Thai word segmentation with bi-directional RNN

Jussi Jousimo

October 14, 2017

Introduction

In recent years, deep learning has provided state-of-the-art results in machine learning including natural language processing (NLP). Majority of the development in NLP has been based on English and other well studied languages, mainly due to the availability of large and standard corpus. In contrast to this, we show an example how to process Thai language.

Like many other East Asian languages such as Chinese and Japanese, words in Thai language are typically written together without boundary markers. For these languages, word segmentation is one of the first tasks in building applications of NLP such as topic classification, sentiment analysis, document similarity and chatbots. In Thai language, there is no clear definition what constitutes a word as it depends on the context and writing down all possible rules is difficult. This sets a challenge which we approach with deep learning.

Recurrent neural network (RNN) is a special kind of artificial neural network (ANN) which has been one of the most successful models to NLP to date. We describe a simple tokenizer based on a RNN and provide Python code for TensorFlow 1.4 or newer, which is a popular framework from Google to build and train computational graphs. In overall, the model works so that it learns weights in an ANN which provide rules to predict word boundaries. The weights are obtained by training the network iteratively by matching a sequence of characters in a sentence with a sequence of manually labelled word boundaries. As the training data, we use the InterBEST 2009 corpus with 148,995 sentences.

Preprocessing

In the first step found in file `preprocess.py`, we preprocess the BEST corpus by separating each sentence into a list, creating input data by removing word boundary markers and output data by creating a binary sequence, where 1 indicates beginning of a word and 0 indicates a character inside or at the end of a word.

Each character in the input is mapped to a unique integer with a dictionary, which serves as a lookup table for converting between characters and labels. The dictionary contains Thai alphabets (or abugidas to be specific), Latin alphabets, numerals and punctuation. Rest of the characters are mapped to an "unknown" label since these are deemed to contain no information needed in the tokenization.

The sentences are split into training and validation sets by ratio 9:1 and are saved along with the sentence lengths as TFRecords format files. The format allows TensorFlow to access the data during training and validation of the model.

Batching

The model is trained with batches of data for efficiency and each batch contains multiple sentences. Since the sentences are of different lengths, we must pad the input and output data to the maximum length T determined by the longest sentence in the batch. Below, a batch of three input sentences with character labels and corresponding word boundary sequences with padding is illustrated. For simplicity, we leave the batch dimension out in the notation from now on unless otherwise specified.

