Plug-Tagger: A Pluggable Sequence Labeling Framework with Pre-trained Language Models

Anonymous ACL submission

Abstract

Fine-tuning the pre-trained language models (PLMs) on downstream tasks is the de-facto paradigm in NLP. Despite the superior performance on sequence labeling, the fine-tuning requires large-scale parameters and time-consuming deployment for each task, which limits its application in real-world scenarios. To alleviate these problems, we propose a pluggable sequence labeling framework, plug-tagger. By switching the task-specific plugin on the input, plug-tagger allows a frozen PLM to perform different sequence labeling tasks without redeployment. Specifically, the plugin on the input are a few continuous vectors, which manipulates the PLM without modifying its parameters, and each task only needs to store the lightweight vectors rather than a full copy of PLM. To avoid redeployment, we propose the label word mechanism, which reuses the language model head to prevent task-specific classifiers from modifying model structures. Experimental results on three sequence labeling tasks show that the proposed method achieves comparable performance with fine-tuning by using 0.1% task-specific parameters. Experiments show that our method is faster than other lightweight methods under limited computational resources

1 Introduction

Pre-trained language models (PLMs) (Devlin et al., 2019; Peters et al., 2018; Radford et al., 2019), which are trained on a huge amount of data to learn universal language representations, have been shown to be beneficial for improving many natural language processing (NLP) tasks. In addition to the performance of PLM, as the size of PLMs grows (eg. GPT3 (Brown et al., 2020) has 175B parameters), there has been an increasing interest in efficiently transferring the PLM to downstream tasks.

The past few years have witnessed the prevailing of fine-tuning the PLM on downstream NLP tasks.

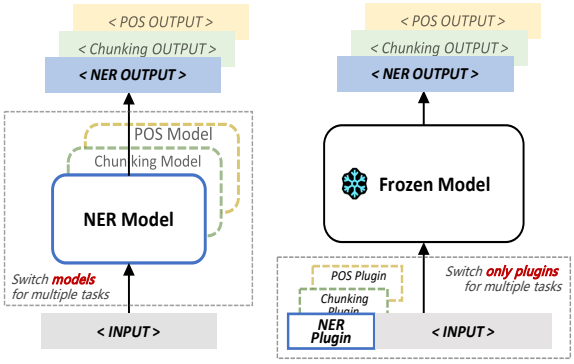

Figure 1: Comparison of fine-tuning (left) and pluggable method (right). When performing different tasks, fine-tuning needs to redeploy the model, which leads to huge memory demand and time cost. The pluggable method only needs to switch the lightweight plugin vectors on input without redeploying the model.

However, fine-tuning requires training an entirely new model for every new task. The large-scale parameters of the PLM make it expensive to keep a copy of parameters for each task, and deploying models for a large number of tasks dramatically increases the cost of time (Sharma et al., 2018), as shown in Figure 1. A more idealistic way to transfer PLM is the pluggable method, which uses the lightweight pluggable module to manipulate the frozen model to output desired response without redeployment.

There are two conditions for achieving pluggability. One is the lightweight pluggable module, which manipulates PLM behavior without modifying its parameters. The other is an adaptive classifier that can perform different tasks without modifying the model structure. Currently, there are two types of attempts to reach this goal. Adapter-tuning (Rebuffi et al., 2017; Houlsby et al., 2019; Raffel et al., 2020) optimizes lightweight modules called adapters between each layer of PLM and only the adapters are stored for each task. Despite the lightweight, adapter-tuning does not satisfy the conditions for the adaptive classifier,

which inevitably leads to redeployment. Prefix-tuning (Li and Liang, 2021; Lester et al., 2021), which prepends several continuous vectors to the input of each layer in the PLM, is proposed as a pluggable method for natural language generation. Different generation tasks can share the structure of the language model head, thus prefix-tuning can be adapted to different generation tasks without redeployment. However, the different label space makes it impossible for sequence labeling tasks to share a classifier. As a result, it is difficult to use existing solutions to achieve pluggability in sequence labeling tasks.

In this paper, we propose plug-tagger, a pluggable sequence labeling framework. Specifically, to solve the problem of pluggable modules, we insert plugin vectors to the input to manipulate the PLM without modifying its parameters. As for the adaptive classifier, we reformulate sequence labeling as a special language modeling task to reuse the language model head. The PLM with the language model head predicts the label word at each position in the sentence, the label of each position is determined by the label word rather than the task-specific classifier. Label word is a label-related word collected from vocabulary, we take high-frequency words predicted by PLM to serve as the label words for corresponding labels. Benefiting from the reuse of the entire architecture of PLMs, our method can be adapted to different tasks without redeployment.[1]

The main contributions of this paper can be summarized as follows:

- We reformulate the sequence labeling task as a label word prediction task by reusing the language model head of PLM.
- We proposed a pluggable sequence labeling framework, which leverages lightweight pluggable modules to manipulate the model behavior without redeployment.
- Experiments on a variety of sequence labeling tasks demonstrate the effectiveness of our approach. Besides, in experiments with limited computational resources, our method is faster than other lightweight methods.

