DAP: LSTM-CRF Auto-encoder

Yuan Liu
Carnegie Mellon University
yuanl4@andrew.cmu.edu

Advisor: Matthew R. Gormley
Carnegie Mellon University
mgormley@cs.cmu.edu

Abstract

The LSTM-CRF is a hybrid graphical model which achieves state-of-the-art performance in supervised sequence labeling tasks. Collecting labeled data consumes lots of human resources and time. Thus, we want to improve the performance of LSTM-CRF by semi-supervised learning. Typically, people use pre-trained word representation to initialize models embedding layer from unlabeled data. However, these word representations are model-agnostic, which means they were trained without knowledge of the LSTM-CRFs structure. Thus, we introduce auto-encoder training for the LSTM-CRF to tune the models parameters by adding a decoder after the CRF. We compare the performance of these two methods and find that while the auto-encoder can improve performance in some situation, the model-agnostic approach achieves more universal improvement. We also how to use mini-batches in LSTM-CRF.

1 Introduction

In sequence labeling tasks, a single categorical label is assigned to each element of the sequential input. It covers lots of natural language processing (NLP) tasks, including part of speech (POS) tagging, chunking, and named entity recognition (NER). How to establish an accurate and computationally efficient sequence labeling model is a cornerstone of NLP.

The key point of sequence labeling is to utilize both the past and future context to make a prediction. Based on this idea, a number of probabilistic graphical models and neutral networks have been established to solve sequence labeling tasks. The hidden Markov model (Eddy, 1996), maximum entropy Markov model (McCallum et al., 2000) and conditional random field (CRF) (Lafferty et al., 2001) are high-performance sequence labeling models. They rely heavily on hand-crafted features and task-specific resources. Other more complex and deeper neural networks have also been proposed to solve these tasks, such as long short term memory networks (LSTM) (Hochreiter and Schmidhuber, 1997).

In recent years, the bi-directional LSTM-CRF achieves state-of-the-art performance on some sequence labeling tasks (Huang et al., 2015). It is a hybrid model that combines the feature learning of a neural network with the effective structured prediction of probabilistic graphical models. In this paper, we want to increase the performance of this model via semi-supervised learning.

The embedding layer plays a vital role in the LSTM-CRF (Collobert et al., 2011). It converts each one-hot encoded word into a continuous vector space with many fewer dimensions. Lots of work has shown a good initialization of this layer can increase the performance of models significantly (Lample et al., 2016). People often use pre-trained word representation, such as GloVe (Pennington et al., 2014) and SENNA (Collobert et al., 2011) to initialize it. However, the word representation training process does not fully utilize the structure of LSTM-CRF. It is a model-agnostic method.

In this paper, we introduce a model-specific semi-supervised method to tune the embedding layer. Ammar et al. (2014); Lin et al. (2014) propose the CRF auto-encoder, which regenerates the input sentences according to the marginal distribution of a CRF. Thus, they can use large unlabeled datasets to train the whole model. We generalize this method to obtain the LSTM-CRF auto-encoder.

Our primary contribution is the application of the auto-encoder structure to the LSTM-CRF. We first maximize the supervised log-likelihood of CRF layer to initialize the LSTM-CRF. Then train the embedding layer's parameters by the auto-encoder structure. When we are tuning the embedding layer, the information of LSTM-CRF is accessible. In this sense, we obtain a model-specific method. According to the result of Ammar et al. (2014); Lin et al. (2014), modeling unlabeled data using CRF autoencoders did not improve prediction accuracy. We figure out this issue, thanks to the subsampling probabilities mentioned in Arora et al. (2016).

We use CoNLL-2000 (Tjong Kim Sang and Buchholz, 2000) as labeled datasets, and evaluate the performance of our model-specific method with various proportion of unlabeled Wikipedia datasets (Al-Rfou et al., 2013).

The remainder of the paper is organized as follows. Section 2 describes the LSTM-CRF. Section 3 shows how to apply the auto-encoder structure to the LSTM-CRF. Section 4 shows the training procedure and experiments result.

2 LSTM-CRF

The LSTM-CRF is the state-of-the-art method for some sequence labeling tasks. In this section, we provide a brief introduction of the LSTM-CRF, starting from its three key components: embedding layer, LSTM, CRF.

