-LSTM cell. The dashed delimited box depicts the task-specific cell. For clarity reasons, we only show one such cell, while in practice there is one task-specific cell for each task.

stands for Shared-Cell LSTM [33].

The shared LSTM cell outputs a cell state $\mathbf{c}_t$ and a hidden state $\mathbf{h}_t$ taken as task-invariant representation. This is because the parameters of the Shared LSTM cell can be updated by the gradients of all tasks involved, its output gate will select global information from the internal memory $\mathbf{c}_t$. Next, we calculate the task-specific representation $\mathbf{q}_t^k$ for the task k. First, we run a sigmoid layer which decides what parts of the cell state to output based on current input $\mathbf{x}_t$ and task-specific hidden representation $\mathbf{q}_{t-1}^k$ of last time step. The $\mathbf{q}_{t-1}^k$ contains the history information for task k only, thereby the sigmoid layer can be considered as a filter to select useful information only for task k. Then, we pass the cell state through *tanh* function (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid layer, so that the task-specific part (dashed box in Figure 2.4) outputs the parts $\mathbf{q}_t^k$ only related with task k. Task-specific cells are each parametrized by an output gate $\mathbf{o}_t^k$ which learns to select the useful information from the shared internal memory $\mathbf{c}_t$ and outputs $\mathbf{q}_t^k$. This is formally described in Equation 2.2.1, where $\mathbf{W}_k$ and $\mathbf{U}_k$ are two extra weight matrices that parametrize the kth task, and $\mathbf{q}_t^k$ has to be understood as a task-specific hidden representation since parameters of kth task-specific cell are only updated by supervision from task k.

$$\begin{aligned} \mathbf{o}_t^k &= \sigma(\mathbf{W}_k \mathbf{q}_{t-1}^k + \mathbf{U}_k \mathbf{x}_t^k) \\ \mathbf{q}_t^k &= \mathbf{o}_t^k * \tanh(\mathbf{c}_t) \end{aligned} \tag{2.2.1}$$

In order to make use of both shared and task-specific information [24, 44, 21], for the kth task, we concatenate the output of the shared cell $\mathbf{h}_t$ and of the task-specific one $\mathbf{q}_t^k$ to generate the final latent representation, as noted in Equation 2.2.2, where $\oplus$ is the concatenation operation. In practice, we stack SC-LSTM layers. The top-most layer uses $\mathbf{s}_t^k$ as a representation of the current input, while cells in lower layers pass the current shared hidden state $\mathbf{h}_t$ to the upper SC-LSTM cell.

$$\mathbf{s}_t^k = \mathbf{q}_t^k \oplus \mathbf{h}_t \tag{2.2.2}$$

The parameters of the shared cells will be updated by the loss of different tasks, respectively. So the output of it will preserve useful information across multiple tasks. The final representation of current input contains shared and private features. During the training process, the SC-LSTM preserves a parametrized output gate for each task that learn private features for that task. This task can update parameters of the output gate to filter task-specific information. In the multi-task learning setup, the model is trained jointly for three tasks in a random manner. In the following part, we first present the training objective function and then describe the training procedure.

2.2.3. Training objective and procedure

In MTL setting, we concentrate on two or more tasks and train a model jointly over a combined loss of each task.

$$\epsilon = \sum_{k=1}^{K} \lambda^k L(\mathbf{y}^k, \hat{\mathbf{y}}^k) \tag{2.2.3}$$

The objective function ϵ we minimize is a linear combination of task-specific loss functions, where the weighting coefficients (λ^k in Equation 2.2.3) are hyper-parameters. As the numbers of training data of various tasks are different, we need λ^k to balance the weight of loss in the objective function; K is the number of tasks and $L(\mathbf{y}^k, \hat{\mathbf{y}}^k)$ is the loss of task k.

We seek to minimize cross-entropy of the predicted and true distributions, therefore task-specific loss functions are defined according to Equation 2.2.4.

$$L(\mathbf{y}^k, \hat{\mathbf{y}}^k) = -\sum_{i=1}^{n_k} \sum_{j=1}^{m_i} \mathbf{y}_{i,j}^k \ \log \hat{\mathbf{y}}_{i,j}^k \tag{2.2.4}$$

where $\hat{\mathbf{y}}_{i,j}^k \in R^{T_k}$ is the predicted vector of the kth softmax classifier parametrized by a projection matrix $\mathbf{W}^k \in R^{h \times T_k}$ and a bias vector $\mathbf{b}^k \in R^{T_k}$; T_k is the number of tags for task k and h is the dimension of hidden state $\mathbf{s}_{i,j}^k$.

$$\hat{\mathbf{y}}_{i,j}^k = softmax(\mathbf{W}^k \mathbf{s}_{\mathbf{i,j}}^{\mathbf{k}} + \mathbf{b}^k) \tag{2.2.5}$$

where the hidden state $\mathbf{s}_{i,j}^k$ encodes the jth token of the ith sequence for task k. In an SC-LSTM cell with k tasks, we will add k matrices and k bias vectors, compared with a vanilla LSTM cell, which increases the capacity of thew resulting model. This extra calculation is conducted in parallel with the original LSTM computations. The softmax function is a normalized exponential function:

$$softmax(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \ for \ i = 1, ..., N \ and \ \mathbf{z} = (z_1,...,z_N) \in R^N \tag{2.2.6}$$

The training material available may be gathered from different datasets S_k which means that input sequences differ from one task to another. Therefore in practice we build K

dataloaders, and the training is achieved in a stochastic manner by looping over the tasks at each epoch, as detailed in Algorithm 1.

Algorithm 1 Stochastic training procedure

```
1: procedure TRAINING
2:     for each epoch do
3:         Randomly choose a task $k$.
4:         Randomly choose a training example not yet considered in the dataset $S_k$
5:         Update the parameters for this task by taking a gradient step based on this example.
6:         Go to step 3 until using all examples.
7:     end for
8: end procedure
```

Chapter 3

EXPERIMENTS WITH SINGLE-TASK LEARNING

We split our experiments into two parts. The first part concentrates on single-task learning, namely, all results in chapter3 are from models trained on a single task. The second part in chapter 4 will focus on multi-task learning. All experiments in these two chapters utilize the same datasets and identical setups. For convenience, we only present the detail of datasets in chapter 3.

3.1. Datasets

Dataset		**Train**	**Dev**	**Test**	**Tag #**
NER	Sent #	14,987	3,644	3,486	4
	Token #	205k	52k	47k	
	Entity #	23,523	5,943	5,654	
Chunking	Sent #	8,936	-	2,012	10
	Token #	212k	-	47k	
	Chunk #	107k	-	24k	
POS	Sent #	12,543	2,002	2,077	17
	Token #	204k	25k	25	

Table 3.1. The statistics of three datasets used in chapter 3 and chapter 4

We test several neural tagging systems on three sequence tagging tasks: the CoNLL2003 [61] for named entity recognition, the CoNLL2000 [60] and the universal dependency dataset for part of speech tagging [41]. We conformed to the pre-defined splits into train/dev/test except for chunking that does not contain a validation set. For chunking, sections 15-18 of the Wall Street Journal are used for training, and we randomly sampled 1000 sentences in the training set as the development set, and section 20 is used for tests following recent works [45, 30]. Table 3.1 presents the main characteristics of the training, development and test sets we used.

Most work uses the development set to choose optimal hyper-parameters [28, 35, 58], while others combine development set into the training set to improve the performance [9, 45]. In our work, development datasets are used to select the optimal model among all epochs, and we report scores of the selected model on the test dataset. To reduce the volatility of the system, we conduct each experiment 5 times under different random seeds and report the mean, and standard deviation for each model. All models are implemented using the Pytorch library [43]

3.2. Implementation details for twelve neural models

We analyze neural sequence tagger systems on three benchmarks in an identical setting: these systems can be considered as different combinations of three components: (a) word-level inputs encoder; (b) character-level inputs encoder; (c) inference layer. Previous works can cover 9 kinds of these models shown in table 2.1. Here, we have to point out that the results reported by these papers cannot be compared directly since some papers utilize different handcrafted features [35, 65, 47, 23], some of them are evaluated on different datasets [28, 49, 64, 35, 45] and some of them are trained on the combination of training and development sets [9, 45] or utilizes different data split [23]. In order to make fair comparisons, we build a unified framework to produce the results of twelve neural sequence tagging systems. Our implemented models can give better or comparable results on reproducing reported works. We report the hyper-parameters used by all models in table 3.2. And we report the hyper-parameter settings for word-level and char-lavel encoding layers in table 3.3 left part and table 3.3 right part respectively. All models are implemented using the Pytorch library [43].

