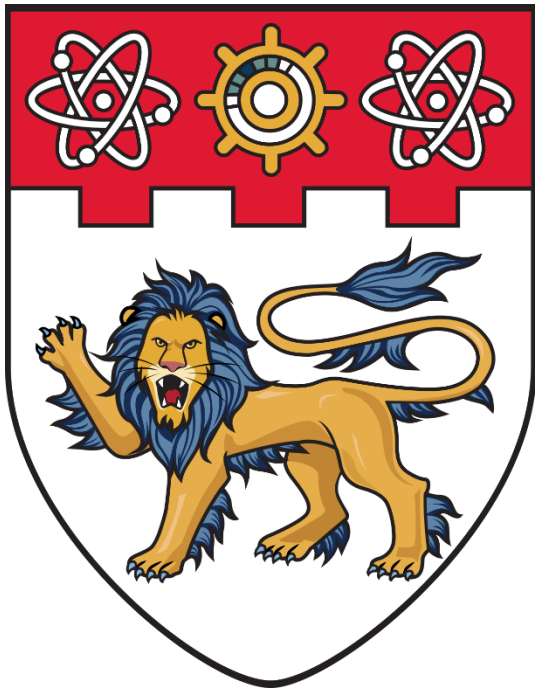

Named Entity Recognition Model
Bert-BiLSTM-CRF
Project Report

By: Sun Xinyu, Wang Qian
Wang Sijie, Wang Chenxi

Abstract

“Named Entity Recognition”(NER) is widely used in Natural Language Processing (NLP). The essential part of NER is information extraction that seeks to locate and classify information elements in unstructured test into predefined categories such as person names, locations, time, organizations, etc. The NER technology has matured over the years and can now extract information at a high accuracy. The most common and yet effective way of an NER in the industry now involves a standard CharCNN-BiLSTM-CRF format. The project report aims to explain why such a standard model is able to extract and categorize information accurately and on top of that, propose modifications to further enhance the performance of the NER model. Further, results of the improved model are given and compared with the industry standard, along with discussion and plans for further improvement.

Table of Contents

1. Introduction

1.1 Background

In any text document, there are particular terms that represent specific entities that are more informative and have a unique context. These entities are known as named entities, which more specifically refer to terms that represent real-world objects like people, places, organizations, and so on, which are often denoted by proper names. A naive approach could be to find these by looking at the noun phrases in text documents.

Named-entity recognition (NER) (also known as entity identification, entity chunking and entity extraction) is a sub-task of information extraction that seeks to locate and classify named entities in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. These extractions can be useful for finding events, finding relationships between entities, determining which organization were mentioned in a news article, and so on.

1.2 Dataset

As with any Deep Learning model, it requires a ton of data to test and evaluate results. High-quality datasets are the basis for any Deep Learning project. Luckily, there are several annotated, publicly available and mostly free datasets. Conll 2003 is the widely used and authorised public dataset to evaluate and test neutral language processing models. This dataset includes 1,393 English and 909 German news articles. For this project, we will be using English data from the Reuters Corpus which consists of thousands of news articles. News sentences in training dataset are divided into 4 categories which are LOC (location), MISC(miscellaneous), ORG(organization) and PER(person).

1.3 Aims

The goal of our NER project is to find all of the named people, places, and things within the test news datasets and correctly classify them. The gold standard benchmark for NER was laid out in a 2003 academic challenge called CoNLL. The CoNLL data set consists of news articles with all of the named entities hand-labeled by humans. This established the four standard NER classes: person (PER), organization (ORG), location (LOC), and miscellaneous (MISC). A human reader can deduce that "Paris Hilton" is a person, "the Hilton" is an organization, and "Paris" is a location.

2. Common Methodology

2.1 Common Structure

First of all, before introducing the model structure of our method, let us take a deep look at the common structures. Traditionally speaking, the model structure should start with a CNN structure followed by a BiLSTM structure and ends with the CRF.

2.2 Data Preprocessing

Before the training process, the data input will be preprocessed into word index, and every character in the word will be labelled with character index afterwards.