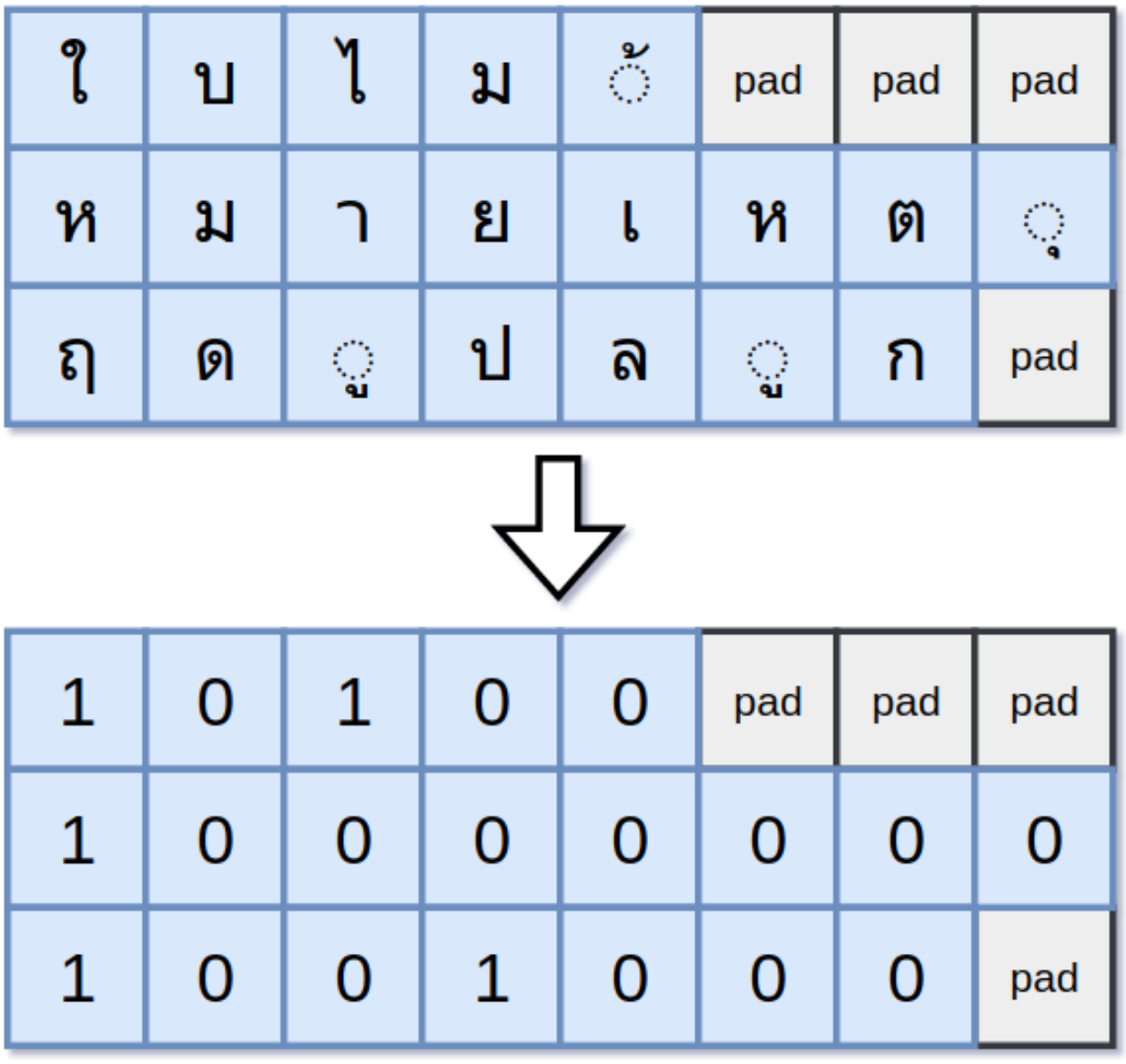

Figure 1: A batch of three sentences with padding in the end. Upper one contains input labels (illustrated with the actual characters instead of the corresponding integer labels) and the lower one output labels of the word boundaries.

Model architecture

Architecture of the ANN is show on the figure below with a formulation of a single character at position t in the input sequence for each layer. Blue entities indicate vectors and red GRU units in the RNN. In the following, we describe each layer in more detail. Source code for the model is found in file `thai_word_segmentation/model.py` which defines class `ThaiWordSegmentationModel` that includes methods for reading the data, building the model, validating the results and training the model. These are run from `train.py`.