2 Approach

In this work, we propose a lightweight and pluggable sequence labeling framework, which aims to make a deployed PLM perform different sequence labeling tasks without redeployment. In this section, we first introduce the key challenges of achieving pluggability on sequence labeling. Next, we propose an overview of our approach, and finally, we detail the two primary components of our approach.

2.1 Problem Statement

Given a sequence of words $\mathbf{X} = [x_1, ..., x_n]$, the goal of sequence labeling is to predict the gold labels $\mathbf{Y} = [y_1, y_2, ..., y_n]$ with equal length. The predictions of a sequence labeling system can be expressed as $\hat{\mathbf{Y}} = F(\mathbf{X}; \mathbf{\Theta})$ where $\mathbf{\Theta}$ is the parameters of the system. The traditional system based on fine-tuning and classifier can be decomposed into the following equations:

$$\begin{aligned} \mathbf{H} &= Encoder(\mathbf{X}; \phi) \\ \hat{\mathbf{Y}} &= argmax(Softmax(\mathbf{HW} + \mathbf{b})) \end{aligned} \tag{1}$$

where $\mathbf{W} \in \mathbb{R}^{h \times l}$, l is the size of label set, h is the dimension of hidden state. $Encoder$ is a PLM without language model head, ϕ is its parameters. Parameters of system $\mathbf{\Theta}$ is decomposed into ϕ, $\mathbf{W}$ and $\mathbf{b}$.

There are two challenges to be lightweight and pluggable. The first challenge arises from the large-scale ϕ, standard fine-tuning requires an entire new model for every task. That is, each task requires a large number of task-specific parameters when using PLM. The second one is that the dimension of $\mathbf{W}$ cannot be frozen due to different label sets. For example, NER's label consists of entity types such as person, location and organization, but POS's label consists of part of speech types like adjective, noun and adverb. It's challenging to map so many different task-specific labels onto the same label space. This prevents the model's classification layer from remaining frozen. Apart from this, even if we find a way to reduce the scale of task-specific parameters, the classifier still introduces a non-negligible number of parameters when the number of labels is large.

2.2 Model Overview

The architecture of the proposed model is shown in Figure 2. We switch the lightweight plugin vectors on the input rather than reloading large-scale parameters of PLM to perform different tasks. The label word mechanism replaces the task-specific classifier to avoid the modification of the model architecture. Under the influence of the

[1] URL of codes is omitted here pending the review process.

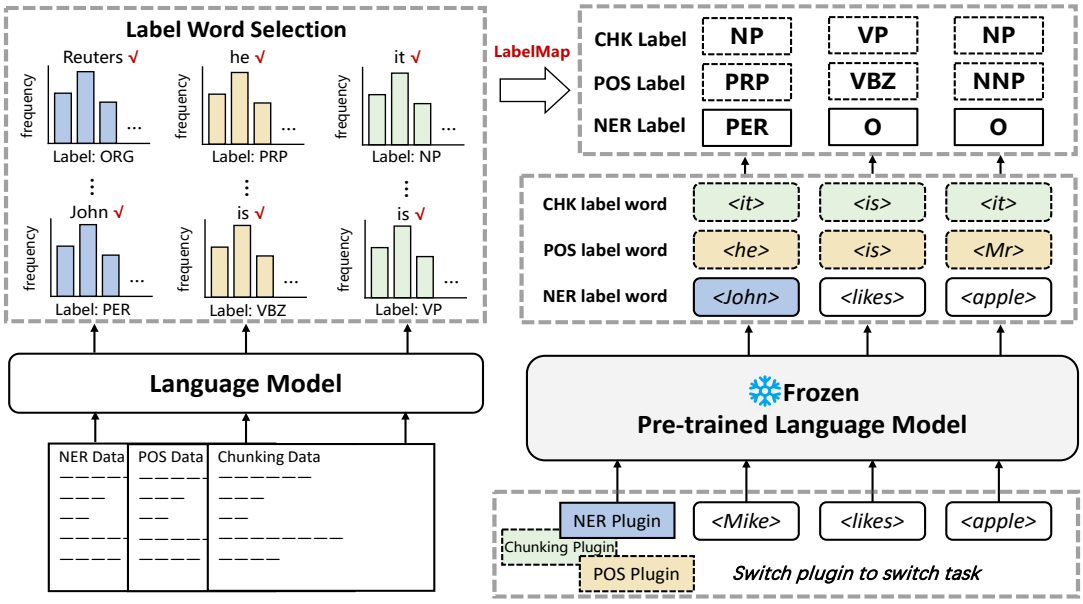

Figure 2: An overview of Plug-Tagger. Influenced by the task-specific plugin vectors on the input, the frozen language model predicts the label word for each word in the sentence. The actual label is obtained by label word mapping. The left side shows how we get the label word: the language model traverses a large amount of data. The word that related to a particular label is selected as a label word according to its frequency predicted by language model.