2.3 CharCNN

The Character CNN layer extracts useful information from the preprocessed data. Its methodologies, for example, max pooling makes it possible to output information that are irrelevant from the length of vocabulary input. The most useful information is pulled out. This, combined with the initial information, is sent to the later layers. An instinctive question would be, why do we add a CNN structure instead of an LSTM layer? There are mainly two reasons. Firstly, although LSTM could handle the character index directly, the whole procedure takes much longer. Starting with a CNN model could make the full algorithm faster. Secondly, unlike LSTM, CNN focuses more on the important character in the input token. For example, for the word ‘unhappy’, after going through the CNN layer, the prefix ‘un’, which is a much more important character, is enlarged compared to the alphabet a’, whose impact is relatively small. Hence, adding a CNN layer in the front will make the whole structure more efficient and precise.

2.4 BiLSTM

After going through CNN, the important character embedding will be picked out and combined with the original embedding and then be passed into the second structure: BiLSTM with one layer. In this layer, the character input will be combined with a more complex time sequence index. The reason we apply a BiLSTM model is to try to improve the output outcome. Compared to the traditional unidirectional LSTM which only preserves information of the past while bidirectional will run your inputs in two ways. Running information backwards preserves information from the future. Hence it is significant to consider two hidden states combined. The BiLSTM can thus enrich and extract the word features from sentences, which were later used in CRF prediction.

2.5 CRF

Finally, the features output from the LSTM layer will go into the CRF layer, which is a superior hidden Markov chain to perform the graph processing, and be classified with different labels. The LSTM CRF method is combining Deep learning’s strength of extracting high dimensional features, and the advantages of Traditional Machine Learning Graph Model CRF. This ultimately leads to better interpretability.

3. Modification Design

The model mentioned above is the most common NER model and has proved to be effective over the years. Yet, our group wanted to explore ways to potentially further enhance the model in order to achieve higher accuracy. The following two ways have been proposed.

3.1 Two-layer BiLSTM

Increasing the number of layers is the most direct method if we want to enhance the accuracy of our model. While a single layer of BiLSTM has proved to be effective enough to process relatively easy datasets, it seems slightly incompetent when the dataset becomes extremely complicated. Since the depth of neural networks is generally attributed to the success of the approach on a wide range of challenging prediction problems, adding a second layer to further process the data seems like a promising way to address the issue.

The additional hidden layers are understood to recombine with the learned representation from prior layers and create new representations at high levels of abstraction. For example, from lines to shapes to objects. This largely enhances the understanding of the dataset.

However, with each additional layer added into a neutral network, the amount of calculations needed increases tremendously. That means, a much longer period of time will be needed to run the program. This is especially so given that the NER model is much more complicated than a vanilla neural network. Hence, we have decided to use two layers of the BiLSTM model. This enhances the accuracy and yet still remains efficient enough in terms of data processing.

3.2 Bert

We have decided to incorporate Bert into our NER model. Bidirectional Encoder Representations from Transformers(BERT) is a language model that comes from a google paper. Used bidirectional training of Transformer to language model, Bert can have a deeper sense of language context. That allows it to achieve state-of-the-art results in a wide variety of NLP tasks.

One major setback of the current NER model is the ambiguity of words. The original word2vector is designed so that each word has a fixed index. But this causes ambiguity especially when a word has several different meanings. Taking the word “apple” as an example, it can either mean a fruit or a company name. This causes confusion, especially with the NER model, whose purpose is to assign different words to different categories.

Bert model easily fixes this problem. The model is able to fine tune a word’s index according to the words surrounding it. That means, if the word “orange” is spotted next to the word “apple”, the model adjusts the word “apple”’s index so that it leans more towards the fruit “apple”.

We have thus decided to use Bert as the model for initial word embedding. This theoretically decreases ambiguity and makes the NER model more accurate. Also, since Bert has already

divided sentences into separate word index, Character-Level CNN will no longer be needed. The modified model will have the structure of Bert-BiLSTM-CRF.