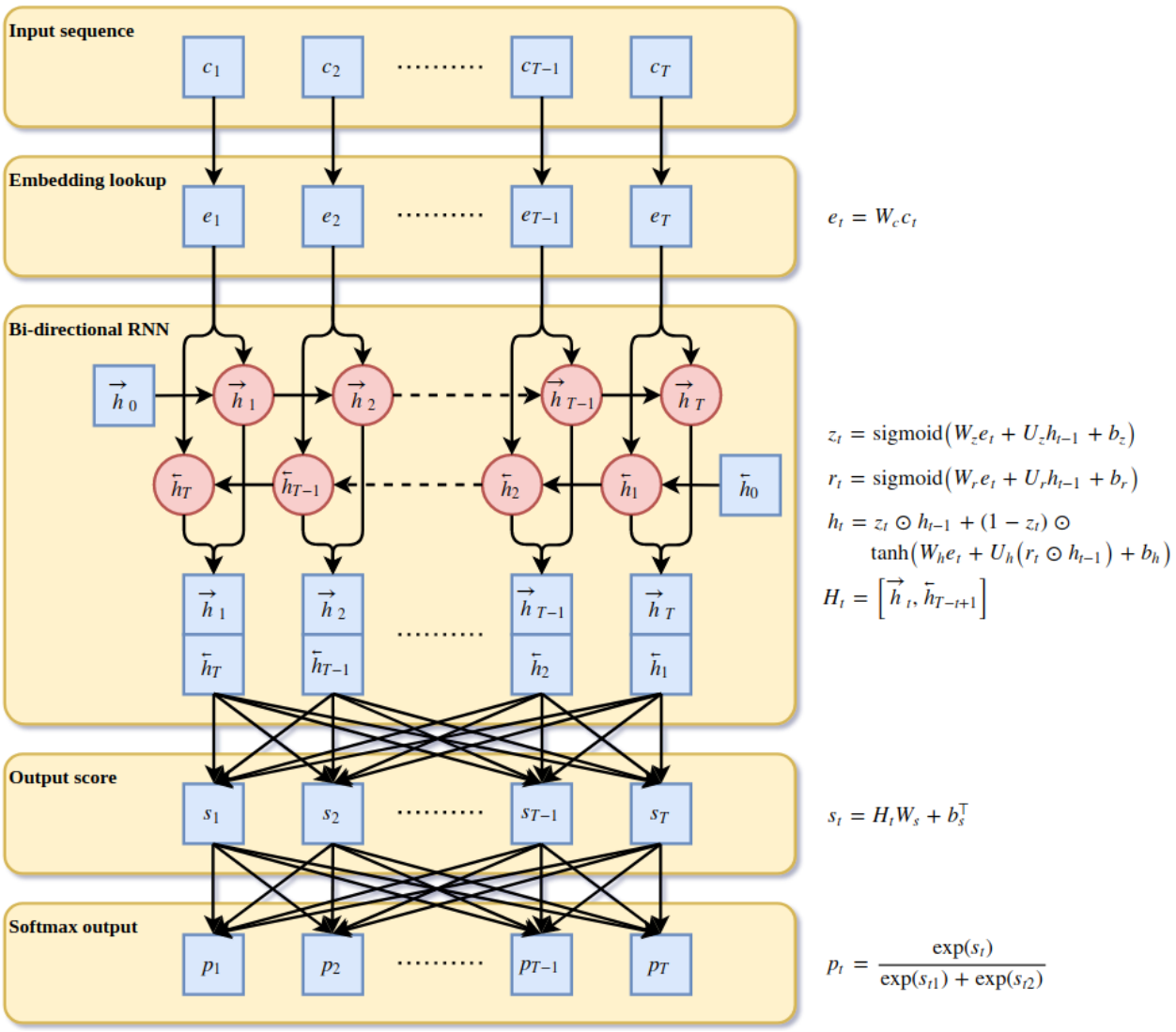

Figure 2: Artificial neural network architecture for Thai word segmentation. Vectors are indicated in blue and GRU units in red. Each layer is described with corresponding formulation for a single character at position t in the input sequence.

Character embeddings

First layer in the ANN learns character embeddings by mapping the discrete representation of each character label to a vector in continuous space. Perhaps, the best known approach to embeddings is word2vec for

words, but the idea can be applied to characters as well. Each integer is turned into a one-hot encoded vector $\boldsymbol{c}$ with the length of the dictionary $|\boldsymbol{c}|$ and multiplied by the embedding weights matrix $\boldsymbol{W}_c$ of size $|\boldsymbol{e}| \times |\boldsymbol{c}|$, i.e. $\boldsymbol{e} = \boldsymbol{W}_c\boldsymbol{c}$. Since this operation effectively selects a row from the matrix, we may simply select the row corresponding to the integer directly to speed up the computation using the TensorFlow's `tf.nn.embedding_lookup` function. The character embedding is implemented in the `_build_embedding_rnn` method of the `ThaiWordSegmentationModel` class:

```
embedding_weights = \
  tf.Variable(tf.random_uniform([vocabulary_size, state_size], -1.0, 1.0))
embedding_vectors = tf.nn.embedding_lookup(embedding_weights, tokens)
```