plugin vector, the model predicts the corresponding label words of input, and the actual labels can be obtained by label word mapping. Take NER as an example, we feed the input "*Olivia likes apple*" with the plugin of NER into the frozen language model, the output of language model will be "*John likes apple*". "*John*" is the label word of PER (person) in NER, after label word mapping, we get the NER label of the input: "PER O O".

We define the label word map as M, the parameters of the frozen language model as $\mathbf{\Theta_{lm}}$ and plugin vectors as $\mathbf{\Theta_P}$. The label words predicted by PLM can be describe as $\tilde{\mathbf{Y}} = F(\{\mathbf{X}, \mathbf{\Theta_P}\}; \mathbf{\Theta_{lm}})$ where $\{\mathbf{X}, \mathbf{\Theta_P}\}$ is the inputs, and we use label word mapping to get the real labels $\hat{\mathbf{Y}} = M(\tilde{\mathbf{Y}})$. The following two sections detail essential parts of the plug-tagger: plugin vector and label word mechanism.

2.3 Plugin Vector

To solve the problem of storing and reloading large-scale parameters for various tasks, we draw inspiration from continuous prompts. We use plugin vector to control the model behavior without modifying the architecture and parameters of the model. The plugin vector $\mathbf{\Theta_P}$ consists of a few continuous vectors and is combined with input. In the following two subsections, we show two ways to combine the plugin vector with the input.

2.3.1 Plugin in Embedding

The input of PLM is the text $\mathbf{X}$ processed by embedding, which can be described as $\mathbf{X} = [\mathbf{x_1}, ..., \mathbf{x_n}]$ where $\mathbf{x_i} = Emb(x_i)$. We insert the plugin vectors $\mathbf{\Theta_P} = [\theta_\mathbf{1}, ...\theta_{\mathbf{l_p}}]$ into the input $\mathbf{X}$ directly, the information in Θ_P flows through each layer and ultimately affects the predictions. The new input can be described as follow:

$$\mathbf{X}' = [\mathbf{\Theta_P}; \mathbf{x}_1, ..., \mathbf{x}_n], \tag{2}$$

where $\mathbf{\Theta_P} \in \mathbb{R}^{l_p} \times h$, l_p is the length of the plugin vectors, h is the dimension of embedding, [;] means concatenation in the first dimension.

2.3.2 Plugin in Layers

The plugin vectors in embedding are not expressive enough, which leads to unsatisfactory performance. To extend the influence of plugin vectors, we insert them into inputs at each layer of the model. Given a PLM with l transformer layers, the input of j^{th} layer can be described as $\mathbf{X}^{(j)} = [\mathbf{x}_1^{(j)}, ..., \mathbf{x}_n^{(j)}]$ where $\mathbf{X}^{(j)} \in \mathbb{R}^{n \times d}$, d is the dimension of hidden state and and n is the length of inputs. Transformer (Vaswani et al., 2017) layers are structured around the use of query-key-value (QKV) attention, which is calculated as:

$$
\begin{aligned}
Att(\mathbf{X}) &= Softmax(\frac{Q(\mathbf{X})K(\mathbf{X})^T}{\sqrt{d_k}})V(\mathbf{X}) \\
Q(\mathbf{X}) &= \mathbf{W}_q^{(j)}\mathbf{X} \\
K(\mathbf{X}) &= \mathbf{W}_k^{(j)}\mathbf{X} \\
V(\mathbf{X}) &= \mathbf{W}_v^{(j)}\mathbf{X},
\end{aligned} \tag{3}
$$

where $\mathbf{W_k}^{(j)}, \mathbf{W_v}^{(j)}, \mathbf{W_q}^{(j)} \in \mathbb{R}^{d\times d}$ and d_k is the number of multi head. In order to avoid adding additional layers of PLM, we combine the plugin vector with K and V, which can be describe as:

$$
\begin{aligned}
\mathbf{X}' &= \{\theta_k^{(j)}, \theta_v^{(j)}, \mathbf{X}\} \\
Att(\mathbf{X}') &= Softmax(\frac{Q(\mathbf{X})[\theta_{\mathbf{k}}^{(\mathbf{j})}; K(\mathbf{X})]^T}{\sqrt{d_k}})[\theta_{\mathbf{v}}^{(\mathbf{j})}; V(\mathbf{X})],
\end{aligned} \tag{4}
$$

where $\theta_{\mathbf{k}}^{(\mathbf{j})}, \theta_{\mathbf{v}}^{(\mathbf{j})} \in \mathbb{R}^{l_p\times d}$, l_p is the length of the plugin vectors and d is the dimension of hidden state. Plugin vectors on all layers can be represented as $\mathbf{\Theta_P} = \{(\theta_{\mathbf{k}}^{(\mathbf{1})}, \theta_{\mathbf{v}}^{(\mathbf{1})})..., (\theta_{\mathbf{k}}^{(\mathbf{l})}, \theta_{\mathbf{v}}^{(\mathbf{l})})\}$. Thus, we extend the influence of plugin vectors to every layer without modifying parameters of PLM.

2.4 Label Word Mechanism

To alleviate problems caused by the task-specific classifier, we propose the label word mechanism, which reformulates the sequence labeling to the label word prediction.

2.4.1 Label Word Selection

Algorithm 1 represents the entire label word selection processing. For each label $c \in L$, a dictionary $freq_c$ is built to counts its candidate label words and corresponding frequency. We the traverse training set, for each word x in the sentence, we use the language model to get top-k high-probability candidate words and update dictionary $freq_c$ where c is the label of the word x. After traversing the training set, we filter some words that are not suitable such as the words that frequently occur in all $freq$. Under the condition that the label word of each label is not the same, the remaining word with the highest frequency in $freq_c$ is selected as label word of label c.

In particular, for tasks that need to use the BIO schema, two special treatments are needed: 1) We don't count label word for label O. It's hard to pick a representative word for the others category. In the training and inference phase, the word with label O predicts itself. 2) We look for label words respectively for B and I of the same category, because distinguishing BI is beneficial for tasks that require boundary information. In the experiment section, we will discuss the influence of the label word mechanism on the performance in detail.

Algorithm 1 Label Word Selection

Input: Train set $D = \{X_i, Y_i\}_{i=1}^N$; Label set $L = \{c_i\}_{i=1}^l$; Vocabulary $V = \{w_i\}_{i=1}^v$; Pre-trained language model LM; Maximum candidates of label word k.

Output: LabelMap M

```
Initialize label map M = ∅;
for c ∈ L do
    Initialize freq_c = {w_i : 0}_{i=1}^v;
    Add label word pair {c : None} to M;
end for
for (X = {x_i}_{i=1}^n, Y = {y_i}_{i=1}^n) ∈ D do
    Select top-k candidate words {ỹ_i}_{i=1}^n where ỹ_i ∈ R^k based on predictions of language model LM(X);
    for i ∈ [1...n] do
        Update the frequency of label c = y_i;
        freq_c[w] ← freq_c[w] + 1 for w ∈ ỹ_i;
    end for
end for
for c ∈ L do
    Filter out irrelevant words in freq_c;
    while M[c] is None do
        Select the word w in freq_c with the highest frequency;
        if w not used by M then
            M[c] ← w;
        else
            Remove w from freq_c;
        end if
    end while
end for
return M
```

2.4.2 Training Objective

We reformulate sequence labeling to a special language modeling task. After selecting label words, we get the label map M to map label set to words in vocabulary. For sequence $\mathbf{X}$, the gold label Y is reintroduced to $\widetilde{\mathbf{Y}} = [\widetilde{y}_1, ..., \widetilde{y}_n]$ where $\widetilde{y}_i = M(y_i)$. The sentence-level loss can be

described as follows:

$$Loss = -\sum_{i=1}^{N} log(P(\widetilde{\mathbf{Y_i}}|\mathbf{X_i})), \quad (5)$$

where N is the number of sentences. When combined with the plugin vectors, label words embedding and parameters of plugin vectors $\mathbf{\Theta_P}$ are the only trainable parameters. During the inference phase, we take the prediction result of the first subword of each word and find its corresponding label according to the label map.

3 Experiments

In this section, we present the experimental results to show the efficiency and pluggability of plug-tagger. To verify whether plug-tagger could adapt to different tasks, we conduct experiments on three common sequence labeling tasks: NER, POS, and chunking. As for the pluggability, we simulate scenarios that require redeployment to verify whether plug-tagger could ease the inconvenience caused by redeployment.