2.1 Embedding Layer

Representations of words in a continuous vector space play a vital role in natural language processing by grouping similar words (Mikolov et al., 2013), which is the goal of embedding layer in sequence labeling tasks. Following prior work (Lample et al., 2016), we concatenate word-level embedding and LSTM based character-level embedding to do word representation.

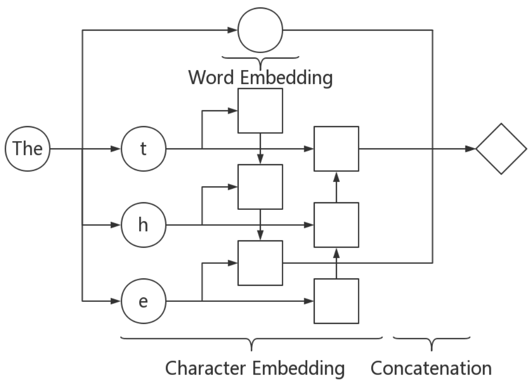

Figure 1: Embedding Layer.

2.1.1 Word-level Embedding

For efficiency, words are fed to neural network as indices taken from a finite dictionary. Obviously, a simple index does not carry much useful information about the word. It has been shown in (Collobert et al., 2011) that word-level embedding plays a vital role to improve sequence labeling performance.

Thus, we need a word-level embedding layer to map each word index into a vector in continuous space, by a lookup table operation. Mathematically, for the x^{th} word in the finite dictionary $\mathcal{D}$, the d_{word}-dimensional representation vector is given by the lookup table layer $LT_W(\cdot)$:

$$LT_W(x) = \boldsymbol{W}_{x,:} \tag{1}$$

In Eq.1, $\boldsymbol{W} \in \mathbb{R}^{|D| \times d_{word}}$ is a matrix of parameters to be learned, and $\boldsymbol{W}_{x,:}$ is the x^{th} row of $\boldsymbol{W}$.

In large text, the most frequent words occur much more times than rare words. To counter this imbalance, Mikolov et al. (2013) introduce the subsampling of frequent words. In this paper, we adopt the weighted subsampling, which is mentioned in Arora et al. (2016).

$$q(w) = \frac{a}{a + p(w)} \tag{2}$$

When updating the word-level embedding layer, we times the derivative of word w by $q(w)$.

2.1.2 Character-level Embedding

Prefixes and suffixes are sets of letters that can be added at the beginning or the end of words. It is useful for natural language processing particularly in morphologically rich languages. Even in morphologically-poor languages, such as English, such features are useful: for example, in part-of-speech tagging, the suffix -able transforms the verb like into the adjective likeable. By recognizing this suffix, we can distinguish likeable as adjective.

Character-level modeling has long been effective across NLP (Verwimp et al., 2017). We adopt the character-level embedding approach of (Lample et al., 2016). Its architecture is shown in Fig.1. A character lookup table keeps mapping each character in a given word to corresponding feature vector. A bi-directional LSTM treats these feature vectors as input, and concatenates the last hidden state of both forward LSTM and backward LSTM as output. More details about the bi-directional LSTM can be found in Section 2.2. The forward LSTM is capable to learn the prefix information, while the backward LSTM can learn the suffix information.

In our paper, we concatenate the word-level embedding layer and character-level embedding layer as the embedding layer of the LSTM-CRF.

2.2 LSTM Network

When you read this sentence, you understand each word based on your understanding of all previous ones. Your thought is persistent, and you know the order of words in the sentence. The recurrent neural network (RNN) is a family of neural networks, which is designed to capture this intuition via feature learning. Due to the internal memory in RNN, it allows information to pass from one time-slot to the next. It has shown great promise in many NLP problems.

An RNN can take a sequence of vectors $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$ as input and generate another sequence of vectors $(\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_n)$ as output. The output $\boldsymbol{h}_t$ at time t is depended on the previous $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t)$. The dependence is attained by its internal memory. Unfortunately, a vanilla RNN cannot take advantage of all the history information. It tends to be biased towards the most recent input due to the vanishing gradient problem (Bengio et al., 1994).