4. Findings

4.1 Result Values

```
02/15/2020 04:27:58 - INFO - utils -   guid: 0
02/15/2020 04:27:58 - INFO - utils -   tokens: [CLS] [UNK] - [UNK] [UNK] [UNK] [UNK] [UNK] [UN
02/15/2020 04:27:58 - INFO - utils -   input_ids: 101 100 1011 100 100 100 100 100 100 100 100
02/15/2020 04:27:58 - INFO - utils -   input_mask: 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
02/15/2020 04:27:58 - INFO - utils -   segment_ids: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
02/15/2020 04:27:58 - INFO - utils -   label_ids: 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0
convert examples: 3465it [00:02, 1160.42it/s]
02/15/2020 04:28:01 - INFO - __main__ -   ***** Running training *****
02/15/2020 04:28:01 - INFO - __main__ -     Num examples = 5500
02/15/2020 04:28:01 - INFO - __main__ -     Num Epochs = 10
02/15/2020 04:28:01 - INFO - __main__ -     Total optimization steps = 1000
Epoch:   0% 0/10 [00:00<?, ?it/s]
Iteration: 100% 688/688 [16:24<00:00,  1.43s/it]
02/15/2020 04:29:26 - INFO - __main__ -   ***** Running eval *****

Evaluating: 100% 155/155 [04:12<00:00, 1.62s/it]

           LOC   ; precision:   93.66%; recall:   92.93%; F1:   93.29; support 1668
           ORG   : precision:   90.81%; recall:   91.75%; F1:   91.28; support 1661
           PER   : precision:   96.95%; recall:   96.23%; F1:   96.59; support 1617
           MISC  : precision:   80.87%; recall:   85.19%; F1:   82.97; support 719
           total : precision:   92.21%; recall:   92.62%; F1:   92.41; support 5665
Epoch:  100% 15/15 [20:46<20:46, 20:46 min/it]


Process finished with exit code 0
```

The precision, recall and F1 values for every separate category are shown in the picture above. The results for all categories added up together are also given. The precision is 92.21%, recall is 92.62% and F1 value is 92.41.

4.2 Comparison With Other Models

The following table gives the top precision, recall and F1 value worldwide for various common models found in NER.

Model	% P	% R	% F1
BLSTM	88.61	88.50	88.56
BLSTM-CRF	90.33	88.81	89.56
CNN-BLSTM	89.23	90.97	90.09

CNN-BLSTM-CRF	91.36	91.24	91.29
BERT BASE(Devlin et al., 2018)	/	/	92.4
CNN large + fine tune	/	/	93.5
RNN-CRF + flair	/	/	93.47
BERT-BLSTM(2 layers)-CRF(ours)	92.21	92.62	92.41

4.3 Evaluation and Discussion

The table in section 4.2 shows a clear comparison between the various neural-based NER models. The evaluation metric for all models has been standardized for easy comparison. The precision means the right rate of samples that model believes they are right. The recall means the right rate of samples that the model should predict. And the most important one, F1 measure means the harmonic trade-off between these two metrics. Since model should consider both precision and recall. As the gap between these two metrics gets bigger, the F1 measure gets worse.

Despite the fact that all these values represent the highest accuracy worldwide for that specific model, our model still manages to get higher value for all evaluation metrics than most of the models (other than CNN large+fine tune and RNN-CRF+flair, which has a rather drastic structural difference with the traditional model). It is thus not difficult to tell that our modification of the model does lead to a better interpretation and hence higher accuracy.

5. Potential Issues

5.1 Identify Potential Issues

Size of the training set: CoNLL-220 is a standard dataset for NER training purpose with 946 articles, 14987 sentences and 203621 tokens for English data. The size of the data is not very large. When applying our model on other data sets with increasing data amount, the training time and F1 measure can change and we need to take this into consideration.

Potential Overfitting: When we propose our Bert-BiLSTM-CRF model, we do realize that both Bert and BiLSTM have the feature of extracting the surrounding words information. With both models doing the same thing, there might be a potential overfitting issue. Whether the combination of Bert and BiLSTM together will yield better results than just Bert model alone is hard to conclude. However, after running our Bert and BiLSTM model, the result is slightly higher than even the world's best Bert base Model. It is thus safe to say that combining Bert and BiLSTM does not cause a reduction of accuracy. In fact, it even slightly enhances accuracy.

5.2 Future Improvement

Changing the design of loss function: The model has the problem of imbalanced sample distribution (a large portion of data is with label as 0). We can try to use the method of focal loss or dice loss in the future.