3.1 Datasets

CoNLL 2003 shared task (CoNLL2003) (Sang and De Meulder, 2003) is the standard benchmark dataset that provides the annotations for NER, POS and chunking, we evaluate on CoNLL2003 for all tasks. In addition, we select another representative dataset for each task, including ACE 2005 (ACE2005) [2] for NER, Wall Street Journal (WSJ) (Marcus et al., 1993) for POS and CoNLL 2000 (CoNLL2000) (Tjong Kim Sang and Buchholz, 2000) for chunking. We use the BIO2 tagging scheme for NER and chunking. We also follow the standard dataset preprocessing and split. The CoNLL2000 does not have an officially divided validation set, we use the test set as the validation set. The statistics of the datasets are summarized in Table 1.

Task	Dataset	#Train	#Dev	#Test	Class
NER	CoNLL 2003	204, 567	51, 578	46, 666	9
	ACE 2005	144, 405	35, 461	30, 508	14
POS	CoNLL 2003	204, 567	51, 578	46, 666	45
	WSJ	912, 344	131, 768	129, 654	46
Chunking	CoNLL 2003	204, 567	51, 578	46, 666	20
	CoNLL 2000	211, 727	-	47,377	22

Table 1: Statistics of the datasets on NER, POS and chunking. # means number of tokens in dataset.

[2]https://catalog.ldc.upenn.edu/LDC2006T06

3.2 Baselines

Our method is compared with recently proposed lightweight methods and the standard fine-tuning.

FT-Full optimizes the all parameters of PLM, which can show the standard performance of a PLM.

FT-Classifier keeps the most PLM parameters frozen, and only the parameters of the classifier are optimized, which is the straightforward lightweight method.

Prompt-Tuning (Lester et al., 2021; Hambardzumyan et al., 2021; Liu et al., 2021) inserts continuous vectors into the input sentence to control the model. Follow (Lester et al., 2021), the soft prompt are optimized directly. We combine this method with PLM with classifier as a variation of soft prompt.

Bitfit (Zaken et al., 2021) only optimizes the bias term of PLM, which shows good performance with small-to-medium training data.

Adapter-Tuning (Houlsby et al., 2019) is a well-known lightweight which optimizes the parameters of additional layers inserted in PLM.

Prefix-Classifier (Li and Liang, 2021) prepends continuous vectors to each layer of PLM with a task-specific classifier.

Plug-Tagger is our proposed method. In the setting of 0.1% task-specific parameters, we insert plugin to the input sequence, in the setting of 0.01% parameters, we insert plugin to the inputs of PLM's layers.

3.3 Experiment Details

Plug-tagger and all baselines are based on Roberta-base (Liu et al., 2019). The parameters and architecture of Roberta-base are reloaded directly from HuggingFace[3]. The implementation of adapter-tuning is based on Adapter-Hub[4], which combines adapter-tuning and Transformers released by (Pfeiffer et al., 2020). We control the parameters of adapter-tuning by adjusting the dimension of the adapter layers. AdamW optimizer and linear scheduler are used for all datasets, as suggested by the Hugging Face default setup. For all baselines, we keep the epoch at 10 and batch size at 16. The hyperparameters of our method are detailed in appendix A. All label words are collected from the training set. We select the best model on the validation set to evaluate the test set.

[3]https://huggingface.co/

[4]https://github.com/Adapter-Hub/adapter-transformers

Methods	L	S	P	NER (F1)		Chunking (F1)		POS (Acc.)	
				CoNLL2003	*Ace2005*	*CoNLL2003*	*CoNLL2000*	*CoNLL2003*	*WSJ*
FT-Full (Liu et al., 2019)	×	×	×	91.45	89.02	91.41	97.05	95.64	97.69
FT-Classifier (Liu et al., 2019)	✓	×	×	84.27	77.37	79.94	83.13	88.97	94.94
Prompt-Tuning (Lester et al., 2021)	✓	×	×	86.58	82.22	84.47	93.76	93.84	96.27
Bitfit (Zaken et al., 2021)	✓	×	×	89.44	81.12	88.46	93.25	92.65	97.03
Adapter-Tuning (Houlsby et al., 2019)	✓	×	×	88.89	**88.03**	88.52	94.63	93.51	97.51
Plug-Tagger (0.01%)	✓	✓	✓	87.68	82.33	84.99	94.15	94.10	97.15
Plug-Tagger (0.1%)	✓	✓	✓	**91.50**	87.71	**90.50**	**96.41**	**94.86**	**97.60**

Table 2: Experimental results on the test set for all datasets. 0.01% means the task-specific parameters are 0.01% of FT-Full. 0.1% follows the same way. **L** means the method is lightweight, **S** means the method do not modify the model structure, **P** means the method is pluggable. Bold term means the best result in the lightweight methods.