Long Short-term memory network (LSTM) is a typical variant of RNN, which is designed to fix this issue. The key to LSTM is the cell state. We adopt the conveyor belt analogy of (Olah, 2015). The cell state is kind of like a conveyor belt. The information flows through the belt, with only some minor linear interactions, and keeps long-term dependency. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to let information through optionally. An LSTM has four kinds of gates to protect and control the cell state. They are the input gate, forget gate, cell gate, and out gate. There are many different implementations of these gates. We use the same implementation as Pytorch (Paszke et al., 2017):

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_{ii}\boldsymbol{x}_t + \boldsymbol{b}_{ii} + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hi}) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_{if}\boldsymbol{x}_t + \boldsymbol{b}_{if} + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hf}) \\
\boldsymbol{g}_t &= \tanh(\boldsymbol{W}_{ig}\boldsymbol{x}_t + \boldsymbol{b}_{ig} + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hg}) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{io}\boldsymbol{x}_t + \boldsymbol{b}_{io} + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{ho}) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned} \tag{3}$$

In Eq.3, σ is the element-wise sigmoid function, $\odot$ is the element-wise product, $\boldsymbol{h}_t$ is the hidden state at time t, $\boldsymbol{c}_t$ is the cell state at time t, $\boldsymbol{x}_t$ is the input at time t, and $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{g}_t$, $\boldsymbol{o}_t$ are the input, forget, cell, and out gates, respectively.

According to the above formula, an LSTM can generate an output $\boldsymbol{h}_t$ for the left context of the t^{th} word in the given sentence $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$. We can rewrite $\boldsymbol{h}_t$ as $\overrightarrow{\boldsymbol{h}}_t$. In sequence labeling task, we can have access to both past (left context) and future (right context) input tokens at the same time. It is natural to generate an output of the right context $\overleftarrow{\boldsymbol{h}}_t$. It can be obtained using a second LSTM that reads the same input sequence in reverse. We refer to the former as the forward LSTM and the latter as the backward LSTM. This forward and backward LSTM pair is known as a bi-directional LSTM (Graves and Schmidhuber, 2005). We concatenate the left and right context representation, $\boldsymbol{h}_t = [\overrightarrow{\boldsymbol{h}}_t; \overleftarrow{\boldsymbol{h}}_t]$ as the output of Bi-LSTM.

2.3 CRF

Conditional random field (CRF) is a type of undirected probabilistic graphical model. It is often used for labeling or parsing sequential data, such as part-of-speech (POS), chunking, and name-entity-recognition (NER). In this paper, we mainly use a bigram linear chain CRF to predict labels for sequential input words.

We consider the score matrix $\boldsymbol{P}$ as the input of the CRF. The size of $\boldsymbol{P}$ is $n \times (k+2)$, where n is the length of the corresponding input sentence $(\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n))$, k is the number of distinct tags. We plus 2 here because of the $\underline{\text{start}}$ and $\underline{\text{stop}}$ tag. We use $\boldsymbol{P}$ as the emission matrix in the linear chain CRF and $\boldsymbol{P}_{i,j}$ is the score of the j^{th} tag of the i^{th} word in a sentence. $\boldsymbol{A}$ is the transition matrix of the linear chain CRF, and $\boldsymbol{A}_{i,j}$ represents the score of a transition from the i^{th} tag to the j^{th} tag. y_0 and y_{n+1} is the start and stop tag of a sentence, which is common trick used by natural language processing. Thus the size of matrix $\boldsymbol{A}$ is $(k+2) \times (k+2)$. For a sequence of predictions $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$, we define its score as:

$$S(\boldsymbol{X}, \boldsymbol{y}) = \sum_{i=0}^{n} \boldsymbol{A}_{y_i, y_{i+1}} + \sum_{i=1}^{n} \boldsymbol{P}_{i, y_i} \tag{4}$$

The CRF defines the probability for the sequence of predictions $\boldsymbol{y}$ can be driven based on this score:

$$p(\boldsymbol{y}|\boldsymbol{X}) = \frac{1}{Z(\boldsymbol{X})} e^{S(\boldsymbol{X}, \boldsymbol{y})} \tag{5}$$

In this equation, $Z(\boldsymbol{X}) = \sum_{\tilde{\boldsymbol{y}} \in \boldsymbol{Y_X}} e^{S(\boldsymbol{X}, \tilde{\boldsymbol{y}})}$ is the partition function, and $\boldsymbol{Y_X}$ contains all possible tag sequences for the input sentence. During training, we maximize the following log-likelihood of the correct tag sequence:

$$\log p(\boldsymbol{y}|\boldsymbol{X}) = S(\boldsymbol{X}, \boldsymbol{y}) - \log Z(\boldsymbol{X}) \tag{6}$$

The calculation of $Z(\boldsymbol{X})$ is the key to maximize the log-likelihood function. We use the forward-backward algorithm to calculate it.

