Pretrained Word Embeddings and Character-Level Representations for Noisy NER

Hope McGovern
Newnham College
hem52@cam.ac.uk

Abstract

In this paper, we present our approach to the Novel and Emerging Named Entity Recognition shared task at the EMNLP 2017 Workshop on Noisy User-generated Text (W-NUT). We show that the combination of pre-trained word embeddings and character-level encodings with some hyperparameter tuning can be used to improve upon a baseline obtained with a traditional Bidirectional Long Short-Term Memory (BiLSTM) network. Our approach is straightforward and does not require the use of external information such as gazetteers or additional data; however, we are able to achieve a surface F1-score comparable to the results from the shared task.

1 Introduction

Named Entity Recognition (NER) is a sequence labelling task which aims to identify certain kinds of entities within a span of unstructured text, e.g. a person's name, a location, an organization, etc. and typically serves at the first step in many downstream applications such as information extraction and text summarization (Aguilar et al., 2017). Traditional NER systems commonly make use of extensively designed hand-crafted syntactic features or gazetteers (external lists of named entities) to achieve high performance; however, recent models tend to favor end-to-end approaches which use the text itself as the input to a neural network (Zhou and Xu, 2015). Incidentally, the top performing paper from the shared task used a mix of gazetteers and neural representations at the character- and word-level (Derczynski et al., 2017).

There is much interest in being able to accurately tag Named Entities (NEs) in user-created content posted to social media outlets such as Twitter, Instagram, and Facebook. Beyond the sheer volume of data availability, social media data typically represents a wider swath of linguistic diversity than is captured by many widely used NLP corpora, e.g. the Brown Corpus, WSJ, Switchboard (von Däniken and Cieliebak, 2017). However, that same linguistic variety also introduces a bevy of challenges to NE systems, among which are the prevalence of non-standard dialects, rare spellings, code switching (alternating between two or more languages in conversation), slang terms, and pictographic tokens (emojis).

All of these phenomena result in a large number of emerging named entities and rare surface forms, which remain difficult to detect (Derczynski et al., 2017; Augenstein et al., 2017). Traditional NER systems do not generalize well for novel and emerging settings, with the best performing system from the shared task reporting a surface F1-score of 40.24 and entity F1-score of 41.86. In comparison, state-of-the-art models trained on English Wikipedia data and a large amount of web data achieve span-level (entity) F1-score of more than 90 (Baevski et al., 2019; von Däniken and Cieliebak, 2017).

A common network architecture for NER is a bidirectional LSTM with the addition of a Conditional Random Field (CRF) classifier (Aguilar et al., 2017; Lin et al., 2017; Yamada et al., 2020; Huang et al., 2015). While we do not explore the use of a CRF layer in this paper, it is worth noting that a linear-chain CRF is almost always beneficial in an NER context because it takes into account the relationship between the current label and its neighbors. For example, there is a hard constraint within the IOB2 tagging scheme that 'I-Loc' cannot follow 'B-Per', and CRF exploits that constraint. Another popular method which has emerged since the shared task is the use of contextualized embeddings with BERT or ELMo (Yamada et al., 2020; Peters et al., 2018).

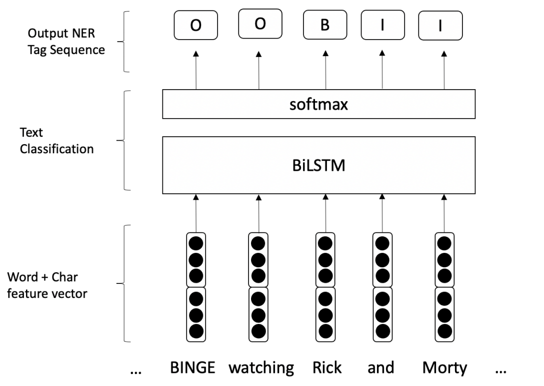
Output NER Tag Sequence
O
O
B
I
I
softmax
Text Classification
BiLSTM
Word + Char feature vector
... BINGE watching Rick and Morty ...

Figure 1: Overview of our approach applied to a sentence fragment present in the development set.

2 Methodology

Figure 1 shows the overview of our approach: we first use a bidirectional LSTM to build character-level representations from embeddings of every character in each word, and then combine that vector with token-level representations built from pre-trained embeddings. The concatenated representation is then fed into a standard BiLSTM.