3.4 Efficiency

Tabel 2 presents the performance of all methods. With 0.1% task-specific parameters, plug-tagger almost outperforms all other lightweight methods and achieves a comparable performance with fine-tuning. Under the setting of 0.01% parameters, our method performs worse than fine-tuning, which can be seen as a trade off between parameters and performance. Next, we analyze the experimental results in detail according to the task based on the Plug-Tagger (0.1%).

NER is the most difficult task in our experiments. We find that our method performs well on CoNLL2003, but on ACE2005, both prefix-classifier and plug-tagger underperform the adapter. We deduce that the adapter's relatively complex structure would help with difficult tasks.

Chunking's label words are the most difficult to find. For example, there are so many common nouns that it's hard to find a perfect one. However, compared with fine-tuning, we only obtain the performance drops 1% in CoNLL2003, and 0.5% in CoNLL2000, and we outperform other lightweights methods. The results prove that the label word does not need to be too precise but only needs to be related to the label.

POS has the largest number of labels. The good performance in CoNLL2003 and WSJ proves that the increased number of labels does not affect the performance of the plug-tagger.

3.5 Pluggability

In the real-world scenario where an NLP system needs to perform lots of tasks, maintaining the PLM simultaneously for each task is prohibitively expensive. An alternative approach is to release resources of the old model and load the new model when switching tasks, we call this approach

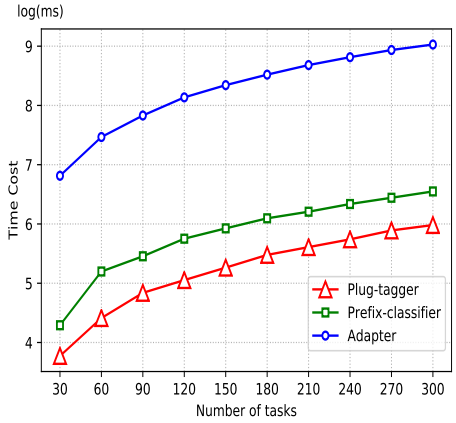

Figure 3: Time cost of preparing the model when all tasks are completed. As the discrepancy of time cost is too large, here we take the log value to better show the results.

redeployment. However, redeployment brings additional time consumption, and the larger task-specific parameters required, the more time it takes. pluggability means a method can perform new tasks without redeploying model, which reduces the cost of time.

The importance of pluggability could be justified through the redeployment time required in different approaches. We structured a task set by random sampling from three sequence labeling tasks, each sample can be treated as a new task. Non-pluggable methods need to reload the model parameter when encountering a new task, pluggable methods like plug-tagger only need to prepare the plugin vectors. Like this, through calculating the time needed in different methods required to redeployment of all tasks, the answer of if pluggability is more efficient can be clarified. We conduct our experiments on three lightweight methods: adapter, prefix-classifier, and plug-tagger. The plug-tagger here optimizes only the plugin vectors. Data is obtained

through random sampling from CoNLL2003. All experiments are conducted in the same NVIDIA GeForce RTX 1080Ti.

Experimental results are shown in Figure 3. As the discrepancy of time cost is too large, here we take the log value to better show the results. We find that despite there is not much difference between the number of parameters of adapter-tuning (119,880) and plug-tagger (92,160), the adapter-tuning takes 20 times longer to deploy than the plug-tagger. We deduce there are two reasons: 1) Adapter-tuning needs to release resources of old adapter layers before reloading the new one. 2) Adapter-tuning needs to modify the parameters of each layer. These can also be verified in the experimental results of prefix-classifier. The prefix-classifier only needs to load the classifier layer without going deep into each layer of the model, which is twice as time-consuming as the plug-tagger. This demonstrates the importance of the pluggable approach in real-world scenarios.

4 Analysis

4.1 Impact of Label Word Mechanism

Task	Dataset	Prefix-Classifier	Plug-Tagger
NER	*CoNLL2003*	90.98	91.50
	ACE2005	86.86	87.68
Chunking	*CoNLL2003*	89.53	90.50
	CoNLL2000	95.78	96.41
POS	*CoNLL2003*	94.59	94.86
	WSJ	97.44	97.60

Table 3: Performance comparison of Plug-Tagger and Prefix-Classifier. The key difference between these two methods is whether to use the task-specific classifier or the language model head.

In this subsection, we discuss whether the label word mechanism has a negative effect on downstream tasks compared to the classifier. Table 2 shows the performance of the plug-tagger and prefix-classifier, the key difference between these two methods is whether to use the classifier or the language model head. we find that the performance of plug-tagger is is slightly better than the prefix-classifier on all datasets. Thus, we deduce the label word mechanism can achieve pluggability without adversely affecting performance.