In the forward-backward algorithm, we define forward variables α and backward variables β as follows:

$$\boldsymbol{\alpha}_{t,j} := e^{\boldsymbol{P}_{t,j}} \sum_{\tilde{y}_{t-1}} e^{\boldsymbol{A}_{\tilde{y}_{t-1}, j}} \boldsymbol{\alpha}_{t-1, \tilde{y}_{t-1}} \tag{7}$$

$$\boldsymbol{\beta}_{t,j} = \sum_{\tilde{y}_{t+1}} e^{\boldsymbol{A}_{j, \tilde{y}_{t+1}} + \boldsymbol{P}_{t+1, \tilde{y}_{t+1}}} \boldsymbol{\beta}_{t+1, \tilde{y}_{t+1}} \tag{8}$$

Then the partition function $Z(\boldsymbol{X})$ and the marginal distribution can be calculated $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$:

$$Z(\boldsymbol{X}) = \boldsymbol{\alpha}_{n+1, s} \tag{9}$$

$$p(y_t|\boldsymbol{X}) = \frac{\boldsymbol{\alpha}_{t, y_t} \boldsymbol{\beta}_{t, y_t}}{\sum_{\tilde{y}_t} \boldsymbol{\alpha}_{t, \tilde{y}_t} \boldsymbol{\beta}_{t, \tilde{y}_t}} \tag{10}$$

where s is the stop tag.

2.4 Hybrid System

Embedding layer, Bi-directional LSTM layer and CRF layer are key components of this model. Two Linear + Relu layers work as pipes to transfer the data through this model. The model's structure is shown in Fig.2.

- **Embedding layer:** Each input of the embedding layer is a sentence $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$. The goal of the embedding layer is to reduce dimensions. In sentence $\boldsymbol{X}$, embedding layer convert each word x_i to a continuous vector space with much lower dimensions. In this paper, we combine word-level embedding and character-level embedding to do embedding, which we talk about in section 2.1.

- **Linear + Relu layer 1:** Using the output of embedding layer $\boldsymbol{E} = (\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n)$ as input, the linear layer projects each vector in $\boldsymbol{E}$ to fit the input size of the bi-directional LSTM layer. A Relu activation function follows this linear projection.

- **Bi-directional LSTM layer:** Using the projected embedding vectors as input, the Bi-directional LSTM layer works like section 2.2. The output of the forward and backward LSTM is concatenated to be the output. We add drop-out for this output and set drop-out ratio $p = 0.5$ (Zaremba et al., 2014).

- **Linear + Relu layer 2:** Using the output of the bi-directional LSTM $\boldsymbol{L} = (\boldsymbol{l}_1, \boldsymbol{l}_2, \ldots, \boldsymbol{l}_n)$ as input, the linear layer projects each vector in $\boldsymbol{L}$ to fit the input size of CRF layer, which is the size of tag set plus two. A Relu activation function follows this linear projection.

- **CRF layer:** Using the output of Linear + Relu layer 2 to be the $\boldsymbol{P}$ in Eq.4, we can compute the log-likelihood and marginal distribution of the given sentence, following the CRF in section 2.3.

3 Model-specific method

To fully utilize the information from LSTM-CRF and train the whole model with both labeled and unlabeled dataset, we apply the auto-encoder structure to the LSTM-CRF. It is a model-specified method, based on the work of Ammar et al. (2014). They propose the CRF auto-encoder, which regenerates the input sentences according to the marginal distribution of CRF and guarantees the capability to train the whole model with large unlabeled datasets. In this section, we introduce the CRF auto-encoder model first. Then we describe how to generalize this method to obtain the LSTM-CRF auto-encoder.