Layer	Parameter	
Word Embedding	type	GloVe
	dimension	100
Character-level Embedding	dimension	30
Dropout	rate	0.5

Table 3.2. Hyper-parameters of twelve neural based models

Layer	Parameter	
Word-LSTM	hidden size	256
	layer#	2
Word-CNN	kernel size	3
	padding	1
	stride	1
	channel	50
	layer#	4

Layer	Parameter	
Char-LSTM	hidden size	50
	layer#	1
Char-CNN	kernel size	3
	padding	1
	stride	1
	channel	50
	layer#	1

Table 3.3. Hyper-parameters of word encoding layers (left) and character encoding layers (right).

3.3. Results on three tasks

3.3.1. NER

In Table 3.4, we show results of our models and reported by others on NER task. We need to mention that it is hard to reproduce their best results. Five of eighth of our models achieve better results than their corresponding models. For the other three models, we can give competitive results for Word-LSTM + CRF and Word-LSTM + CRF + Char-CNN, the difference is -0.03 and -0.1, respectively. We fail to reproduce comparable result of Strubell et al. [58]. In our experiments, we observe that i) Word-LSTM models get better results than Word-CNN models; ii) Models using character representations outperform models with no character representations; iii) CRF layer increases the accuracy consistently for all models.

F1-score		**NER**		
		No Char	**Char-LSTM**	**Char-CNN**
Word-LSTM	Reported	87.00 [35]	89.15 [28]	89.36 [35]
	Ours (Mean±std)	88.55 ± 0.15	90.63 ± 0.12	90.60 ± 0.22
Word-CNN	Reported	89.97 [58]	-	-
	Ours (Mean±std)	88.45 ± 0.11	90.35 ± 0.21	90.33 ± 0.11
Word-LSTM + CRF	Reported	90.10 [23]	90.94 [28]	91.21 [35]
	Ours (Mean±std)	89.98 ± 0.09	91.04 ± 0.10	91.11 ± 0.25
Word-CNN + CRF	Reported	89.59 [12]	-	-
	Ours (Mean±std)	89.65 ± 0.11	90.47 ± 0.21	90.57 ± 0.11

Table 3.4. F1-scores on the CoNLL03 NER dataset.

3.3.2. Chunking

In Table 3.5, we show results of our models and reported by others on chunking task. Compared with six reported results, we can get better or competitive results. We need to mention that directly comparing our results with reported results is not fair, as for chunking task, several works use different setups. Again, in our experiments, we observe that i) Word-LSTM models get better results than Word-CNN models; ii) Models using character representations outperform models with no char; iii) The CRF layer can increase the accuracy consistently for all models.

3.3.3. POS

Table 3.6 shows the accuracy of POS task. We first need to mention that in [47, 64, 6], they report results evaluated on UD POS dataset v1.2, and Bjerva et al. [6] further test their model on v1.3 and we use this result. The best reported result is 96.73%, as they utilize adversarial training to boost the performance. Plank et al. [47] add an auxiliary loss during his training and gets 96.4%. This makes it unfair to compare the results directly. In our experiments, we find that i) Word-LSTM models also get better results than Word-CNN models; ii) Models using character representations outperform models with no char; iii) the

F1-score		Chunking		
		No Char	Char-LSTM	Char-CNN
Word-LSTM	Reported	94.13 [65]	-	-
	Ours (Mean±std)	94.39 ± 0.12	94.55 ± 0.05	94.56 ± 0.10
Word-CNN	Reported	-	-	-
	Ours (Mean±std)	94.25 ± 0.06	94.42 ± 0.02	94.43 ± 0.02
Word-LSTM + CRF	Reported	94.46 [23]	93.15 [49]	95.00 [45]
	Ours (Mean±std)	94.61 ± 0.09	94.87 ± 0.10	94.93 ± 0.05
Word-CNN + CRF	Reported	94.32 [12]	-	-
	Ours (Mean±std)	94.41 ± 0.16	94.65 ± 0.09	94.6 ± 0.13

Table 3.5. F1-score on the CoNLL00 Chunking dataset.

Accuracy		POS		
		No Char	Char-LSTM	Char-CNN
Word-LSTM	Reported	96.40 [47]	-	-
	Ours (Mean±std)	95.21 ± 0.25	95.54 ± 0.13	95.41 ± 0.11
Word-CNN	Reported	-	-	-
	Ours (Mean±std)	95.13 ± 0.09	95.32 ± 0.15	95.26 ± 0.07
Word-LSTM + CRF	Reported	-	96.73 [64]	-
	Ours (Mean±std)	95.22 ± 0.11	95.67 ± 0.12	95.53 ± 0.10
Word-CNN + CRF	Reported	-	-	95.67 [6]
	Ours (Mean±std)	95.08 ± 0.14	95.44 ± 0.10	95.35 ± 0.13

Table 3.6. Accuracy on the UD POS dataset.

CRF layer has positive influence on POS, but the effect is not as strong as for NER and chunking. We will analyze this in the next section for the three tasks together.

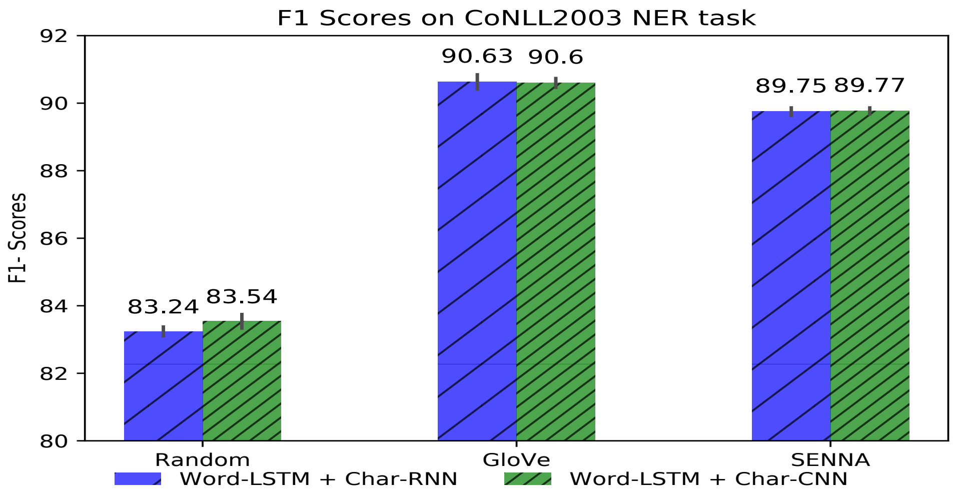
3.4. Comparisons of neural tagger models

Word Representation Methods: For each task, we compute the average difference of performance of models based on two encoding approaches (Word-LSTM and Word-CNN).