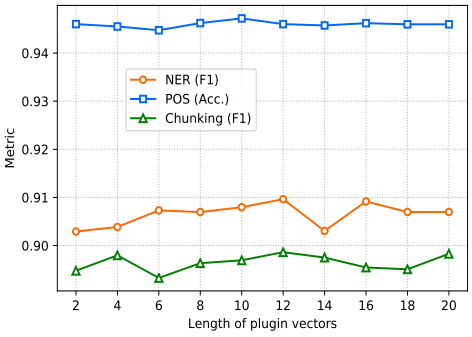

Figure 4: Performances on NER, POS and Chunking as the length of plugin vectors varies. Metric for NER and Chunking is F1, and for POS is accuracy.

4.2 Impact of Plugin Length

Since the length of plugin vectors we used in the experiment of pluggability is short, a key problem is whether such a short plugin vector would be expressive for downstream tasks. In this section, we discuss the impact of plugin length. We experimented with NER, POS, and Chunking on CoNLL2003, all hyperparameters are the same except the length of plugin vectors. The experiment results are shown in Figure 4.

We find that increasing the length of the plugin does not significantly improve the performance of all three tasks, which indicates that a short plugin is enough for the best performance in simple tasks. This further reflects the advantage of the plug-tagger. When the number of labels is large, the parameters of the classifier may be beyond those of the plugin vectors, bringing extra time cost to the deployment. However, the plug-tagger does not need the task-specific classifier, so it can complete tasks more quickly, as mentioned in section 3.5.

4.3 Comparison of Label Word Option

Recall in section 2.4, we discuss the option of selecting label words. We compare two ways to select a label word for the BIO schema. One is to select a label word for B label, and I label respectively. When performing label Word mapping, label words can be matched directly, so this option is called **DirectBI**. The other is that select a label word to represent both the B label and I label. During the inference phase, adjacent and identical label words are merged, the first label word is considered B, and the rest are considered I. We called this method **MergeBI**.

As shown in Figure 5, we find that MergeBI works better on the NER task of CoNLL2003

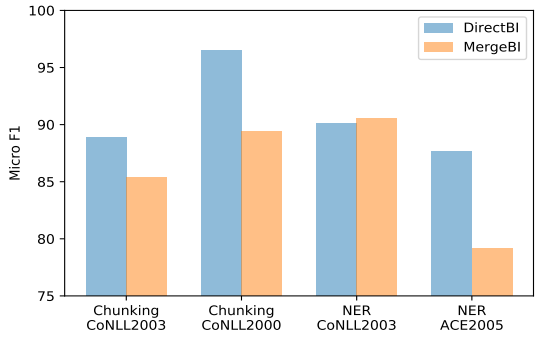

Figure 5: Performance comparison of two label word options in datasets based on BIO tagging schema.

but otherwise performs worse than DirectBI on all other datasets, especially the performance degradation is very obvious for each dataset of chunking. We find that there are adjacent words of the same label in the chunking task, but they represent two different phrases and therefore cannot be merged. For NER, the same phenomenon occurs in ACE2005 but is not found in CoNLL2003. This leads to inconsistencies in the trends of the two NER datasets. We suppose that the following two reasons cause MergeBI to improve NER's performance on CoNLL2003: a) We find that there are no adjacent similar entities in CoNLL2003. b) Some simple labels may be better represented by the same word. But MergeBI cannot be widely used because there are not many suitable scenarios. DirectBI is the more practical option since it achieved better performance in more situations.

5 Related Work

5.1 Sequence Labeling

Sequence labeling, such as named entity recognition (NER), part-of-speech (POS) tagging and chunking, is one of the fundamental tasks of natural language processing (Ma and Hovy, 2016). Recently neural network models achieve competitive performances (Chiu and Nichols, 2016; Dos Santos and Zadrozny, 2014; Luo et al., 2020), and fine-tuning the PLMs (Devlin et al., 2019; Liu et al., 2019; Yang et al., 2020) have been shown to achieve state-of-art results on sequence labeling (Bell et al., 2019). The above approach treats sequence labeling as token-level classification, works of (Athiwaratkun et al., 2020; Yan et al., 2021) convert sequence labeling into a generation task, avoiding task-specific classifier by using the Seq2Seq framework (Sutskever et al., 2014; Cho et al., 2014; Vaswani et al., 2017; Lewis et al., 2020). But most of them still need to modify the model structure and can't use native PLM, which defeats our goals.

5.2 Pre-trained Language Model

Self-supervised representation models (Radford et al., 2018, 2019; Yang et al., 2019; Peters et al., 2018; Devlin et al., 2019) have shown substantial advances in natural language understanding after being pre-trained on large-scaled text corpora and fine-tuned on downstream tasks. Given an NLP task, the mainstream paradigm to use PLM is finetuning, which stacks a linear classifier on top of the pre-trained language model and then updates all parameters (Zhao et al., 2020). Our method does not rely on the task-specific classifier to perform different task but instead predicts the label words for all sequence labeling tasks.

