Deep learning methods have received increasing research attention in natural language processing (NLP), with neural models being built for classification (Kalchbrenner et al., 2014), sequence labeling (Collobert et al., 2011), parsing (Socher et al., 2013; Dyer et al., 2015; Zhou et al., 2015; Weiss et al., 2015), machine translation (Cho et al., 2014), fine-grained sentiment analysis (Zhang et al., 2015) and other tasks. This surge of the interest gives rise to a demand of software libraries, which can facilitate research by allowing fast prototyping and modeling for experimentation.

For traditional methods such as conditional random fields (CRF) (Lafferty et al., 2001) and SVM (Vapnik, 1995), there has been various software toolkits, implemented in different programming languages, including Java, Python and C++. These toolkits offer a large degree of variety for building NLP models by using or adapting the machine learning algorithms. For deep learning, a number of software tools have been developed, including Theano\(^1\) (Bergstra et al., 2010), caffe\(^2\) (Jia et al., 2014), CNN\(^3\), torch\(^4\) etc. These tools are based on different programming languages and design concepts. On the other hand, most of these libraries are not designed specifically for NLP tasks. In addition, many existing libraries define a complex class hierarchy, making it difficult for some users to use or adapt the modules.

We present another deep learning toolkit in C++, designed specifically for NLP applications. The main objective is to make it extremely light-weight, so as to minimize the effort in building a neural model. We take a layered approach, offering high-level models for classification and sequence labeling, such as neural CRF (Do et al., 2010), recurrent neural networks (RNN) (Graves, 2012) and long-short-term memories (LSTM) (Hochreiter and Schmidhuber, 1997), which are frequently used in NLP. On the other hand, we minimize encapsulation, implementing neural structures strictly abiding by their formal definitions, so as to make it easy to work directly with neural layers and facilitate extensions to existing network structures.

Our design is centralized in the structure of a neural layer, which performs the standard feed-forward function and back-propagation. We provide a wide range of built-in neural activation functions, and common operations such as concatenation, pooling, window function and embedding lookup, which are needed by most NLP tasks. We support flexible objective functions and optimization methods, such as max-margin, max likelihood criterions and AdaGrad (Duchi et al., 2011), and also verification functions such as gradient check. One uniqueness of our toolkit is the support of both dense continuous features and sparse indicator features in neural layers, making it convenient also to build traditional discrete models such as the perceptron, logistic regression and CRF, and to combine discrete and continuous features (Ma et al., 2014; Durrett and Klein, 2015; Zhang and Zhang, 2015).

Taking word segmentation, POS-tagging and name entity recognition (NER) as typical examples, we show how state-of-the-art discrete, neural and hybrid models can be built using our toolkit. For example, we show how a bidirectional LSTM model can be built for POS tagging in only 23-lines (12 for inference and 11 for back-propagation) of codes, which gives highly competitive accuracies on standard benchmarks.

\(^1\)https://github.com/Theano/Theano
\(^2\)http://caffe.berkeleyvision.org/
\(^3\)https://github.com/clab/cnn
\(^4\)http://torch.ch/
Atomic Layers

| Dense   | uni-layer: $y = f(Wx + b)$  
|         | bi-layer: $y = f(W_1x_1 + W_2x_2 + b)$  
|         | tri-layer: $y = f(W_1x_1 + W_2x_2 + W_3x_3 + b)$  
|         | tensor-layer: $y = f(x_1T_1x_2 + b)$  
| Discrete| uni-layer: $y = f(Wx)$  

Pooling

$y = \sum_{i=1}^{n} \alpha_i \odot x_i$

- max: $\alpha_{i,j} = 1$, when $i = \arg \max_j (x_{i,j})$, otherwise 0;
- min: $\alpha_{i,j} = 1$, when $i = \arg \min_j (x_{i,j})$, otherwise 0;
- average: $\alpha_{i,j} = \frac{1}{n}$;
- sum: $\alpha_{i,j} = 1$.