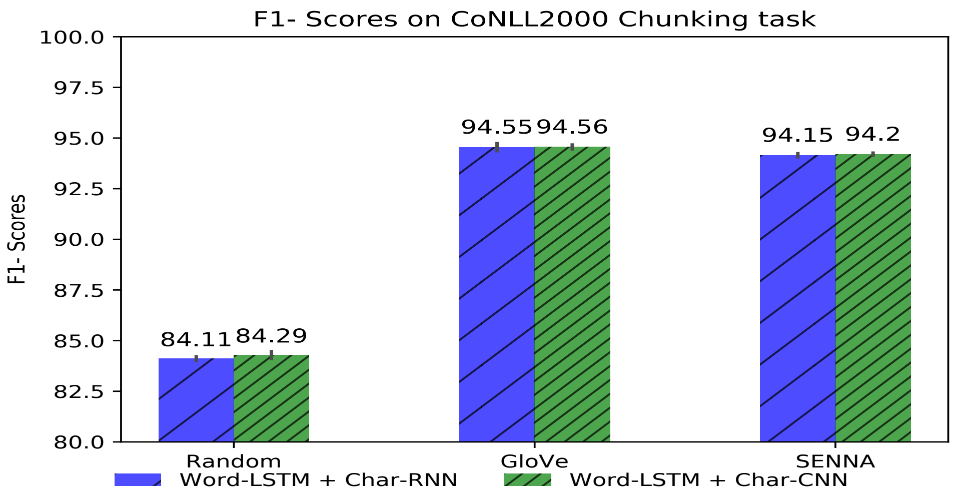
Compared with Word-CNN, we observe that Word-LSTM achieves +0.2, +0.1 and +0.1 on NER, chunking and POS task respectively. We conclude that LSTM models are more effective than CNN models in most cases, which is also consistent with results reported in previous papers shown in Table 3.4, 3.5, 3.6. And the best reported results of three tasks (NER:91.21, chunking:95.0 and POS: 95.67) are all based on Word-LSTM. This suggests that global word context information is necessary for sequence labeling.

Character Representation Methods: We first inspect the effectiveness of character representations by comparing results with or without character representations. In our experiments, character representations can obviously boost both Word-LSTM and Word-CNN based models. Word-LSTM models get a promotion of +1.58, +0.23 and +0.32 on NER, chunking and POS respectively, and Word-CNN models get an improvement of +1.37, +0.20 and +0.20 on NER, chunking and POS respectively. We conclude that character-level representations can improve the performance significantly in three sequence labeling tasks. Next we compare the effect of Char-LSTM and Char-CNN. In our experiments, the Char-LSTM can improve the performance by +1.57, +0.21 and +0.29 on average for NER, chunking and POS, while Char-CNN give +1.58, +0.19 and +0.26, the difference is small, so we conclude that both Char-LSTM and Char-CNN methods have similar effects on the improvement of performance, while the difference between Char-LSTM and Char-CNN is not significant. In most cases, Char-LSTM and Char-CNN tend to give comparable results under different settings and different tasks. Char-CNN gives the best NER result under the Word-LSTM+CRF framework, while Char-LSTM gets better NER results in all other configurations. For chunking and POS tagging, Char-LSTM consistently outperforms Char-CNN under all settings, while the difference is statistically insignificant.

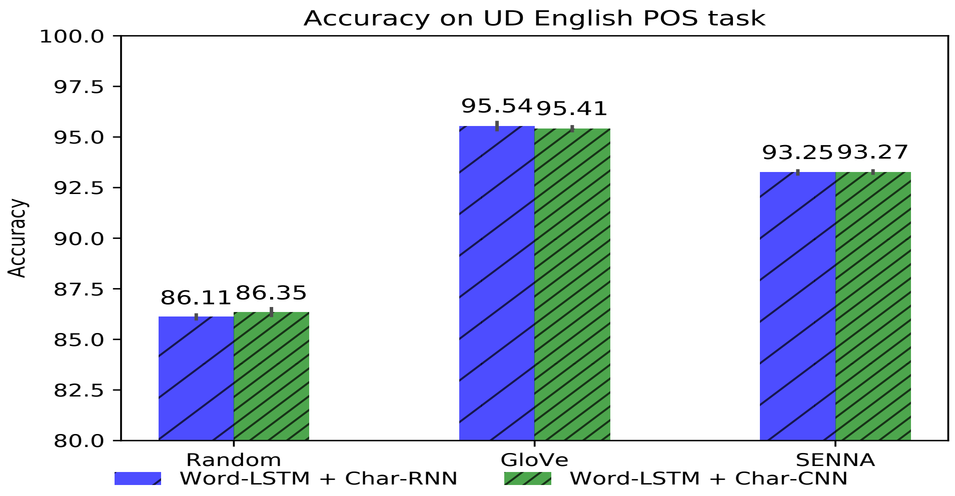
Inference Methods: We compute the performance difference for three tasks between models with a CRF layer and corresponding models with a softmax layer and get +0.65 +0.25 +0.06 for NER, chunking and POS, respectively. As for NER and chunking, the promotions are more important than for POS. Besides, for NER and chunking tasks, models with a CRF layer outperform models with a softmax layer under all configurations consistently, while it is not always the case for the POS task. In most cases, the POS task gets slight improvements. This shows that considering the dependence of labels is still helpful for the POS task. We conclude that CRF always performs better than softmax



(a)

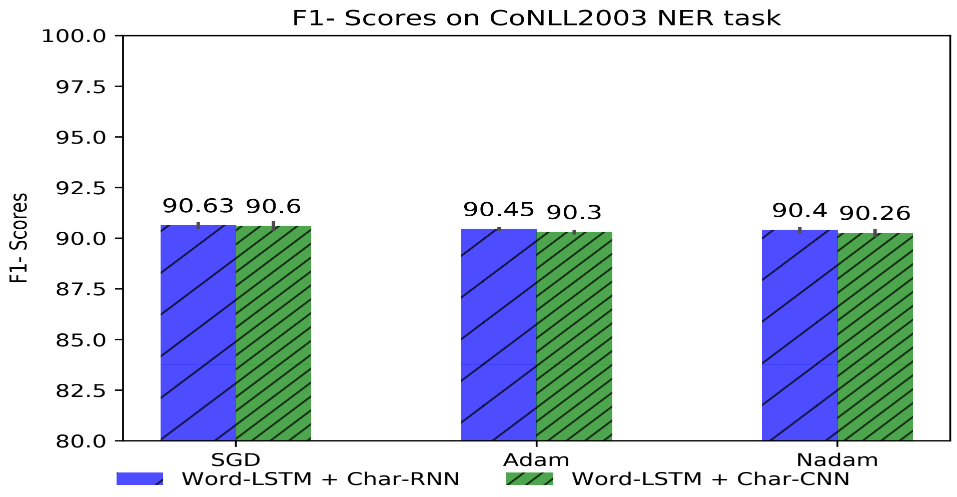


(b)

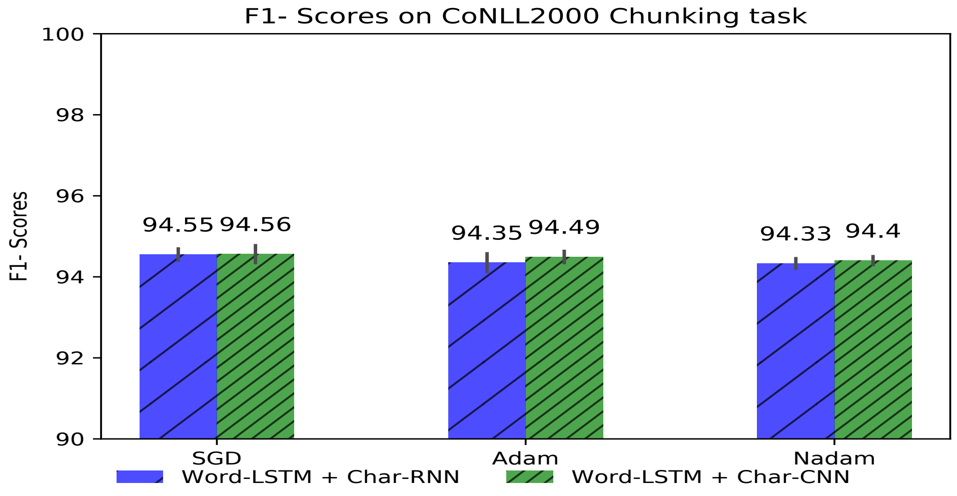


(c)