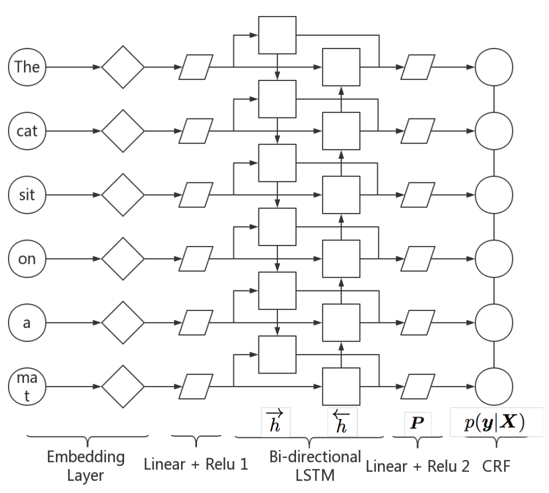

Figure 2: An LSTM-CRF

3.1 CRF Auto-encoder

The CRF auto-encoder framework consists of an encoding model and a decoding model. The encoding model is a linear-chain CRF, which is described in section 2.3. The decoding model independently generates the tokens $\hat{\boldsymbol{X}}$ conditional on the corresponding labels, using a simple categorical distribution.

$$\begin{aligned} p(\boldsymbol{y}, \hat{\boldsymbol{X}}|\boldsymbol{X}) &= \prod_{i=1}^{n} \theta_{\hat{x}_i|y_i} p(y_i|\boldsymbol{X}) \\ w.r.t \ \sum_{\hat{x}_i} \theta_{\hat{x}_i|y_i} &= 1, \ \ \text{for } i = 1, \dots, n \end{aligned} \tag{11}$$

We can use the cross entropy between $\boldsymbol{X}$ and $p(\hat{\boldsymbol{X}}|\boldsymbol{X})$ as the loss function for unlabeled datasets.

3.2 LSTM-CRF Auto-encoder

The CRF auto-encoder generates each token independently. However, this independence assumption is meaningless. Each word within a sentence should be dependent on all prior ones. Using an LSTM as decoder may be a good attempt to utilize this dependency.

We tried two different decoders in our work. The first decoder is a linear layer, as we describe in the CRF auto-encoder. The second decoder is an LSTM layer.

- Linear decoder: Given a sentence $\boldsymbol{X}$, the input of the linear decoder is the CRF layer's marginal distribution $p(\boldsymbol{y}|\boldsymbol{X})$, which is a 2-dimensional matrix of size $l \times k$, where l is the length of the input sentence. The parameters of the linear layer is θ, which is a 2-dimensional matrix of size $k \times v$. v is the size of the dictionary $\mathcal{D}$. We can compute $p(\hat{\boldsymbol{X}}|\boldsymbol{X})$ as we mentioned in section 3.1. We use the cross entropy between $p(\hat{\boldsymbol{X}}|\boldsymbol{X})$ and $\boldsymbol{X}$ as the loss function.

- LSTM decoder: We view $p(\boldsymbol{y}|\boldsymbol{X})$ as a sequence of vectors, and the size of each vector is k. Then we use it as the input of the LSTM decoder layer. The output of this LSTM layer is $\hat{\boldsymbol{E}}$, and its dimension equals to the embedding layer's output dimension. We use the mean square between $\hat{\boldsymbol{E}}$ and $\boldsymbol{E}$ as the loss function. According to Eq.3, the number of parameters in the LSTM decoder is $4e^2+4ek$, where e is the embedding dimension. Because $v \gg e$, the linear decoder and LSTM decoder almost has the same number of parameters. We reconstruct the output of embedding layer $\boldsymbol{E}$ instead of the input sentence $\boldsymbol{X}$ because the size v of the word dictionary is too large to learn. If we still reconstruct $\boldsymbol{X}$, there will be $4v^2 + 4vk$ parameters in the LSTM decoder, which is too many to learn[1] .

We use Pytorch to implement the whole model (Paszke et al., 2017). Pytorch is built on an auto differentiation system, so we can use backpropagation to train our model. In this system, the time and memory cost for training is proportional to inference (Maclaurin et al., 2015). If we can speed up the inference, then the training process will be accelerated automatically.

4 Acceleration

In this paper, we want to implement the LSTM-CRF and tune the parameters based on the Wikipedia dataset, which contains 88,083,626 unlabeled sentences (Al-Rfou et al., 2013). However, most of existing LSTM-CRF implementations do not have the ability to deal with this huge dataset. We test the speed of several implementations of LSTM-CRF (without character-level embedding), on the CoNLL-2000 dataset (8,931 sentences) (Tjong Kim Sang and Buchholz, 2000).

[1] According to the parameters we choose in section 5. If we reconstruct $\boldsymbol{X}$ by the LSTM decoder, the number decoder's parameters is 640 times more than the linear decoder's, and the training speed is unacceptable. If we reconstruct $\boldsymbol{E}$ by the LSTM decoder, the number decoder's parameters is 25% of the linear decoder's, and the training speed is almost the same as the linear decoder.