Loss Function

**Classifier**

- max entropy (MAXENT): $o, y \rightarrow \partial o$
  - $\text{loss}(o) = -y \log \text{softmax}(o)$;
  - $\partial o = \frac{\partial \text{loss}(o)}{\partial o}$

**Structural Learning**

- CRF, max likelihood (CRFML): $o^1, y_i^1 \rightarrow \partial o_i^1$
  - $\text{loss}(o_i^1) = -\log p(y_i^1|o_i^1)$, where $p(\cdot)$ can be computed via the forward-backward algorithm (Sutton and McCallum, 2007);  
  - $\partial o_i^1 = \frac{\partial \text{loss}(o_i^1)}{\partial o_i^1}$

- CRF, max margin (CRFMM): $o_i^1, y_i^1 \rightarrow \partial o_i^1$
  - $\text{loss}(o_i^1) = \max_{s,j}(s(y_i^1) + \delta(y_i^1, y_i^1)) - s(y_i^1)$, where $\tilde{y_i^1}$ is an answer sequence with one label for each position;
  - $\partial o_i^1 = \frac{\partial \text{loss}(o_i^1)}{\partial o_i^1}$

**Others**

- LookupTable: $E$, specifying vector representations for one vocabulary.
- Concatenation: $y = x_1 \oplus x_2 \oplus \cdots \oplus x_M$
- Dropout: $y = m \odot x$, where $m$ is a mask vector.
- Window function: $x_i^w \rightarrow y_i^w$, where $y_i = x_{i-w} \oplus \cdots \oplus x_i \oplus \cdots \oplus x_{i+w}$

Table 1: Base classes.

<table>
<thead>
<tr>
<th>RNN</th>
<th>$x_i^w \rightarrow y_i^w: y_i = f(Wx_j + Uy_{j-1} + b)$</th>
</tr>
</thead>
</table>
| GRNN| $x_i^w \rightarrow y_i^w$, where $y_i$ is computed by:
| LSTM| $i_j = \sigma(W_i x_j + U_i y_{j+1} + b_i)$
|     | $f_j = f(W_f x_j + U_f r_j \odot y_{j+1} + b_f)$
|     | $z_j = \sigma(W_z x_j + U_z y_{j+1} + b_z)$
|     | $y_j = (I - z_j) \odot y_{j+1} + z_j \odot y_j$

| LSTM| $i_j = \sigma(W_i x_j + U_i y_{j+1} + V_i c_{j+1} + b_i)$
|     | $f_j = \sigma(W_f x_j + U_f y_{j+1} + V_f c_{j+1} + b_f)$
|     | $c_j = f(W_c x_j + U_c y_{j+1} + b_c)$
|     | $o_j = \sigma(W_o x_j + U_o y_{j+1} + V_o c_j + b_o)$
|     | $y_j = o_j \odot f(c_j)$

| Attention Model | $x_i^w, a_i^w \rightarrow y$, where $y$ is computed by:
|----------------|---------------------------------|
|                | $h_j = f(W_j x_j + U a_j + b)$
|                | $a_j = \exp(h_j)$
|                | $z = \sum_{j=1}^{n} a_j$
|                | $y = \sum_{j=1}^{n} \frac{a_j}{z}$

Table 2: Classes of neural network structures.

1 Classes

1.1 Base Layers

Shown in Table 1, we provide several basic classes, which are widely used in neural networks and discrete machine learning algorithms, including atomic layers, pooling functions, loss functions and others. All classes have three interfaces, one for obtaining forward outputs, one for computing backward losses, and the last for update parameters.

**Neural Layers** The neural layers are single atomic layers used in neural networks, which support one, two or three input vectors. In Table 1, $f$ can be any activation function, such as the simple $id$ operation or non-linear functions including tanh, sigmoid and exp. For discrete features, we support only one vector input. A logistic regression classifier can be built using one single discrete layer.

**Pooling** Pooling functions are widely used to obtain fixed-dimensional output from sequential vectors of variable lengths. Commonly-used pooling techniques include max, min and averaged function. We implement sum pooling also.