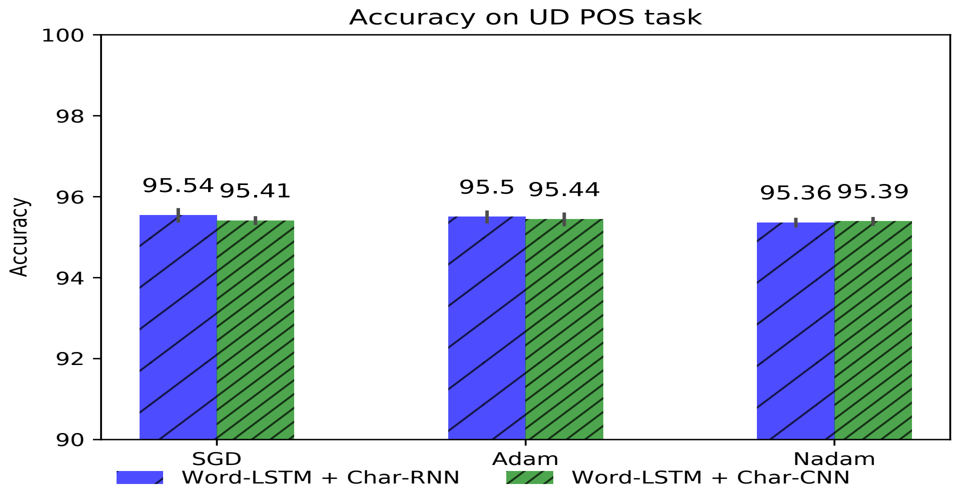
Figure 3.1. Influence of pre-trained embeddings on three tasks.



(a)

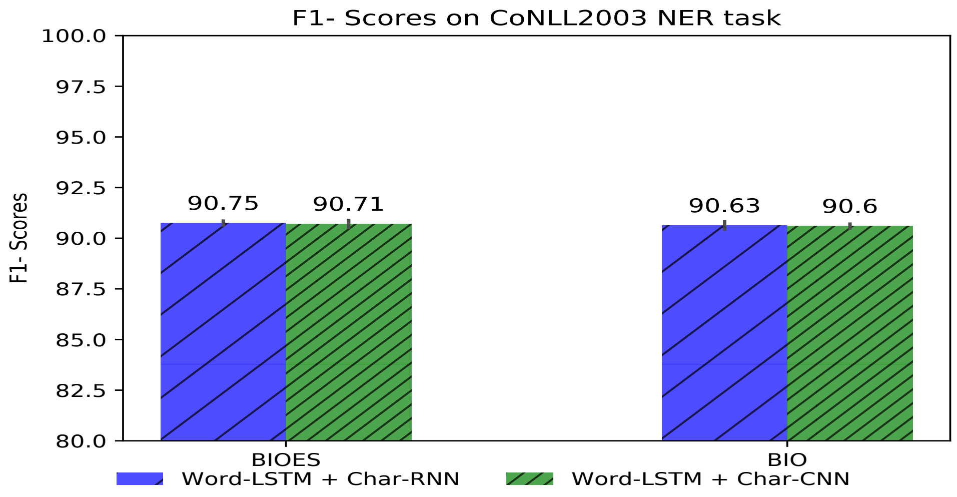


(b)

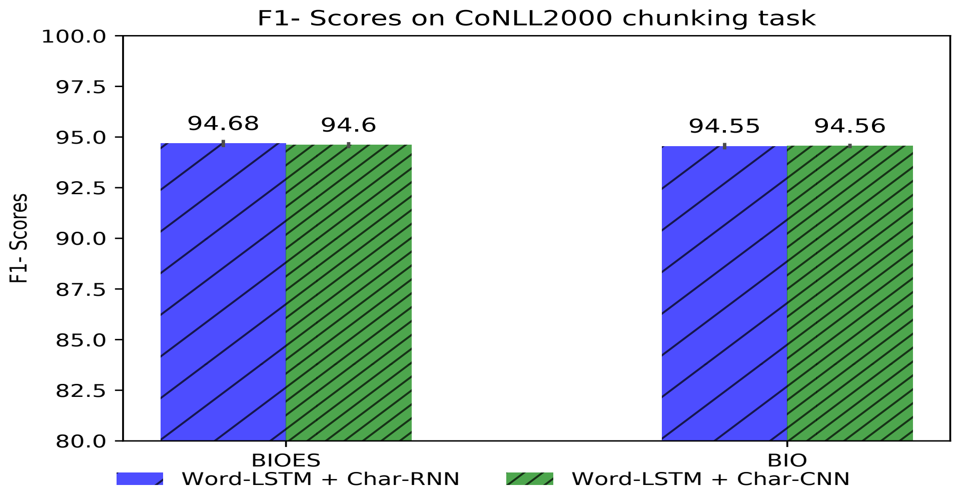


(c)

Figure 3.2. Influence of optimizers on three tasks.



(a)



(b)

Figure 3.3. Influence of tagging schemes on NER and chunking.

classifier for NER and chunking, while CRF fails to give an obvious gain for POS. The reason maybe that by using a tagging scheme, the labels of NER and chunking task tends to contain more information and constrains than POS. As compared to the softmax layer, the process of CRF inference is more time-consuming algorithms (i.e. viterbi algorithm), therefore for a task without tag scheme, we might choose a softmax based model to save time.

In the end, we conclude that LSTM based models outperform CNN based models for encoding word-level information; Both LSTM and CNN based methods can encode character-level information which improves performance for all sequence labeling tasks; while the improvement varies slightly according to different tasks. The CRF inference approach performs better on NER and Chunking than on the POS task.

Besides different neural components in the tagger system, previous works endeavour to make use of different extra resources like pre-trained word embeddings or different optimization algorithms to achieve best accuracy. Such external factors can significantly influence model performance. To make an empirical analysis, we examine three factors (pre-trained embeddings, optimizers, and tagging schemes). These comparisons are based on two kinds of models Word-LSTM + Char-LSTM and Word-LSTM + Char-CNN. To simplify the experimental setting and save time, we do not consider Word-CNN based models and CRF layers. The hyper-parameters are kept same as in Table 3.2 and 3.3 except when we test different pretrained embeddings.

Pretrained embedding. We examine two models on NER, chunking and POS tasks by using random initialization and two different pre-trained embeddings (GloVe and SENNA). Results are shown in Figure 3.1(a)(b)(c). Models with pre-trained embeddings achieve considerable gains over models with random initialization. In our experiments, the GloVe embeddings always give higher accuracy than SENNA on both models, while the difference is not significant for the chunking task. For NER and POS, GloVe embeddings performs much better than SENNA embeddings. Here we have to mention that we used the available resource of GloVe and SENNA[1] directly. The dimension of SENNA embedding is 50, while GloVe is 100. Retraining these embeddings will introduce extra efforts and more uncontrollable factors, which further complifies the comparison. Even if results are consistent with the analysis in [35, 62], we further find it is hard to compare the results in a fair setting, as Ma and Hovy [35] gets about +0.9% and +0.1% gains on NER and POS respectively, Yang et al. [62] gets + 0.46% on NER. We all report the Glove embeddings perform better, but the gains can be influenced by different settings. For the two remaining factors, such issues still exist, so we will only focus on qualitative analysis rather than quantitative analysis.

[1]The embedding files `glove6B` and `SENNA` are available at `https://nlp.stanford.edu/projects/glove/` and `https://ronan.collobert.com/senna/download.html`

Optimizer: We compare the results by training our models under three types of optimizers including SGD, Adam and Nadam on three datasets. We also measure number of epochs that the model needs to converge for each optimizer. In our experiments, to achieve the accuracy reported, SGD optimizer needs 70 epochs on average, while Adam and Nadam optimizer only need less than 30 epochs. Our results in Figure 3.2 show that SGD optimizer always produce better performance than other two optimizers on three tasks. The difference between Nadam and Adam is not significant. For single task learning models, we conclude SGD tends to achieve best results. This observation is consistent with [9, 28, 35], while in [50] they report opposite result.