The running time of each implementations are shown in Table 1.

Implementation	Speed
Huang et al. (2015)	within 1 hour
Pytorch tutorial	28 minutes
Lample et al. (2016)	164 seconds
This work	12 seconds

Table 1: Time for training one epoch CoNLL-2000 dataset.

Training one epoch of the Wikipedia dataset needs at least 2 weeks with the implementation of (Lample et al., 2016)[2] . In this case, we need to accelerate the LSTM-CRF.

With GPU acceleration, neural net training is 10-20 times faster than with CPUs (Chetlur et al., 2014). Mini-batch gradient descent can fully utilize capacity of GPU, and provide computationally more efficient process than stochastic gradient descent. We want to implement high-speed LSTM-CRF based on mini-batch gradient descent.

Mini-batch gradient descent splits the training dataset into small batches to train the model. The length of each input data should be consistent in a given batch. However, the length of each input sentence varies in natural language processing tasks. So we need to appending multiple 0s at the end of sentence, and guarantee the length of input sentences is consistent within one batch.

We need to make sure the loss function unchanged after padding. Modern deep learning framework can do automatic differentiation (Maclaurin et al., 2015). If we keep the loss function unchanged, the parameters' derivative will either not be changed after padding.

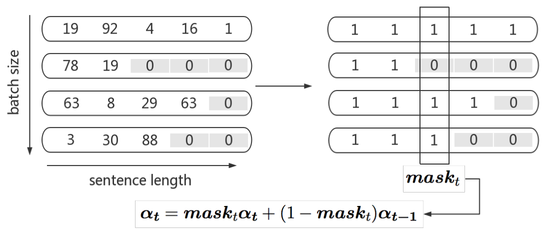

Figure 3: Mini-batch in CRF.

The loss function of LSTM-CRF is the supervised neg-log-likelihood function, which is driven from the forward variable $\boldsymbol{\alpha}_t$. For a given sentence $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)$ within one batch sentences, the calculation of $\boldsymbol{\alpha_t}$ for $\boldsymbol{X}$ should be stopped when $n < t < l$, where l is the maximum sentence length within this batch. So we create a $\boldsymbol{mask}$ matrix to fix this issue. $\boldsymbol{mask}_{i,j} = 0$ means this batch's i^{th} sentence's j^{th} index is 0.

$$\boldsymbol{\alpha_t} = \boldsymbol{mask}_t\boldsymbol{\alpha}_t + (1 - \boldsymbol{mask}_t)\boldsymbol{\alpha}_{t-1} \quad (12)$$

As shown in Fig.3, $\boldsymbol{mask}_{i,j}$ will equal to zero, if j is greater than the i^{th} sentence's length. It leads to this sentence's $\boldsymbol{\alpha}_t$ will keep unchanged when t is greater than its length.

The main difference between ours implementation and the implementation of (Lample et al., 2016) is that we use mini-batches. The speed result in table 1 shows this trick can improve the speed 13 times more.

5 Experiment

5.1 Data

We evaluate the performance of both LSTM-CRF and LSTM-CRF auto-encoder model with CoNLL-2000 dataset which contains 8,936 training sentences and 2,012 testing sentences (Tjong Kim Sang and Buchholz, 2000). The Wikipedia dataset is an unlabeled dataset, which contains 88,083,626 sentences (6 billion tokens) (Al-Rfou et al., 2013). We use the Wikipedia dataset to tune the embedding layer's parameters. We also use parts of Wikipedia datasets to train the GloVe word representations.

5.2 Training procedure

We use block coordinate descent to optimize our model, which is described in (Lin et al., 2014), as the following process:

- Supervised train the LSTM-CRF on CoNLL-2000 datasets until it overfits, by maximizing the log-likelihood of the CRF layer in Eq.6. In this way, we initialize the parameters of LSTM and CRF layer;
- Update the parameters of the embedding layer and decoder layer with the unlabeled wikipedia datasets, by minimizing the loss function we defined in the LSTM-CRF auto-encoder part;
- Supervised train the LSTM-CRF again to get the final accuracy.

[2]It is implemented in Theano (Bastien et al., 2012).