where the weights matrix $\boldsymbol{W}_c$ is initialized uniformly from [-1,1]. The resulting embedding vector $\boldsymbol{e}$ has length $|\boldsymbol{e}|$ which corresponds to state size of the RNN cells in the next layer.

Recurrent layer

In the second layer, we implement a bi-directional RNN. In general, RNN adds a recurrence to an ANN by taking in input vector $\boldsymbol{x}_t$ at position t in a sequence and state vector $\boldsymbol{h}_{t-1}$ from the previous step, where $\boldsymbol{h}_0$ is typically initialized with zeros. In this way, RNN can use information from previous steps which is crucial to tasks where consequtive elements in a sequence share relevant information. In our case, we feed the embedding vectors $[\boldsymbol{e}_t]_{t=1}^T$ of a sentence to the forward direction RNN and get T state vectors $[\boldsymbol{h}_t]_{t=1}^T$ of length $|\boldsymbol{e}|$ from each step as output. Second component in the layer is the backward RNN which accounts for the dependencies that come after the character at index t. This can be achieved simply by reversing the input sentence. TensorFlow provides function `tf.nn.bidirectional_dynamic_rnn` to unroll the RNN dynamically at each training iteration to length T. To clarify the notation, we denote the forward state vector by $\overrightarrow{\boldsymbol{h}}_t$ and the backward state vector by $\overleftarrow{\boldsymbol{h}}_{T-t+1}$. The output matrix of the layer consists of the stacked state vectors $\boldsymbol{H} = [\overrightarrow{\boldsymbol{h}}_1, \overleftarrow{\boldsymbol{h}}_T; \ldots; \overrightarrow{\boldsymbol{h}}_T, \overleftarrow{\boldsymbol{h}}_1]$.

ANNs are often trained with stochastic gradient descent or a related method. Weights in the ANN are adjusted based on their contribution to the prediction error using backpropagation algorithm which iteratively finds the weights that minimize the error. However, vanilla RNNs suffer frequently from the vanishing gradient problem, where the amount of adjustment needed for the weights becomes very small due to multiple consequtive multiplications. This effectively prevents training weights further away from the error function as tiny adjustments do not change them enough. A common solution to this problem is to replace the simple logic of vanilla RNNs with a long short-term memory (LSTM) units. In our case, we computed the state vectors with gated recurrent units (GRU), which are faster to train due to the fewer number of parameters but has been said to be on a par with LSTM in performance. Formulas for these are given in the model architecture figure.

The layer contains also dropout which randomly selects a set of weights with some probability in GRU cells not to be trained. This is to avoid overfitting the model to the data which is a common problem in deep learning. The above put together is provided in the `_build_embedding_rnn` method as well:

```
cell = tf.nn.rnn_cell.GRUCell(state_size)
cell = tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=1-dropout)
(forward_output, backward_output), _ = \
    tf.nn.bidirectional_dynamic_rnn(cell, cell, inputs=embedding_vectors,
                                    sequence_length=lengths, dtype=tf.float32)
outputs = tf.concat([forward_output, backward_output], axis=2)
```

Output layer

The last layer is a fully connected network which provides a $T \times 2$ score matrix, $\boldsymbol{S} = \boldsymbol{H}\boldsymbol{W}_s + \boldsymbol{b}_s^\top$. Each $\boldsymbol{S}_t$. is a score vector for the boundary and non-boundary labels. The weight $\boldsymbol{W}_s$ has dimension $2|\boldsymbol{e}| \times 2$

and the bias vector $\boldsymbol{b}_s$ length 2. To obtain probabilities for the boundaries in the sentence for each label $i \in$ ["boundary", "non-boundary"], the scores are fed to the softmax function

$$\boldsymbol{P}_{\cdot i} = \frac{\exp(\boldsymbol{S}_{\cdot i})}{\sum_{i'} \exp(\boldsymbol{S}_{\cdot i'})}.$$