Tagging scheme. As Figure 3.3 shows, we examine two models NER and chunking tasks by using two different tag schemes on : BIO and BIOES. The experimental results show that models with BIOES are better than BIO on NER and chunking tasks, while the gain on chunking is not significant. Our observation that BIOES can give obvious improvement over BIO on NER task is consistent with [48, 62], while Reimers and Gurevych [50] report the difference is not considerable between BIO and BIOES.

In the end, we conclude that pre-trained word embeddings boost the performance of sequence labeling tasks greatly; and the pre-trained GloVe embeddings outperform SENNA ones. For optimization methods, SGD gives slightly better results than Adam and Nadam methods, while it consumes much longer training time to converge. For tagging scheme, BIOES has a positive influence on NER and Chunking.

Chapter 4

EXPERIMENTS WITH MULTI-TASK LEARNING

This chapter investigates the neural sequence labeling models in the multi-task learning setting. We first compare our proposed SC-LSTM based models with two shared encoder methods: LSTM-s and LSTM-d; Next, we compare our best models with other state-of-the-art methods. Finally, we analyze the influence of original LSTM and SC-LSTM on training in MTL setting.

4.1. Implementation details

Our experiments are based on the results shown in the previous chapter. In this chapter, all models are based on bidirectional LSTM or SC-LSTM. The input of these models can consist of four components: word embedding (GloVe), capitalization features[1], character-level representations and contextual representations. In section 4.2, the inputs of models are word embeddings with capitalization features. In section 4.3, we test two additional variants of the SC-LSTM based model: i) SC-LSTM-CNN-CRF takes the concatenation of the GloVe embedding and the character-level representation of the current token as input. ii) The input of SC-LSTM-LM-CNN is the concatenation of ELMo embeddings, GloVe and character-level representation.

Our best model is SC-LSTM-LM-CNN which gives our best results for three tasks and is used to compare with other state-of-the-art methods. We used mainly the configuration advocated in [46] except for the hidden size of SC-LSTM, which we report in Table 4.2. During the training, we further weighted the NER task in the objective function of Equation 2.2.3 to

[1]Eight features encode capitalization patterns, such as AllUpper, InitialUpper, etc.

	GloVe	Cap. feat.	Char. Rep,	ELMo
LSTM (STL)	✓	✓		
LSTM-s	✓	✓		
LSTM-d	✓	✓		
SC-LSTM	✓	✓		
SC-LSTM-CNN-CRF	✓		✓	
SC-LSTM-LM-CNN	✓		✓	✓

Table 4.1. Models with different input components.

3 (λ_{NER}), the weights of the other tasks where set to 1.[2] We trained this model SC-LSTM-LM-CNN using the Adam optimizer [25] as a default setting. Such a model spends typically less than 30 epochs to converge. We choose the mini-batch size of 10 and used the gradient clipping threshold 5. All models are implemented using the Pytorch library [43].

For chunking and NER tasks, there are several tagging schemes. As our purpose is not to investigate the influence of such a factor, we choose the most commonly used BIO one.

Layer	Parameter	
Word Embedding	dimension	100
Character-level Embedding	dimension	16
Character-CNN	kernel size	3
	padding	1
	stride	1
	channel	128
SC-LSTM	hidden size	256∗3
	layer	3
Contextual Embedding (ELMo)	dimension	1024
Dropout	rate	0.5

Table 4.2. Hyper-parameters of SC-LSTM-LM-CNN

[2]We found the loss of the NER task to be around 1/3 of the loss of the chunking and POS tasks.

4.2. LSTM-s, LSTM-d versus SC-LSTM

In this section, we compare MTL approaches based on LSTM or SC-LSTM. The results are shown in Table 4.3. We first train Bi-LSTM models in a single task learning manner (STL) and put corresponding results at line 1 in Table 4.3; the rest of results in Table 4.3 are data in multi-task learning manner.

For LSTM-s and LSTM-d, we regard one task as the main task and the others as auxiliary tasks; a setting consistent with Søgaard and Goldberg [56]. Because LSTM-s and LSTM-d always fail to achieve stable and competitive results on the three tasks we considered at the same time, we report the best performance we could obtain for each task (line 2 and 3) specifically. On the contrary, our SC-LSTM model is trained once jointly on all tasks, and only one model is being tested in the end, which is much easier and more realistic of a real deployment (line 4).

The results show that our SC-LSTM model improves the performance of the three tasks simultaneously compared with LSTM (STL), and outperforms the other two MTL methods. By jointly learning three tasks, both LSTM-s and LSTM-d can boost the chunking task significantly, but both fail to improve NER and POS tasks. This is consistent with observations made in [12, 56]. We also observe that our SC-LSTM model also benefits the chunking task the most. We will further analyze the reason later in this chapter. Here we need to mention that the results of LSTM(STL) in Table 4.3 are different from previous chapter, as we use another set of hyper-parameters.

	POS	chunking	NER
LSTM (STL)	95.46	94.44	89.39
LSTM-s	95.45	95.12	89.35
LSTM-d	95.44	95.24	89.37
SC-LSTM	**95.51**	**96.04**	**89.96**

Table 4.3. Results of models being trained in STL or MTL mode. For all MTL models, we report the best performance via a small grid search over combinations of the hidden size [100, 200, 300, 400] and the number of layers [1, 2, 3]. The best performance of each MTL model was obtained with hidden size 300 and 3 layers.

4.3. SC-LSTM versus state-of-the-art

To further demonstrate the effectiveness of our SC-LSTM model, we compare different variants with state-of-the-art approaches, that we classify into three broad categories:

- **Single sequence labeling** where models are trained without the supervision of other tasks. Specifically, we compare our results to the LSTM-CRF model of [28] and the LSTM-CNN-CRF one [35], since those are state-of-the-art singly tasked sequence labelers. LSTM-CRF uses bi-LSTM for encoding character-level information; while LSTM-CNN-CRF uses CNN for same purpose. Both two approaches use bi-LSTM to encode word-level information and CRF for inference.
- **Multi-tasked sequence labeling** where models leverage the supervision of other tasks. We compare our model with the representative approaches of [34, 56, 11, 12].
- **Models with language model.** Recently, several studies in using contextualized word embeddings achieved great success in a number of tasks. Some recent studies [45, 49, 46, 14] are particularly considered.

NER: Results for the CoNLL 2003 dataset are reported in Table 4.4. We observe that our SC-LSTM-LM-CNN model outperforms all approaches but Devlin et al. [14] and Akbik, Blythe, and Vollgraf [1]. The latter work is using the development set as training material, which avoids a direct comparison. The former model (BERT) is achieving great success by leveraging a huge amount of unannotated data as well as a lot of computation resources we could not afford in this study. We are however pleased that our model is leveraging contextual embeddings with 0.38 absolute F1 improvement over the results of [46].

Chunking: We compared a number of models on the CoNLL2000 chunking dataset. A few of them [35, 28, 46] where not tested on this benchmark, and we reimplemented them. We also trained the companion toolkits of those models, but (as detailed in the next section) got slightly lower results for some reasons. Table 4.5 reports the performance of the many approaches we tested. We observe that our SC-LSTM-LM-CNN architecture achieves a new state-of-the-art F1 score, with over 1 absolute point over the competitive approach of Peters et al. [45], and an improvement of 0.4% over the current state-of-the-art method [10].

Model	F1-score(mean)
Collobert et al. [12]	89.59†
Chiu and Nichols [9] ♣	91.62
Huang et al. [23]	88.83
Ma and Hovy [35]	91.21
Lample et al. [28]	90.94
Shen et al. [55]	90.89
Yang, Salakhutdinov, and Cohen [63]	91.20
Rei [49]	86.26
Liu et al. [30]	91.71
Peters et al. [45]	91.93
Zhang et al. [68]	91.2
Liu et al. [31]	91.95
SC-LSTM	89.96
SC-LSTM-CNN-CRF	91.37
SC-LSTM-LM-CNN	**92.60**

Table 4.4. F1-score on the CoNLL03 NER dataset. Models with ♣ use both train and dev splits for training.