6. Developer Manual

6.1 Data Processing

Since the original Conll 2003 data label format is IOB labeled, we first transform the data format to the standard BIO type.

```
def IOB2BIO(input_file, output_file):
    print("Convert IOB -> BIO for file:", input_file)
    with open(input_file, 'r') as in_file:
        fins = in_file.readlines()
    fout = open(output_file, 'w')
    words = []
    labels = []
    for line in fins:
        if len(line) < 3:
            sent_len = len(words)
            for idx in range(sent_len):
                if "I-" in labels[idx]:
                    label_type = labels[idx].split('-')[-1]
                    if (idx == 0) or (labels[idx - 1] == "O") or \
                            (label_type != labels[idx - 1].split('-')[-1]):
                        fout.write(words[idx] + " B-" + label_type + "\n")
                    else:
                        fout.write(words[idx] + " " + labels[idx] + "\n")
                else:
                    fout.write(words[idx] + " " + labels[idx] + "\n")
            fout.write('\n')
            words = []
            labels = []
        else:
            pair = line.strip('\n').split()
            words.append(pair[0])
            labels.append(pair[-1].upper())
    fout.close()
    print("BIO file generated:", output_file)
```

Secondly, after reading the examples from the file, we define the abstract class DataProcessor which has corresponding abstract methods to collect the train, dev, test data examples, and one class method, namedly read_data.

```python
class DataProcessor(object):
    """Base class for data converters for sequence classification data sets."""

    def get_train_examples(self, data_dir):
        """Gets a collection of `InputExample`s for the train set."""
        raise NotImplementedError()

    def get_dev_examples(self, data_dir):
        """Gets a collection of `InputExample`s for the dev set."""
        raise NotImplementedError()

    def get_labels(self):
        """Gets the list of labels for this data set."""
        raise NotImplementedError()

    @classmethod
    def _read_data(cls, input_file):
        """Reads a BIO data."""
        with codecs.open(input_file, 'r', encoding='utf-8') as f:
            lines = []
            words = []
            labels = []
            for line in f:
                contends = line.strip()
                tokens = contends.split(' ')
                if len(tokens) == 2:
                    words.append(tokens[0])
                    labels.append(tokens[1])
                else:
                    if len(contends) == 0:
                        l = ' '.join([label for label in labels if len(label) > 0])
                        w = ' '.join([word for word in words if len(word) > 0])
                        lines.append([l, w])
                        words = []
                        labels = []
                        continue
                    if contends.startswith("-DOCSTART-"):
```

We then implement NerProcesser to read files from NER format dataset. We reimplement read_data() function due to the specialty that the Conll2003 dataset has 4 items one raw. Meanwhile, we implement get_labels method for avoiding the manual operation of preprocess or predefine the labels in every different dataset.

```
def get_labels(self, labels=None):
    if labels is not None:
        try:
            # Read label from file.
            if os.path.exists(labels) and os.path.isfile(labels):
                with codecs.open(labels, 'r', encoding='utf-8') as fd:
                    for line in fd:
                        self.labels.append(line.strip())
            else:
                self.labels = labels.split(',')
            self.labels = set(self.labels)  # to set
        except Exception as e:
            print(e)
    if os.path.exists(os.path.join(self.output_dir, 'label_list.pkl')):
        with codecs.open(os.path.join(self.output_dir, 'label_list.pkl'), 'rb') as rf:
            self.labels = pickle.load(rf)
    else:
        if len(self.labels) > 0:
            self.labels = self.labels.union(set(["X", "[CLS]", "[SEP]"]))
            with codecs.open(os.path.join(self.output_dir, 'label_list.pkl'), 'wb') as rf:
                pickle.dump(self.labels, rf)
        else:
            self.labels = ["O", 'B-TIM', 'I-TIM', "B-PER", "I-PER", "B-ORG", "I-ORG", "B-LOC", "I-LOC", "
                           "[CLS]", "[SEP]"]
    return self.labels

def _create_example(self, lines, set_type):
    examples = []
    for (i, line) in enumerate(lines):
        guid = "%s-%s" % (set_type, i)
        text = tokenization.convert_to_unicode(line[1])
        label = tokenization.convert_to_unicode(line[0])
        # if i == 0:
        #     print('label: ', label)
        examples.append(InputExample(guid=guid, text=text, label=label))
    return examples
```