5.3 Optimizer

We choose stochastic gradient descent (SGD) method to train our model and update parameters on every training batch. We also use early-stop to train our model. We decrease the learning rate 10 times, whenever the network outperforms the previous best model twice. According to the result, SGD with the 0.1 initial learning rate and gradient clipping to 5.0 achieves the highest accuracy (Pascanu et al., 2013). We try several other gradient methods, such as SGD, Adadelta, Adam and RMSProp (Zeiler, 2012; Kingma and Ba, 2014; Tieleman and Hinton, 2012). Although other gradient methods may achieve faster convergence speed, none of their accuracy is as good as SGD.

5.4 Spelling features

We do not use the POS feature in CoNLL-2000 datasets. Apart from replacing every digit with a zero and using the lower case of every word, we extract the following features for a given word.

- whether it has all lower case letters
- whether it has all upper case letters
- whether it starts with a capital letter
- whether it has any capital letter except the first one
- whether it mixes letters and digits
- whether it contains no letters
- whether it contains no digits
- whether it has all special characters

5.5 Choice of model

Parameter	Size
Vocab Size	8000
Word-level Embedding Dimension	50
Character Size	25
Character-level Embedding Dimension	25
LSTM Input Dimension	150
LSTM Hidden Dimension	150
Initial Learning Rate	0.1
Gradient Clipping Norm	5.0

Table 2: Model's hyper-parameters

In Table 2, we list our hyper-parameters. We tried different vocabulary sizes, from 6,000 to 17,000 (17,000 is about the total number of distinct words in CoNLL-2000), and found 8,000 achieves the best performance in supervised training. We set the word-level embedding dimension equal to 50, the same as the SENNA embedding dimension (Collobert et al., 2011). We chose the same character size and character-level embedding dimension as (Lample et al., 2016). We tried several LSTMs with different number of layers and hidden size, and their performances are almost the same.

We also test another supervised loss function of CRF, which is an approximation of a Hamming loss function (Hamming, 1950), and seeks to maximize the per-label accuracy of the prediction (Gross et al., 2006). However, this loss function's performance is not as good as the supervised negative log-likelihood function. To get corresponding tags from the CRF layer, we evaluate the performance of the marginal decoder and Viterbi decoder (Forney Jr, 2005). Their accuracy performance is almost the same, and Viterbi decoder is twice faster than the marginal decoder. So in this work, we choose the negative log-likelihood as the supervised loss function for the LSTM-CRF, and use the Viterbi decoder. We also test other training methods, such as update all the parameters when doing unsupervised training or first do supervised learning then do unsupervised learning alternately. None of them is better than our method.

Model	F1 score
w.e.+random	89.82
w.e.+random+dropout	90.54
w.e.+s.f.+random+dropout	91.71
w.e.+ s.f.+c.e.+random+dropout	92.00
w.e.+s.f.+c.e.+GloVe+dropout	93.80
w.e.+s.f.+c.e.+SENNA+dropout	94.37
w.e.+s.f.+c.e.+linear+dropout	92.09
w.e.+s.f.+c.e.+linear+s.s+dropout	92.90
w.e.+s.f+c.e.+LSTM+dropout	91.94
(Huang et al., 2015)+random	94.13
(Huang et al., 2015)+SENNA	94.46
(Yang et al., 2016)	94.66
(Collobert et al., 2011)	94.32

Table 3: w.e, s.f.,s.s. and c.e. means word-level embedding, spell features, sub-sampling and character-level embedding, respectively. Glove, SENNA and random points out how to initialize the word-level embedding layer. Linear and LSTM shows the type of decoder that used by us.

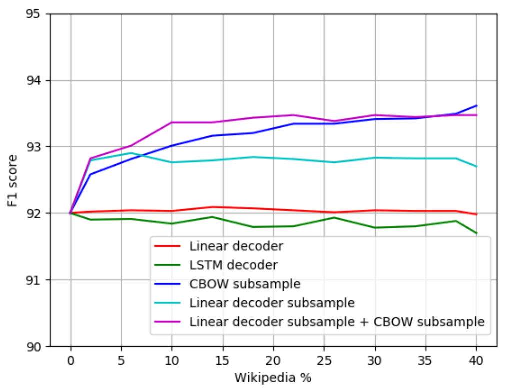

Figure 4: Comparison between different semi-supervised methods.

The performance of our model is shown in Table 3 and Fig.4. In Table 3, we show the F1 scores of our model according to different initialization methods and different decoders. In Fig.4, we compare the performance between different semi-supervised methods. The x-axis of this figure shows the size of unlabeled dataset that we used.