POS: We conducted experiments on the Universal Dependency POS English dataset and present the results in Table 4.6. The only study we found that reports results on the UD v1.3 benchmark we used here is [6], and we report the results they published. For Liu et al. [30], Peters et al. [46] we used the available companion toolkits[3] that we trained ourself with the default settings. We re-implemented the other approaches.

Again, we observe that SC-LSTM-LM-CNN outperforms all other approaches we tested. The absolute improvement in F1 score over the current state-of-the-art of Peters et al. [46] is 0.21%.

[3]`https://github.com/LiyuanLucasLiu/LM-LSTM-CRF` and `https://github.com/allenai/allennlp`

Model	F1-score(mean)
Collobert et al. [12]	94.32†
Huang et al. [23]	94.13
Ma and Hovy [35]	94.81†
Lample et al. [28]	94.68†
Hashimoto et al. [20]	95.77
Søgaard and Goldberg [56]	95.56
Shen et al. [55]	93.88
Yang et al. [63]	94.66
Liu et al. [30]	95.96
Peters et al. [45]	96.37
Liu et al. [31]	96.13
Akbik et al. [2]	96.72
Clark et al. [10]	97.00
SC-LSTM	96.04
SC-LSTM-CNN-CRF	96.41
SC-LSTM-LM-CNN	**97.40**

Table 4.5. F1-score on the CoNLL00 chunking dataset. Configurations with a † sign are approaches we reimplemented.

In order to further validate our implementations, we also ran the toolkits of Ma and Hovy [35] and Lample et al. [28] and obtained slightly lower results[4].

4.4. Analysis of SC-LSTM

We conducted a number of investigations in order to understand better why our multi-task learning model is effective. We first concentrate on the training process to investigate how the combinations of different tasks influence on the optimization process; then we study the compatibility of different tasks in MTL.

[4]We obtained 95.7% for the toolkit we took from `https://github.com/XuezheMax/NeuroNLP2` and 95.6% for the one at `https://github.com/glample/tagger`.

POS	Accuracy
Collobert et al. [12]	95.41†
Huang et al. [23]	95.63†
Ma and Hovy [35]	95.80†
Lample et al. [28]	95.78†
Bjerva et al. [6]	95.67
Alonso and Plank [3]	94.54
Changpinyo et al. [8]	95.4
Liu et al. [30]	95.95†
Peters et al. [46]	96.62 †
SC-LSTM	95.51
SC-LSTM-CNN-CRF	95.83
SC-LSTM-LM-CNN	**96.83**

Table 4.6. Accuracy on the Universal Dependency English POS dataset. Configurations with a † sign are approaches we reproduced.

4.4.1. Convergence Analysis

We report in Figure 4.1 the convergence of different MTL models on the development set. To obtain those curves, we collected the F1-score on the NER and chunking tasks as well as the accuracy of the POS task, and averaged them after each epoch.

We clearly see that the SC-LSTM model converges faster than other ones. It achieves higher performance after the first epoch, and after about ten epochs, it shows a smooth performance curve, while LSTM-s and LSTM-d models still fluctuate. This indicates that our model can learn the hidden representation of multiple tasks in a faster and smoother way than the other two methods.

Besides, we observe in Figure 4.1c and 4.1d that combinations of tasks involving chunking typically show a smooth training curve, on the contrary to Figure 4.1b where NER and POS tasks are combined. The fact that the training regimen fluctuates in the latter case for both LSTM-s and LSTM-d suggests that conflicts with those two tasks happen during optimisation, which we do not observe for our model. Also, Figure 4.1a illustrates

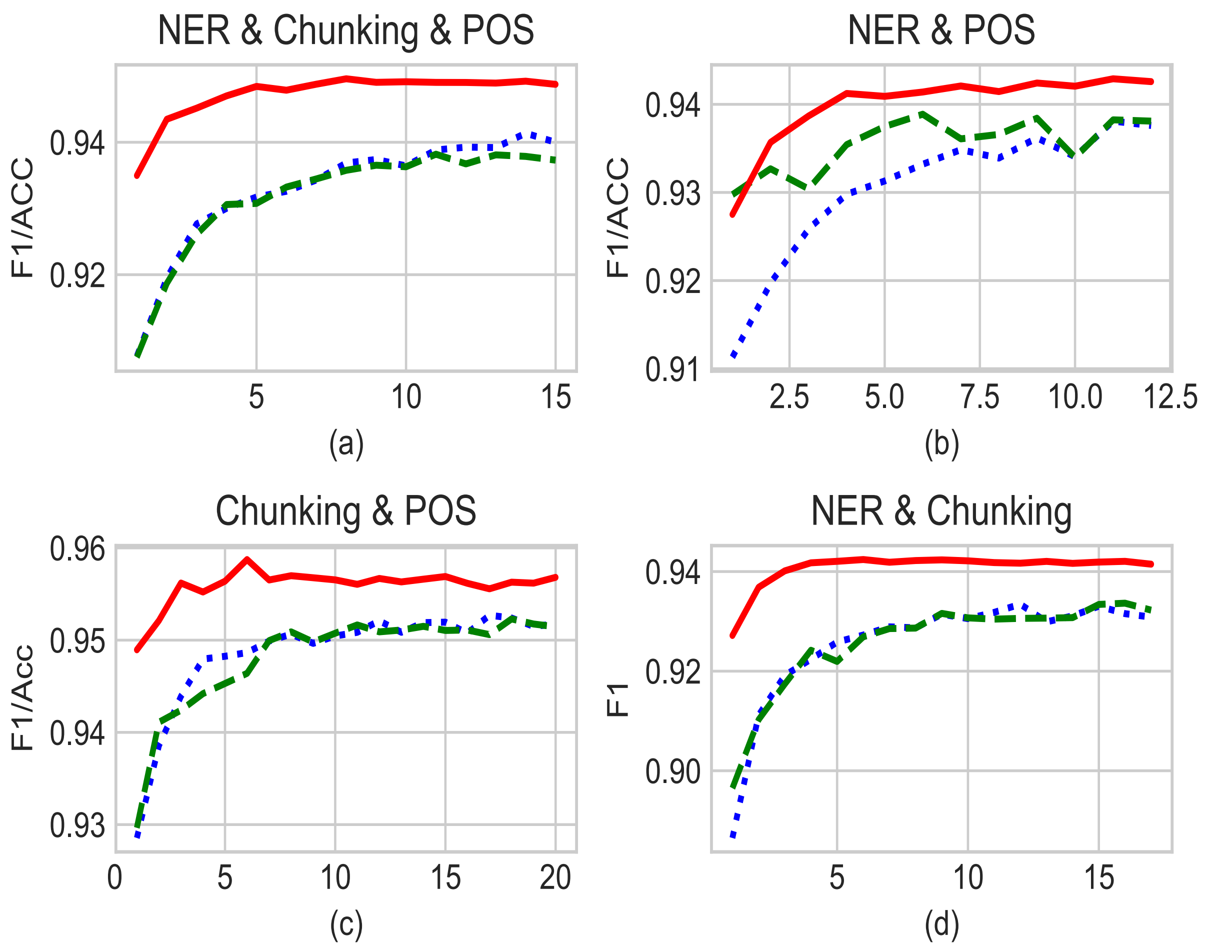


Figure 4.1. The change of F1-score/Accuracy on the development datasets at different epochs for SC-LSTM (red solid line), LSTM-s (blue dotted line) and LSTM-d (green dashed line) baselines. (a) reports results with NER, chunking and POS; (b) results with NER and POS; (c) results with chunking and POS; and (d) results with NER and chunking. All models were trained using the Adam optimizer with the same setting, and the mini-batch size was set to 10.

that combining the three tasks altogether leads to comparably better performance of our model over LSTM-s and LSTM-d.

By tracking the performance of models on development sets, we have shown that the combinations of different tasks have a great influence on the training process in MTL setting. In the following part, we report the corresponding results on test sets to further analyze the effect of grouping different tasks under various multi-task settings. Additionally, this also helps us to better understand the reason why a task can get a higher promotion than another in MTL setting.