2.1 Data Description

For all experiments, we consider the WNUT17 dataset, which was collected from a mixture of data from Twitter (for the training set), Youtube, StackExchange, and Reddit (for the development and test sets). All together, the dataset comprises 1,000 annotated tweets, totalling 65,124 tokens, and was annotated using the IOB2-tagging scheme with the following 6 categories: person, location (including GPE, facility), corporation, product (tangible goods, or well-defined services), creative-work (song, movie, book and so on), and group (subsuming music band, sports team,and non-corporate organisations). The classes are significantly imbalanced: there are 59095 'O'-tagged tokens in the training data, while there are only 1964 'B'-tagged tokens and 1177 'I'-tagged tokens. (Derczynski et al., 2017).

2.2 Feature Description

A common approach to NER is to incorporate syntactic information about the data, such as POS tags and dependency roles, into the feature vectors (Sikdar and Gambäck, 2017; von Däniken and Cieliebak, 2017). While this approach sometimes boosts performance, it adds an unwelcome task of hand-crafted feature extraction. Additionally, it is not always the case that the inclusion of syntactic features will improve performance, as Lin et al. (2017) report. We pursue instead a completely end-to-end architecture with no need for extra feature engineering. Specifically, we make use of information on two different scales: the character-level and the word-level.

Character-level Representation sub-word representations play an important role in capturing orthographic and morphological information of words in noisy settings due to the presence of novel abbreviations and spelling irregularities (Lin et al., 2017). To obtain these representations, we first use regular expressions to replace all urls with the token '<URL>', and calculate a vocabulary of all the unique characters in the data (92 characters comprised of alphanumeric characters [a-z], [A-z], and [0-9], as well as a handful of special characters). Using this vocabulary map, we associate each character c_i of word $w = [c_1, ..., c_p]$ to a vector $c_i \in R^{d_3}$, choosing some fixed length to pad or truncate each character vector (Lample et al., 2016).

The token length in the data is highly variable, reflecting its noisy nature. For example, the longest token in the dev data, *'jaw-dropping-revelations-hearings-motion-dismiss-dnc-fraud-lawsuit'*, is an astonishing 66 characters, while the average over the whole dev set (with URLS removed) is only 3 characters. We choose to represent each character as a 10-dim vector. We then pass the sequence of character embeddings to a bi-LSTM and concatenate the output of the forward LSTM and the backward LSTM to obtain the final vector $w_{chars} \in R^{d_2}$.

Word-level Representation For the word-level representation, we make use of pre-trained word embeddings. Pre-trained embeddings have delivered impressive results on a variety of NLP tasks (Lample et al., 2016). GloVe, introduced by Pennington et al. in 2014, is one such unsupervised learning algorithm that can be used to obtain a vector representation of words. We compare the downstream results with two distinct GloVe models: one trained on the Wikipedia 2014 and Gigaword 5 corpus and comprises 6B tokens, the other trained on 2 billion tweets and comprises 27B tokens. For all out-of-vocabulary words, we use vectors of 0s. As the GloVe embeddings are uncased, we also convert all tokens to lowercase before creating the mapping of token to embedding. Then, we simply concatenate the previous w_{chars} to the word embedding w_{glove} to get the overall word vector, $w = [w_{glove}, w_{chars}]$, whose dimension, R^n is the sum of the dimensions of each vector component, $n = d_1 + d_2$.

3 Model Description

[1] **Character-level LSTM** We build a bi-LSTM to encode the character information contained in each token. The character embeddings are fed through a bi-LSTM of 20 units, with an output embedding size of 10. It is then concatenated with the GloVe embeddings at 256-dim representation. This LSTM has no dense layer, as we do not use it for classification, – we only use it to obtain an embedding. A diagram of this network can be seen in Figure 2.

Main LSTM Our main model is a simple bidirectional LSTM. The architecture is as follows: an embedding layer which is the concatenation of the char-level word representations and the pre-trained GloVe embeddings, then a bi-LSTM with 50 units followed by a dropout layer, which acts as a regularizer (Srivastava et al., 2014). Finally, a dense layer with softmax activation computes the probability of a tag sequence given a sequence of word vectors. A diagram of the main network may be seen in Figure 3.