We added one special label “X”, because of the specialty of WordPiece in BERT tokenizer. It will divide one word into several parts, which will increase the rich information in one word but cause the confusion of labels, especially in NER problem. So we added a special label “X” for every excess word piece of tokenizer output and keep the same label for the first output.

```python
def convert_single_example(ex_index, example, label_list, max_seq_length, tokenizer, output_dir, mode):
    """..."""
    label_map = {}
    # 1表示从1开始对label进行index化
    for (i, label) in enumerate(label_list, 1):
        label_map[label] = i
    # 保存label->index 的map
    if not os.path.exists(os.path.join(output_dir, 'label2id.pkl')):
        with codecs.open(os.path.join(output_dir, 'label2id.pkl'), 'wb') as w:
            pickle.dump(label_map, w)

    textlist = example.text.split(' ')
    labellist = example.label.split(' ')
    tokens = []
    labels = []
    for i, word in enumerate(textlist):
        token = tokenizer.tokenize(word)
        tokens.extend(token)
        label_1 = labellist[i]
        for m in range(len(token)):
            if m == 0:
                labels.append(label_1)
            else:  # speciality
                labels.append("X")
    # tokens = tokenizer.tokenize(example.text)
    if len(tokens) >= max_seq_length - 1:
        tokens = tokens[0:(max_seq_length - 2)]  # -2 one for start and one for end.
        labels = labels[0:(max_seq_length - 2)]
    ntokens = []
    segment_ids = []
    label_ids = []
    ntokens.append("[CLS]")  # start [CLS]
    segment_ids.append(0)
    # append("O") or append("[CLS]") not sure!
    label_ids.append(label_map["[CLS]"])  # O OR CLS
```

We will log some samples for check while converting some examples.

```
    for i, token in enumerate(tokens):
        ntokens.append(token)
        segment_ids.append(0)
        label_ids.append(label_map[labels[i]])
    ntokens.append("[SEP]")  # add [SEP] in the end of sentence
    segment_ids.append(0)
    # append("O") or append("[SEP]") not sure!
    label_ids.append(label_map["[SEP]"])
    input_ids = tokenizer.convert_tokens_to_ids(ntokens)  # to IDS!
    input_mask = [1] * len(input_ids)
    # label_mask = [1] * len(input_ids)
    # padding, 使用
    while len(input_ids) < max_seq_length:
        input_ids.append(0)
        input_mask.append(0)
        segment_ids.append(0)
        # we don't concerned about it!
        label_ids.append(0)
        ntokens.append("**NULL**")
        # label_mask.append(0)
    # print(len(input_ids))
    assert len(input_ids) == max_seq_length
    assert len(input_mask) == max_seq_length
    assert len(segment_ids) == max_seq_length
    assert len(label_ids) == max_seq_length
    # assert len(label_mask) == max_seq_length

    # print some example information
    if ex_index < 5:
        logger.info("*** Example ***")
        logger.info("guid: %s" % (example.guid))
        logger.info("tokens: %s" % " ".join(
            [tokenization.printable_text(x) for x in tokens]))
        logger.info("input_ids: %s" % " ".join([str(x) for x in input_ids]))
        logger.info("input_mask: %s" % " ".join([str(x) for x in input_mask]))
        logger.info("segment_ids: %s" % " ".join([str(x) for x in segment_ids]))
        logger.info("label_ids: %s" % " ".join([str(x) for x in label_ids]))
```

And finally we get all the information in one sentence, and we structure this information to InputFeatures.

```
    # structure this information
    feature = InputFeatures(
        input_ids=input_ids,
        input_mask=input_mask,
        segment_ids=segment_ids,
        label_ids=label_ids,
        # label_mask = label_mask
    )
    # mode='test'的时候才有效
    write_tokens(ntokens, output_dir, mode)
    return feature
```

6.2 Model construction

The full structure of BERT is not described in the report due to its complexity. We directly clone the BERT part from Google AI Research, and the rest of upload files are implemented by us.