4.4.2. Effect of Different Task Combinations

We analyzed which task is benefited or harmed by others under the three MTL settings we considered and present results in Figure 4.2.

We find that by jointly learning with NER or POS leads to better chunking for all MTL models (see in Figure 4.2b). This is in particular the case of our SC-LSTM model which records the largest gain, especially when all tasks are being trained on.

Figure 4.2c shows results obtained on POS. Only our SC-LSTM model achieves a meaningful improvement. We however observe that the NER task tends to hurt the performance of POS, since in most cases, the performance of POS+NER is lower than the one obtained with POS+chunking.

For NER, Figure 4.2 shows that POS hurts LSTM-s and LSTM-d models, while the chunking task is beneficial for all MTL methods.

Clearly, the combination of different tasks has a different effect on the final performance of each task. The chunking task seems compatible with NER and POS tasks, and it boosts the other two tasks in all three MTL settings, which is consistent with the results of Changpinyo et al. [8]. Directly jointly training on POS and NER datasets tends to reduce the performance in LSTM-s and LSTM-d, which is also consistent with the conclusion in [56, 8]. In conclusion, all of the results show that our SC-LSTM model is effective at capturing the mutual benefits of all combined tasks. Since it performs consistently better in various settings, we believe our model to be more robust.

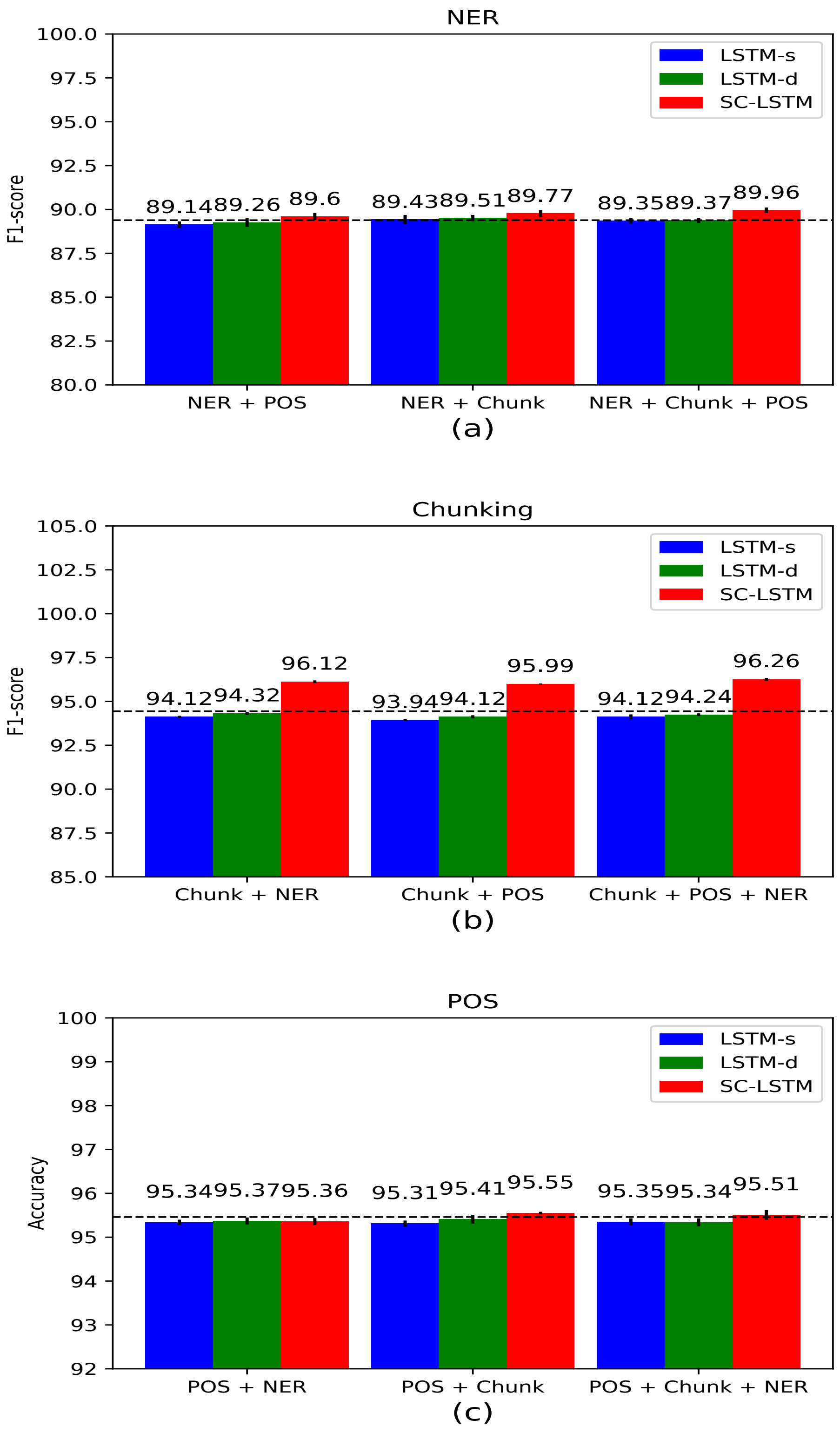


Figure 4.2. Results of different task groups on each test set: (a) NER, (b) chunking, and (c) POS. Horizontal lines show results for single-task models (NER: 89.39, Chunking: 94.44 and POS: 95.46).

Chapter 5

CONCLUSION

In this work, we focus on neural based approaches for sequence labeling tasks. First, We introduce and investigate the different characteristics of three basic components of neural tagger system (word level encoder, character level encoder, and inference layer) and then evaluate them in a unified framework.

We re-implemented and compared recent neural tagger models with different configurations under a unified experimental setting. We first investigate the effect on which LSTM and CNN impact word-level and character-level encoders. Empirical experiments show that models with LSTM encoding perform better than those with CNN encoding. The disadvantage of word LSTM based methods is taking longer decoding time than CNN based approaches. Besides, the experiments show that character-level information is crucial for sequence labeling tasks. We observe that the character information helps to improve performance obviously. Even the character LSTM and character CNN encoding methods capture different structure information in character sequence, in most cases, these two methods will make comparable impacts on results. As the latter one is more efficient in computation, it is better to choose it in practice. Finally, we compare the effects of two inference structures (Softmax layer and CRF layer). The results show that the CRF method outperforms Softmax on NER and text chunking tasks, while it gives comparable results on POS tagging.

External factors such as pre-trained embeddings, tagging schemes, and optimizers have significant influence to the performance. So besides model structures studied above, we also evaluate a set of external factors on the three tasks. We conclude that using pre-trained word embeddings like GloVe and SENNA can greatly improve performance, and that the GloVe embedding has better effect. And for single task learning, SGD is the best optimizer which only need to sacrifice more time during training. The BIOES tagging scheme is also helpful, especially for NER.

Second, we target to solve three sequential tagging by joint training. Despite many successful applications, there are several issues about the effectiveness of MTL for sequence labeling. If we can jointly learn more tasks and benefit them together, we can raise the utility of MTL significantly. There are several avenues of this work we would like to address. We analyze and compare two widely used shared-encoder frameworks for sequence labeling and propose a simple but novel SC-LSTM to tackle the weaknesses of them. We hypothesize that by giving models the capacity to jointly learn task-specific knowledge and task-invariant knowledge can improve the performance. So we propose an LSTM cell that preserves shared parameters and task-specific parameters and allows efficient multi-task training for sequential labeling. We conducted extensive experiments to compare both single-task learning and multi-task learning models. Experiments demonstrate the effectiveness of our model for sequence labeling tasks. Our SC-LSTM model outperforms the other two shared-encoder models when using data of all three tasks to train models. Compared with the current state-of-art results, our best SC-LSTM-LM-CNN model can achieve better results on chunking and POS and gets a comparable result on NER. Furthermore, we conducted analysis to show the effect of our method. Results show that the SC-LSTM model converges faster than others. In most cases, it achieves higher performance after the first epoch and gets smooth while other two shared-encoder methods still fluctuate, which indicates that our model can learn the hidden representation of multiple tasks in a faster and smoother way than the other two methods. The combination of different tasks has a different effect on the final performance of each task. By grouping different tasks together for training, the results show that our SC-LSTM model can efficiently avoid suffering the negative influence introduced by incompatible tasks. Our model is more robust and performs consistently better in various settings.