The WNUT17 dataset is a highly imbalanced dataset: there are far more non-entities than there are NEs, and the addition of the padding label adding in the preprocessing skews the data even further. To help combat this and avoid predicting only the majority class, the softmax layer is initialized with a bias based on the prior distribution of labels in the training set.

For our 4 classes (‘I’, ‘O’, ‘B’, or <PAD>), the softmax layer normalizes the scores into a vector p such that each element $p \in [p_1, ..., p_m]$ can be interpreted as the probability that the word belongs to class i (positive, sum to 1). Then, the probability $P(y)$ of a sequence of tag y is the product:

$$\prod_{t_1}^{m} p_t[y_t]$$

Unlike linear-chain CRFs, the softmax is only able to make local choices and cannot take the history of tag distributions into account.

[1]All models were implemented in Tensorflow 2.0 with Keras

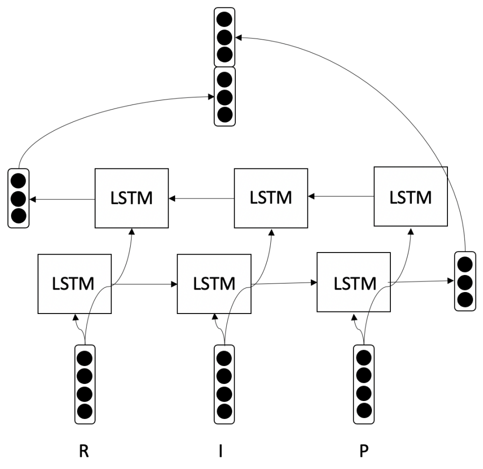

Figure 2: Character-level representation demonstrated with a token present in the development set.

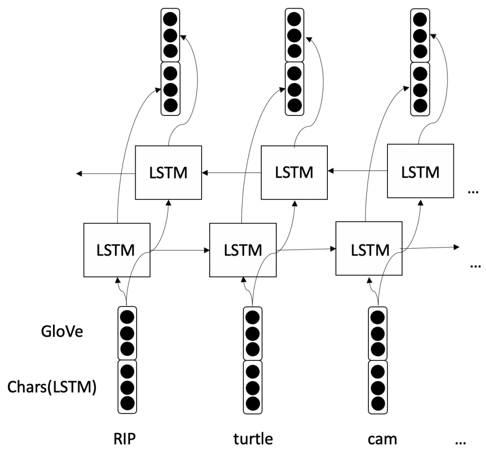

Figure 3: Word-level representation in the main model prior to being passed through a softmax. Demonstrated on a sequence present in the development set.

4 Experiments

Before conducting any experiments, we set a random seed to render the model training deterministic with respect to input data and hyperparameters. For our baseline experiment, the word vectors are randomly initialized embedding matrices. The pseudocode for our training algorithm can be seen in Algorithm 1.

5 Hyperparameter Tuning

We tuned the dimensions of both sets of GloVe embeddings {25, 50, **100**, 200, 300} as well as the dimensions of the character embeddings {**10**, 15, 20}. The values in boldface yielded the most favorable results and were used to obtain our final

Algorithm 1 Bidirectional LSTM model

```
for each epoch do
    for each batch do
        character bi-LSTM model forward pass
        character bi-LSTM model backward pass
        concatenate char and word vectors
        main bi-LSTM forward pass
        main bi-LSTM backward pass
        update parameters
    end for
end for
```

metrics on the test set. Additionally, we tuned the extent to which we 'downweight' the label vectors: down-weighting is a simple technique used to deal with class imbalance wherein the one-hot labels for our majority class or classes are multiplied by small values to decrease the likelihood of their prediction. We try downweighting with values of {1, 0.1, **0.01**, 0.001}. For all experiments, the embedding size of the word-level bi-LSTM is 128, and the length of the input sequences is set at 105.

During training, we use the Adam optimization algorithm, categorical cross entropy loss, a learning rate of $1e^{-3}$, and a batch size of 32. We train for 100 epochs with an early stopping patience of 3 epochs. Early stopping terminates the training loop prematurely if some criterion (in our case, the area under the curve as computed on the validation data) fails to increase.

We use the development set as validation data during training and for all evaluation during tuning. We only evaluate on the test set once all aspects of the architecture have been chosen.