According to Table 3 and Fig.4, the character-level embedding and spelling features can improve the performance a lot. The auto-encoder without subsampling cannot improve the performance of LSTM-CRF, which is the same as the result of Ammar et al. (2014). Although, the performance of auto-encoder becomes much better with subsampling, it still can not beat the model-agnostic method, which uses pre-trained word representation to initialize the word-level embedding layer. Enlarging the unsupervised training in the LSTM-CRF auto-encoder seems useless. The performance of the linear decoder is better than the LSTM decoder. In future work, we hope to explore whether the combination of a language-modeling loss function (akin to those used by SENNA) with our LSTM decoder losses could yield even better semi-supervised learners than either one in isolation.

In Table 3, we also compare the F1 with other works. (Huang et al., 2015) achieve unexpectedly high F1 when the embeddings are randomly initialized because they use lots of spelling features and context features. (Yang et al., 2016) get the best performance among these four works.

6 Conclusions

In this paper, we propose a new variance of the LSTM-CRF to handle sequence tagging task. We apply the auto-encoder structure to improve the performance of the LSTM-CRF. It is shown that our embedding layer can improve the performance slightly, even though it still can not beat the performance of using pre-trained word representation, such as GloVe and SENNA. We have also shown the performance of the LSTM-CRF auto-encoder will not improve, with larger unlabeled datasets. We also show how to apply mini-batch gradient descent to fully utilize GPU resource. The speed of our implementation is as least 13 times faster than other existed implementations.

References

Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. 2013. Polyglot: Distributed word representations for multilingual nlp. arXiv preprint arXiv:1307.1662 .

Waleed Ammar, Chris Dyer, and Noah A Smith. 2014. Conditional random field autoencoders for unsupervised structured prediction. In Advances in Neural Information Processing Systems. pages 3311–3319.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2016. A simple but tough-to-beat baseline for sentence embeddings .

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. 2012. Theano: new features and speed improvements. arXiv preprint arXiv:1211.5590 .

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks 5(2):157–166.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 .

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

Sean R Eddy. 1996. Hidden markov models. Current opinion in structural biology 6(3):361–365.

G David Forney Jr. 2005. The viterbi algorithm: A personal history. arXiv preprint cs/0504020 .

Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural Networks 18(5):602–610.

Samuel S Gross, Olga Russakovsky, Chuong B Do, and Serafim Batzoglou. 2006. Training conditional random fields for maximum labelwise accuracy. In NIPS. pages 529–536.

Richard W Hamming. 1950. Error detecting and error correcting codes. Bell Labs Technical Journal 29(2):147–160.

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

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

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 .

John Lafferty, Andrew McCallum, Fernando Pereira, et al. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Proceedings of the eighteenth international conference on machine learning, ICML. volume 1, pages 282–289.

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

Chu-Cheng Lin, Waleed Ammar, Lori Levin, and Chris Dyer. 2014. The cmu submission for the shared task on language identification in code-switched data. In Proceedings of the First Workshop on Computational Approaches to Code Switching. pages 80–86.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2015. Autograd: Effortless gradients in numpy. In ICML 2015 AutoML Workshop.

Andrew McCallum, Dayne Freitag, and Fernando CN Pereira. 2000. Maximum entropy markov models for information extraction and segmentation. In Icml. volume 17, pages 591–598.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems. pages 3111–3119.

Christopher Olah. 2015. Understanding lstm networks. GITHUB blog, posted on August 27:2015.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In International Conference on Machine Learning. pages 1310–1318.

Adam Paszke, Sam Gross, and Soumith Chintala. 2017. Pytorch.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In Empirical Methods in Natural Language Processing (EMNLP). pages 1532–1543. http://www.aclweb.org/anthology/D14-1162.

Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning 4(2):26–31.

Erik F Tjong Kim Sang and Sabine Buchholz. 2000. 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. Association for Computational Linguistics, pages 127–132.

Lyan Verwimp, Joris Pelemans, Patrick Wambacq, et al. 2017. Character-word lstm language models. arXiv preprint arXiv:1704.02813 .

Zhilin Yang, Ruslan Salakhutdinov, and William Cohen. 2016. Multi-task cross-lingual sequence tagging from scratch. arXiv preprint arXiv:1603.06270 .

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. arXiv preprint arXiv:1409.2329 .

Matthew D Zeiler. 2012. Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 .