5.3 Lightweight Deep Learning

Lightweight deep learning method aims to use small trainable parameters to leverage the ability of PLMs (Houlsby et al., 2019). Some studies argue that redundant parameters in the model should be deleted or masked (Zaken et al., 2021; Sanh et al., 2020; Zhao et al., 2020; Frankle and Carbin, 2019), while others argue that additional structures should be added to the model (Zhang et al., 2020; Houlsby et al., 2019; Guo et al., 2021). For example, adapter-tuning insert some additional layer between each layer of PLMs. Prefix-tuning (Li and Liang, 2021) is a is a pluggable and lightweight method. It inserts continuous vectors into the input to allow a fixed PLM to do different generation tasks. However, the above methods basically need to modify the model structure or parameters, most of them cannot be applied to realize plug-and-play in classification tasks.

6 Conclusion

In this work, we propose plug-tagger, a pluggable framework for sequence labeling. The proposed framework can accomplish different tasks using vectors inserted into the input and a fixed PLM without modifying the model parameters and structure. It achieves competitive performance on the sequence labeling tasks with only a few parameters and is faster than other lightweight methods in real-world scenarios requiring model redeployment.

References

Ben Athiwaratkun, Cicero Nogueira dos Santos, Jason Krone, and Bing Xiang. 2020. Augmented natural language for generative sequence labeling. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 375–385, Online. Association for Computational Linguistics.

Samuel Bell, Helen Yannakoudakis, and Marek Rei. 2019. Context is key: Grammatical error detection with contextual word representations. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 103–115, Florence, Italy. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Jason PC Chiu and Eric Nichols. 2016. Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, 4:357–370.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Cicero Dos Santos and Bianca Zadrozny. 2014. Learning character-level representations for part-of-speech tagging. In *International Conference on Machine Learning*, pages 1818–1826. PMLR.

Jonathan Frankle and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks.

Demi Guo, Alexander M. Rush, and Yoon Kim. 2021. Parameter-efficient transfer learning with diff pruning.

Karen Hambardzumyan, Hrant Khachatrian, and Jonathan May. 2021. Warp: Word-level adversarial reprogramming.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2021. Gpt understands, too.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach.

Ying Luo, Fengshun Xiao, and Hai Zhao. 2020. Hierarchical contextualized representation for named entity recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8441–8448.

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

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. Adapterhub: A framework for adapting transformers. In *Proceedings*

of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020): Systems Demonstrations, pages 46–54, Online. Association for Computational Linguistics.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters.

Erik F Sang and Fien De Meulder. 2003. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*.

Victor Sanh, Thomas Wolf, and Alexander M Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. *arXiv preprint arXiv:2005.07683*.

Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. 2018. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

Erik F. Tjong Kim Sang and Sabine Buchholz. 2000. Introduction to the CoNLL-2000 shared task chunking. In *Fourth Conference on Computational Natural Language Learning and the Second Learning Language in Logic Workshop*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Hang Yan, Tao Gui, Junqi Dai, Qipeng Guo, Zheng Zhang, and Xipeng Qiu. 2021. A unified generative framework for various NER subtasks. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5808–5822, Online. Association for Computational Linguistics.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2020. Xlnet: Generalized autoregressive pretraining for language understanding.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32.

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models.

Jeffrey O Zhang, Alexander Sax, Amir Zamir, Leonidas Guibas, and Jitendra Malik. 2020. Side-tuning: A baseline for network adaptation via additive side networks.

Mengjie Zhao, Tao Lin, Fei Mi, Martin Jaggi, and Hinrich Schütze. 2020. Masking as an efficient alternative to finetuning for pretrained language models.

A Hyperparameters for Plug-Tagger

Task	Dataset	learning rate	plugin length
NER	CoNLL2003	2e-3	5
	ACE2005	2e-3	15
Chunking	CoNLL2003	2e-3	10
	CoNLL2000	2e-3	10
POS	CoNLL2003	4e-6	1
	WSJ	2e-3	5

Table 4: Hyperparameters for Plug-Tagger (0.1%).

Task	Dataset	learning rate	plugin length
NER	CoNLL2003	5e-3	5
	ACE2005	1e-2	5
Chunking	CoNLL2003	1e-3	5
	CoNLL2000	1e-2	5
POS	CoNLL2003	1e-2	5
	WSJ	1e-3	5

Table 5: Hyperparameters for Plug-Tagger (0.01%).

In Table 4 and Table 5, we report the hyperparameters used for our method documented in the experiment section.