**Loss Function** We offer three different loss functions, one for classification, based on the max-entropy
To facilitate model building, we provide some useful classes such as lookup table, drop out, concatenation and window feature extraction. These functions are all shown in Table 1.

1.2 Network structures

Using basic classes, one can build advanced neural network structures in the literature. In this package, we implement four different neural networks, including a simple recurrent neural network (RNN), a gated recurrent neural network (GRNN), a long-short term memory neural network (LSTM) and an attention model. Their definitions are given in Table 2.

2 Evaluation

We show how to apply the package to building neural network models for Chinese word segmentation, POS tagging and NER. All three tasks are formalized as sequence labeling problems. The general framework is shown in Figure 1, where we collect input vectors ($t^n_i$) at the bottom for each word, and then add a windowlized layer to exploit surrounding information, obtaining $x^n_i$. Then, we apply two LSTM neural networks, one being computed from left to right $lh^n_i$ and the other being computed from right to left $rh^n_i$. These two kinds of features are combined using a non-linear combination layer, giving $h^n_i$. Finally, we compute output vectors $o^n_i$, scoring different labels at each position.

During training, we run standard back-propagation. We choose CRF max-margin loss to compute the
Then step by step, we compute the losses of $h^n_i$, $lh^n_i$, $rh^n_i$, $p^n_i$ and $\ell^n_i$, aggregating losses for each parameter at each layer. Finally, we use Adagrad to update parameters for all layers.

Between segmentation, POS tagging and NER, the differences lie mainly in the input vectors $t^n_i$. For Chinese word segmentation, we use the concatenation of character unigram embeddings $E_{c_1}$ and bigram embeddings $E_{c_1-1}$ at each position as the input vector $t_i$. The character unigram and bigram embeddings are pretrained separately. For POS tagging, $t_i$ consists of embedding $E_{w_i}$ of the word $w_i$ and its vector representation $v_{c_i}$ derived from its character sequence $c_1^{m_i}$ ($m_i$ is the length of word $w_i$). $v_{c_i}$ is constructed according to neural network structures shown in Figure 2. For NER, $t_i$ consists of three parts, including $E_{w_i}$, $v_{c_i}$ and the word’s POS tag embedding $E_{p_i}$. The deep neural POS tagging model consists of only 23 lines of code, as marked by red superscripts in Table 3, Figure 2 and Figure 1.

Besides the neural models above, we also implement discrete models for the three tasks. The discrete features are extracted according to Liu et al. (2014), Toutanova et al. (2003) and Che et al. (2013) for word segmentation, POS tagging and NER, respectively. We simply apply the sparse atomic layer and exploit the same CRF max-margin for training model parameters. Finally, we make combinations of the discrete and neural models by aggregating their output vectors.

**Results.** We conduct experiments on several datasets. For Chinese word segmentation, we exploit PKU, MSR and CTB60 datasets, where the training and testing corpus of PKU and MSR can be downloaded from BakeOff2005 website\(^3\). For POS tagging, we perform experiments on both English and Chinese datasets. For English, we follow Toutanova et al. (2003), using WSJ sections of 0-18 as the training dataset, section 19-21 as the development corpus and section 22-24 as the testing dataset. For Chinese, we use the same data set as Li et al. (2015). For NER, we follow Che et al. (2013) to split Ontonotes 4.0 to get the English and Chinese datasets.

Our experimental results are shown in Table 4. As can be seen from the table, our neural models give competitive results compared the state-of-the-art results on each task, which are Zhang and Clark (2007) for Chinese word segmentation, Toutanova et al. (2003) for English POS tagging, Li et al. (2015) for Chinese POS tagging and Che et al. (2013) for English and Chinese NER.

### 3 Code

Our code and examples in this paper is available under GPL at https://github.com/SUTDNLPI, including repositories of **LibN3L, NNSegmentation, NNPSTagging and NNNamedEntity**.
References