6 Evaluation Method

To evaluate our models, we take the best set of hyperparameters from running small tests on the development set and record results on unseen test data.

In traditional NER systems, a common evaluation metric is the entity-level F1-score, which takes into account the granular labels for person, location, etc. in its calculation. However, this paper only focuses on the surface F1-score, which is concerned exclusively with predicting the labels 'I', 'O', or 'B' (for prediction, the padding labels are stripped). F1 is a weighted metric that depends on the precision and recall scores weighted equally. We also report the precision and recall scores to gain a finer understanding of the overall F1-score.

Embed Dim	**Wiki + Giga**	**Twitter**
25	N/A	33.4
50	27.2	31.7
100	30.4	**41.4**
200	31.7	38.4
300	**34.0**	N/A

Table 1: Tuning the embedding dimension and underlying GloVe model. Results on the development data are reported. The best result in each column is in boldface.

Features	Prec	Rec	**Surface F1**
Word (baseline)	0.099	0.474	16.4
Word (no bias)	0.066	0.318	10.9
Word (downweight)	0.060	0.071	6.5
Char+Word	0.210	0.632	31.5
Char+GloVe(Giga)	0.215	0.621	31.9
Char+GloVe(Twitter)	**0.274**	**0.639**	**33.9**

Table 2: Precision, Recall, and Surface F1-score for each model tested. The highest value in each category is in boldface.

7 Results

We report our results in Table 2. We also provide a table comparing the surface F1 score across different embedding dimensions of the two different GloVe models compared, the Wikipedia + Gigaword model and the Twitter model. This is shown in Table 1.

8 Discussion

Our highest performing model, one that incorporated character-level embeddings, pretrained GloVe embeddings trained on Twitter data, a biased softmax layer, and downweighted label vectors, achieves a surface F1-score of 33.9 on the test data. For every model, the recall value is greater than the precision value, and modifications to the network had a larger percentage increase on the precision rather than the recall – in other words, the models all have a tendency to underpredict the labels we care about: 'B' and 'I', and achieving a higher performance is primarily a function of encouraging a model to predict more 'B's and 'I's and only secondarily about improving the accuracy of those predictions. This might help explain why one of the largest performance gains we observed was merely a function of downweighting the label vectors. The class imbalance of the dataset proves to be one of the largest hurdles to accurate NER.

We saw another large performance boost from simply incorporating sub-word information. This suggests that Lin et al.'s claim about the richness of character-level embeddings is correct. The sub-word embeddings seem able to encode morphological and syntactic information that token-level representations seem to miss on the noisy data.

We do observe a difference in the performance of the two GloVe models. However, it is unclear if the increase in performance when using pre-trained embeddings trained on Twitter data arises because of the similarity between the vocabulary or whether it is simply because the twitter GloVe model has 4.5 times the number of tokens. Due to the nature of the dataset – that only the training data is collected from Twitter while the dev and test data is pulled from other social media platforms – we are inclined to believe that the sheer size of the Twitter GloVe model was the prime motivating factor for the improvement.

Extensions to this system that might yield even higher performance include incorporating a CRF classifier instead of the softmax layer from the Bi-LSTM, or incorporating additional features such as POS tags.

9 Conclusion

We have presented a simple, end-to-end system for NER classification that makes use of character level as well as token level information to form a comprehensive word vector and shown that although we have not considered syntactic features or external knowledge bases, we can achieve reasonable accuracy on noisy user-generated social media data.

References

Gustavo Aguilar, Suraj Maharjan, Adrian Pastor López-Monroy, and Thamar Solorio. 2017. A multi-task approach for named entity recognition in social media data. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 148–153, Copenhagen, Denmark. Association for Computational Linguistics.

Isabelle Augenstein, Mrinal Das, Sebastian Riedel, Lakshmi Vikraman, and Andrew McCallum. 2017. Semeval 2017 task 10: Scienceie - extracting keyphrases and relations from scientific publications.

Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. 2019. Cloze-driven pretraining of self-attention networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5360–5369, Hong Kong, China. Association for Computational Linguistics.

Pius von Däniken and Mark Cieliebak. 2017. Transfer learning and sentence level features for named entity recognition on tweets. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 166–171, Copenhagen, Denmark. Association for Computational Linguistics.