Bibliography

[1] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649. Association for Computational Linguistics, 2018.

[2] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649, 2018.

[3] Hector Martinez Alonso and Barbara Plank. When is multitask learning effective? semantic sequence prediction under varying data conditions. In *EACL 2017-15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1–10, 2017.

[4] Waleed Ammar, George Mulcaire, Miguel Ballesteros, Chris Dyer, and Noah A Smith. Many languages, one parser. *Transactions of the Association for Computational Linguistics*, 4:431–444, 2016.

[5] Ann Bies, Justin Mott, Colin Warner, and Seth Kulick. English web treebank. *Linguistic Data Consortium, Philadelphia, PA*, 2012.

[6] Johannes Bjerva, Barbara Plank, and Johan Bos. Semantic tagging with deep residual networks. *arXiv preprint arXiv:1609.07053*, 2016.

[7] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[8] Soravit Changpinyo, Hexiang Hu, and Fei Sha. Multi-task learning for sequence tagging: An empirical study. *arXiv preprint arXiv:1808.04151*, 2018.

[9] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *arXiv preprint arXiv:1511.08308*, 2015.

[10] Kevin Clark, Minh-Thang Luong, Christopher D Manning, and Quoc V Le. Semi-supervised sequence modeling with cross-view training. *arXiv preprint*

arXiv:1809.08370, 2018.

[11] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[12] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[13] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603. IEEE, 2013.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Abbas Ghaddar and Phillippe Langlais. Robust lexical features for improved neural network named-entity recognition. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1896–1907, 2018.

[16] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[17] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[20] Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

[21] Daniel Hershcovich, Omri Abend, and Ari Rappoport. Multitask parsing across semantic representations. In *Proceedings of the 56th Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers), volume 1, pages 373–385, 2018.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[24] Young-Bum Kim, Karl Stratos, and Ruhi Sarikaya. Frustratingly easy neural domain adaptation. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 387–396, 2016.

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Eliyahu Kiperwasser and Miguel Ballesteros. Scheduled multi-task learning: From syntax to translation. *arXiv preprint arXiv:1804.08915*, 2018.

[27] John Lafferty, Andrew McCallum, and Fernando CN Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[28] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[29] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research*, 5(Apr): 361–397, 2004.

[30] Liyuan Liu, Jingbo Shang, Frank Xu, Xiang Ren, Huan Gui, Jian Peng, and Jiawei Han. Empower sequence labeling with task-aware neural language model. *arXiv preprint arXiv:1709.04109*, 2017.

[31] Liyuan Liu, Xiang Ren, Jingbo Shang, Jian Peng, and Jiawei Han. Efficient contextualized representation: Language model pruning for sequence labeling. *arXiv preprint arXiv:1804.07827*, 2018.

[32] Xiaodong Liu, Jianfeng Gao, Xiaodong He, Li Deng, Kevin Duh, and Ye-Yi Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. 2015.

[33] Peng Lu, Ting Bai, and Philippe Langlais. Sc-lstm: Learning task-specific representations in multi-task learning for sequence labeling. In *Proceedings of the 2019 Conference*

of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 2396–2406, 2019.

[34] Gang Luo, Xiaojiang Huang, Chin-Yew Lin, and Zaiqing Nie. Joint entity recognition and disambiguation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 879–888, 2015.

[35] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional lstm-cnns-crf. *arXiv preprint arXiv:1603.01354*, 2016.

[36] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[37] Andrew McCallum, Dayne Freitag, and Fernando CN Pereira. Maximum entropy markov models for information extraction and segmentation.

[38] Makoto Miwa and Mohit Bansal. End-to-end relation extraction using lstms on sequences and tree structures. *arXiv preprint arXiv:1601.00770*, 2016.

[39] Dat Quoc Nguyen, Mark Dras, and Mark Johnson. A novel neural network model for joint pos tagging and graph-based dependency parsing. *arXiv preprint arXiv:1705.05952*, 2017.

[40] Jan Niehues and Eunah Cho. Exploiting linguistic resources for neural machine translation using multi-task learning. *arXiv preprint arXiv:1708.00993*, 2017.

[41] Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, 2016.

[42] Robert Östling and Jörg Tiedemann. Efficient word alignment with markov chain monte carlo. *The Prague Bulletin of Mathematical Linguistics*, 106(1):125–146, 2016.

[43] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[44] Hao Peng, Sam Thomson, and Noah A Smith. Deep multitask learning for semantic dependency parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2037–2048, 2017.

[45] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1756–1765, 2017.

[46] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[47] Barbara Plank, Anders Søgaard, and Yoav Goldberg. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. *arXiv preprint arXiv:1604.05529*, 2016.

[48] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the thirteenth conference on computational natural language learning*, pages 147–155. Association for Computational Linguistics, 2009.

[49] Marek Rei. Semi-supervised multitask learning for sequence labeling. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2121–2130, 2017.

[50] Nils Reimers and Iryna Gurevych. Optimal hyperparameters for deep lstm-networks for sequence labeling tasks. *arXiv preprint arXiv:1707.06799*, 2017.

[51] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[52] Cicero D Santos and Bianca Zadrozny. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1818–1826, 2014.

[53] Cicero Nogueira dos Santos and Victor Guimaraes. Boosting named entity recognition with neural character embeddings. *arXiv preprint arXiv:1505.05008*, 2015.

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

[55] Yanyao Shen, Hyokun Yun, Zachary C Lipton, Yakov Kronrod, and Animashree Anandkumar. Deep active learning for named entity recognition. *arXiv preprint arXiv:1707.05928*, 2017.

[56] Anders Søgaard and Yoav Goldberg. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 231–235, 2016.

[57] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[58] Emma Strubell, Patrick Verga, David Belanger, and Andrew McCallum. Fast and accurate entity recognition with iterated dilated convolutions. *arXiv preprint arXiv:1702.02098*, 2017.

[59] Jun Suzuki and Hideki Isozaki. Semi-supervised sequential labeling and segmentation using giga-word scale unlabeled data. *Proceedings of ACL-08: HLT*, pages 665–673, 2008.

[60] Erik F Tjong Kim Sang and Sabine Buchholz. Introduction to the conll-2000 shared task: Chunking. In *Proceedings of the 2nd workshop on Learning language in logic and the 4th conference on Computational natural language learning-Volume 7*, pages 127–132. Association for Computational Linguistics, 2000.

[61] Erik F Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 142–147. Association for Computational Linguistics, 2003.

[62] Jie Yang, Shuailong Liang, and Yue Zhang. Design challenges and misconceptions in neural sequence labeling. *arXiv preprint arXiv:1806.04470*, 2018.

[63] Zhilin Yang, Ruslan Salakhutdinov, and William W Cohen. Transfer learning for sequence tagging with hierarchical recurrent networks. *arXiv preprint arXiv:1703.06345*, 2017.

[64] Michihiro Yasunaga, Jungo Kasai, and Dragomir Radev. Robust multilingual part-of-speech tagging via adversarial training. *arXiv preprint arXiv:1711.04903*, 2017.

[65] Feifei Zhai, Saloni Potdar, Bing Xiang, and Bowen Zhou. Neural models for sequence chunking. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[66] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[67] Yuan Zhang and David Weiss. Stack-propagation: Improved representation learning for syntax. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1557–1566, 2016.

[68] Yuan Zhang, Hongshen Chen, Yihong Zhao, Qun Liu, and Dawei Yin. Learning tag dependencies for sequence tagging. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4581–4587. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi: 10.24963/ijcai.2018/637. URL `https://doi.org/10.24963/ijcai.2018/637`.