The code of BiLSTM CRF model implemented in Tensorflow is as follow:

```python
class BLSTM_CRF(object):
    def __init__(self, embedded_chars, hidden_unit, cell_type, num_layers, dropout_rate,
                 initializers, num_labels, seq_length, labels, lengths, is_training):
        """..."""
        self.hidden_unit = hidden_unit
        self.dropout_rate = dropout_rate
        self.cell_type = cell_type
        self.num_layers = num_layers
        self.embedded_chars = embedded_chars
        self.initializers = initializers
        self.seq_length = seq_length
        self.num_labels = num_labels
        self.labels = labels
        self.lengths = lengths
        self.embedding_dims = embedded_chars.shape[-1].value
        self.is_training = is_training

    def add_blstm_crf_layer(self, crf_only):
        """..."""
        if self.is_training:
            # lstm input dropout rate i set 0.9 will get best score
            self.embedded_chars = tf.nn.dropout(self.embedded_chars, self.dropout_rate)

        if crf_only:
            logits = self.project_crf_layer(self.embedded_chars)
        else:
            # blstm
            lstm_output = self.blstm_layer(self.embedded_chars)
            # project
            logits = self.project_bilstm_layer(lstm_output)
        # crf
        loss, trans = self.crf_layer(logits)
        # CRF decode, pred_ids
        pred_ids, _ = crf.crf_decode(potentials=logits, transition_params=trans, sequence_length=self.lengths)
        return (loss, logits, trans, pred_ids)
```

One important thing is projecting LSTM hidden states to CRF, which is:

```python
def project_crf_layer(self, embedding_chars, name=None):
    """
    hidden layer between input layer and logits
    :param lstm_outputs: [batch_size, num_steps, emb_size]
    :return: [batch_size, num_steps, num_tags]
    """
    with tf.variable_scope("project" if not name else name):
        with tf.variable_scope("logits"):
            W = tf.get_variable("W", shape=[self.embedding_dims, self.num_labels],
                                dtype=tf.float32, initializer=self.initializers.xavier_initializer())

            b = tf.get_variable("b", shape=[self.num_labels], dtype=tf.float32,
                                initializer=tf.zeros_initializer())
            output = tf.reshape(self.embedded_chars,
                                shape=[-1, self.embedding_dims])  # [batch_size, embedding_dims]
            pred = tf.tanh(tf.nn.xw_plus_b(output, W, b))
        return tf.reshape(pred, [-1, self.seq_length, self.num_labels])
```

And we can calculate the CRF losses in this code:

```
def crf_layer(self, logits):
    """
    calculate crf loss
    :param project_logits: [1, num_steps, num_tags]
    :return: scalar loss
    """
    with tf.variable_scope("crf_loss"):
        trans = tf.get_variable(
            "transitions",
            shape=[self.num_labels, self.num_labels],
            initializer=self.initializers.xavier_initializer())
        if self.labels is None:
            return None, trans
        else:
            log_likelihood, trans = tf.contrib.crf.crf_log_likelihood(
                inputs=logits,
                tag_indices=self.labels,
                transition_params=trans,
                sequence_lengths=self.lengths)
            return tf.reduce_mean(-log_likelihood), trans
```

After introducing the model part of the code, the last step is construct the workflow.

We define the loss mean square error in the tf.metrics module.

```
    output_spec = None
    if mode == tf.estimator.ModeKeys.TRAIN:
        # train_op = optimizer.optimizer(total_loss, learning_rate, num_train_steps)
        train_op = optimization.create_optimizer(
            total_loss, learning_rate, num_train_steps, num_warmup_steps, False)
        hook_dict = {}
        hook_dict['loss'] = total_loss
        hook_dict['global_steps'] = tf.train.get_or_create_global_step()
        logging_hook = tf.train.LoggingTensorHook(
            hook_dict, every_n_iter=args.save_summary_steps)

        output_spec = tf.estimator.EstimatorSpec(
            mode=mode,
            loss=total_loss,
            train_op=train_op,
            training_hooks=[logging_hook])

    elif mode == tf.estimator.ModeKeys.EVAL:
        # modify for ENR
        def metric_fn(label_ids, pred_ids):
            return {
                "eval_loss": tf.metrics.mean_squared_error(labels=label_ids, predictions=pred_ids),
            }

        eval_metrics = metric_fn(label_ids, pred_ids)
        output_spec = tf.estimator.EstimatorSpec(
            mode=mode,
            loss=total_loss,
            eval_metric_ops=eval_metrics
        )
    else:
        output_spec = tf.estimator.EstimatorSpec(
            mode=mode,
            predictions=pred_ids
        )
    return output_spec
```