Leon Derczynski, Eric Nichols, Marieke van Erp, and Nut Limsopatham. 2017. Results of the WNUT2017 shared task on novel and emerging entity recognition. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 140–147, Copenhagen, Denmark. Association for Computational Linguistics.

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

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

Bill Y. Lin, Frank Xu, Zhiyi Luo, and Kenny Zhu. 2017. Multi-channel BiLSTM-CRF model for emerging named entity recognition in social media. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 160–165, Copenhagen, Denmark. Association for Computational Linguistics.

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.

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

Utpal Kumar Sikdar and Björn Gambäck. 2017. A feature-based ensemble approach to recognition of emerging and rare named entities. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 177–181, Copenhagen, Denmark. Association for Computational Linguistics.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. 2020. Luke: Deep contextualized entity representations with entity-aware self-attention.

Jie Zhou and Wei Xu. 2015. End-to-end learning of semantic role labeling using recurrent neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1127–1137, Beijing, China. Association for Computational Linguistics.

A Appendix

We use the keras functional API to combine a character-level encoding with a word-level encoding extracted from an input sequence and use it to predict NER tags over the sequence.

```python

# given some GloVe embedding matrix, keras metrics, and an initial bias, this code snippets builds a keras model. the code for calculating these is given in follow code snippets

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import LSTM, Embedding, Dense, TimeDistributed, Dropout
from tensorflow.keras.layers import Bidirectional, concatenate

n_labs = 4

### WORD LEVEL ###
word_in = Input(shape=(105,))

emb_word = Embedding(input_dim=14803, output_dim=100,
                     input_length=105, \

    embeddings_initializer=keras.initializers.Constant(embedding_matrix]), \
                     mask_zero=True, trainable=False)(word_in)

### CHAR LEVEL ###
char_in = keras.Input(shape=(105, 10,))
emb_char = TimeDistributed(Embedding(input_dim=94, output_dim=10, input_length=10, mask_zero=True))(char_in)

char_enc = TimeDistributed(Bidirectional(LSTM(units=20, return_sequences=False, dropout=0.5)))(emb_char)

# main LSTM
x = concatenate([emb_word, char_enc])

main_lstm = Bidirectional(LSTM(units=50,
     return_sequences=True, dropout=0.2,
     recurrent_dropout=0.2))(x)
main_lstm = Dropout(0.5)(main_lstm)
out = TimeDistributed(Dense(4, activation="softmax", bias_initializer=tf.keras.initializers.Constant(bias])))(main_lstm)
model = Model([word_in, char_in], out)

model.compile(optimizer=keras.optimizers.Adam(lr=1e-3), loss=keras.losses.CategoricalCrossentropy(), metrics=metrics)
```

The bias vector which is used to instantiate the dense layer is computed as follows. It uses code originally presented in the assignment colab notebook.

```
def get_initial_bias(padded_labels):
    all_labs = [l for lab in padded_labels for l in lab]
    label_count = Counter(all_labs)
    total_labs = len(all_labs)
    print(label_count)
    print(total_labs)

    # use this to define an initial model bias
    initial_bias=[(label_count[0]/total_labs), (label_count[1]/total_labs),
                  (label_count[2]/total_labs), (label_count[3]/total_labs)]
    print('Initial bias:')
    print(initial_bias)
    return initial_bias
```

To compute the GloVe embedding matrix:

```

############## adding GloVe pre-trained embeddings ################

   # given a vocab_size and path_to_glove_file, and embedding_dim

   embeddings_index = {}
   with open(path_to_glove_file) as f:
         for line in f:
               word, coefs = line.split(maxsplit=1)
               coefs = np.fromstring(coefs, "f", sep=" ")
               embeddings_index[word] = coefs


   word_index = dict(enumerate(token_vocab))
   embedding_dim = HP['glove_embedding_dim']


   # Prepare embedding matrix
   embedding_matrix = np.zeros((vocab_size, embedding_dim))
   for i, word in word_index.items():
         try:
               word = str(np.char.lower(word))
         except:
               pass
         embedding_vector = embeddings_index.get(word)
         if embedding_vector is not None:
               # Words not found in embedding index will be all-zeros.
               # This includes the representation for "padding" and "OOV"
               embedding_matrix[i] = embedding_vector
```