We setup the CUDA environment and BERT config.

```
def train(args):
    os.environ['CUDA_VISIBLE_DEVICES'] = args.device_map

    processors = {
        "ner": NerProcessor
    }
    bert_config = modeling.BertConfig.from_json_file(args.bert_config_file)

    if args.max_seq_length > bert_config.max_position_embeddings:
        raise ValueError(
            "Cannot use sequence length %d because the BERT model "
            "was only trained up to sequence length %d" %
            (args.max_seq_length, bert_config.max_position_embeddings))

    if args.clean and args.do_train:
        if os.path.exists(args.output_dir):
            def del_file(path):
                ls = os.listdir(path)
                for i in ls:
                    c_path = os.path.join(path, i)
                    if os.path.isdir(c_path):
                        del_file(c_path)
                    else:
                        os.remove(c_path)

            try:
                del_file(args.output_dir)
            except Exception as e:
                print(e)
                print('pleace remove the files of output dir and data.conf')
                exit(-1)
```

We define the early stop process, if the model performance did not increase for 10 epochs, we will stop training and get the current best model.

```
train_input_fn = file_based_input_fn_builder(
    input_file=train_file,
    seq_length=args.max_seq_length,
    is_training=True,
    drop_remainder=True)
# estimator.train(input_fn=train_input_fn, max_steps=num_train_steps)

eval_file = os.path.join(args.output_dir, "eval.tf_record")
if not os.path.exists(eval_file):
    filed_based_convert_examples_to_features(
        eval_examples, label_list, args.max_seq_length, tokenizer, eval_file, args.output_dir)

eval_input_fn = file_based_input_fn_builder(
    input_file=eval_file,
    seq_length=args.max_seq_length,
    is_training=False,
    drop_remainder=False)

# train and eval togither
# early stop hook
early_stopping_hook = tf.estimator.experimental.stop_if_no_decrease_hook(
    estimator=estimator,
    metric_name='loss',
    max_steps_without_decrease=num_train_steps,
    eval_dir=None,
    min_steps=0,
    run_every_secs=None,
    run_every_steps=args.save_checkpoints_steps)

train_spec = tf.estimator.TrainSpec(input_fn=train_input_fn, max_steps=num_train_steps,
                                    hooks=[early_stopping_hook])
eval_spec = tf.estimator.EvalSpec(input_fn=eval_input_fn)
tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)
```

Finally, we predict the evaluation result in the code below:

```
if args.do_predict:
    token_path = os.path.join(args.output_dir, "token_test.txt")
    if os.path.exists(token_path):
        os.remove(token_path)

    with codecs.open(os.path.join(args.output_dir, 'label2id.pkl'), 'rb') as rf:
        label2id = pickle.load(rf)
        id2label = {value: key for key, value in label2id.items()}

    predict_examples = processor.get_test_examples(args.data_dir)
    predict_file = os.path.join(args.output_dir, "predict.tf_record")
    filed_based_convert_examples_to_features(predict_examples, label_list,
                                             args.max_seq_length, tokenizer,
                                             predict_file, args.output_dir, mode="test")

    logger.info("***** Running prediction*****")
    logger.info("  Num examples = %d", len(predict_examples))
    logger.info("  Batch size = %d", args.batch_size)

    predict_drop_remainder = False
    predict_input_fn = file_based_input_fn_builder(
        input_file=predict_file,
        seq_length=args.max_seq_length,
        is_training=False,
        drop_remainder=predict_drop_remainder)

    result = estimator.predict(input_fn=predict_input_fn)
    output_predict_file = os.path.join(args.output_dir, "label_test.txt")

    def result_to_pair(writer):...

    with codecs.open(output_predict_file, 'w', encoding='utf-8') as writer:
        result_to_pair(writer)
    from bert_base.train import conlleval
    eval_result = conlleval.return_report(output_predict_file)
    print(''.join(eval_result))
```

7. Conclusion

Our group wants to explore ways to potentially further enhance the current widely used models in NER.

Despite the fact that the highest F1 score performing model worldwide is CNN+Fine tune, our model still manages to get higher value for evaluation metrics than most of the models. In conclusion, by comparing to the many commonly used models of NER, the capability and code efficiency of our Bert-BILSTM-CRF model has proved to be effective suggested by F1 score of over 92.