Since there is a single degree of freedom, the last layer could provide a single probability for each character as well. The implementation is provided in the `_build_classifier` method:

```
logits = tf.layers.dense(inputs=inputs, units=num_output_labels, activation=None)
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=logits)
```

Prediction error is measured with the cross entropy loss function

$$\boldsymbol{L} = -\sum_i \boldsymbol{y}_i \log \boldsymbol{P}_{\cdot i},$$

where $\boldsymbol{y}_i$ contains the true boundaries of label i in the sentence. The loss is calculated with `tf.nn.sparse_softmax_cross_entropy_with_logits` in TensorFlow. We do not want to include predictions for the padded characters in the total loss, so we remove them from $\boldsymbol{L}$ by creating a mask with `tf.sequence_mask` and applying it with `tf.boolean_mask`. The loss per character for the sentence j of length T_j is then $L_j = \frac{1}{T_j} \sum_{t=1}^{T_j} L_{jt}$. The loss and the predicted sequences are returned by the `_build_classifier` method:

```
mask = tf.sequence_mask(lengths)
loss = tf.reduce_mean(tf.boolean_mask(losses, mask))
masked_prediction = tf.boolean_mask(tf.argmax(logits, axis=2), mask)
masked_labels = tf.boolean_mask(labels, mask)
```

Model training

The network is feeded in a loop with batches of sentences randomly selected from the training dataset and trained with the Adam optimizer, which is an adaptive stochastic gradient descent (SGD) algorithm. The optimizer is defined in the `_build_optimizer` method:

```
global_step = tf.Variable(0, trainable=False)
learning_rate = tf.placeholder(tf.float32, shape=[])
optimizer = tf.contrib.layers.optimize_loss(loss=loss, global_step=global_step,
    learning_rate=learning_rate, optimizer='Adam')
```

where `global_step` keeps track of the current training iteration. The hyperparameters include the state size of the GRU units, learning rate and dropout probability which are set in `train.py`. The learning rate is provided as a TensorFlow variable so it can be changed during training. Batch size (number of sentences) equals usually the maximum number of sentences that the GPU memory can hold considering the length of the longest sentence.

To maximize utilization of the GPU, the data is read and prepared with CPU in parallel while training the model with GPU using TensorFlow's Dataset API. Reading the data is done with the `_parse_record`, `_read_training_dataset`, `_read_training_dataset` methods and initialization of the batch iterators over the data with `_init_iterators`.

Model validation

Performance of the word segmentation model is measured with the F_1 score, which is the harmonic mean of precision and recall,

$$F_1 = 2\frac{\text{"precision"} \cdot \text{"recall"}}{\text{"precision"} + \text{"recall"}}.$$

Precision is the number of correctly predicted boundaries of all boundaries (true and mispredicted) and recall is the number of correctly predicted boundaries of the true boundaries. Since the F_1 measure is not differentiable, a requirement of the SGD and backpropagation methods, the cross entropy loss is used as the proxy. The F_1 score is calculated both for training and validation datasets to monitor potential overfitting.

Results and inference

We set the GRU state size to 128, learning rate to 0.001, dropout to 0.50 and batch size to 128 sentences and trained for 33,000 iterations overnight with NVIDIA GeForce GTX 1070 – resulting precision 99.04%, recall 99.31% and F_1 score 99.18% on the validation set. Training and validation losses are shown on the figure below and there appears to be no indication of overfitting. However, a more throughout test is required to confirm this. There is an example in `predict_example.py` how to infer word boundaries from input text.

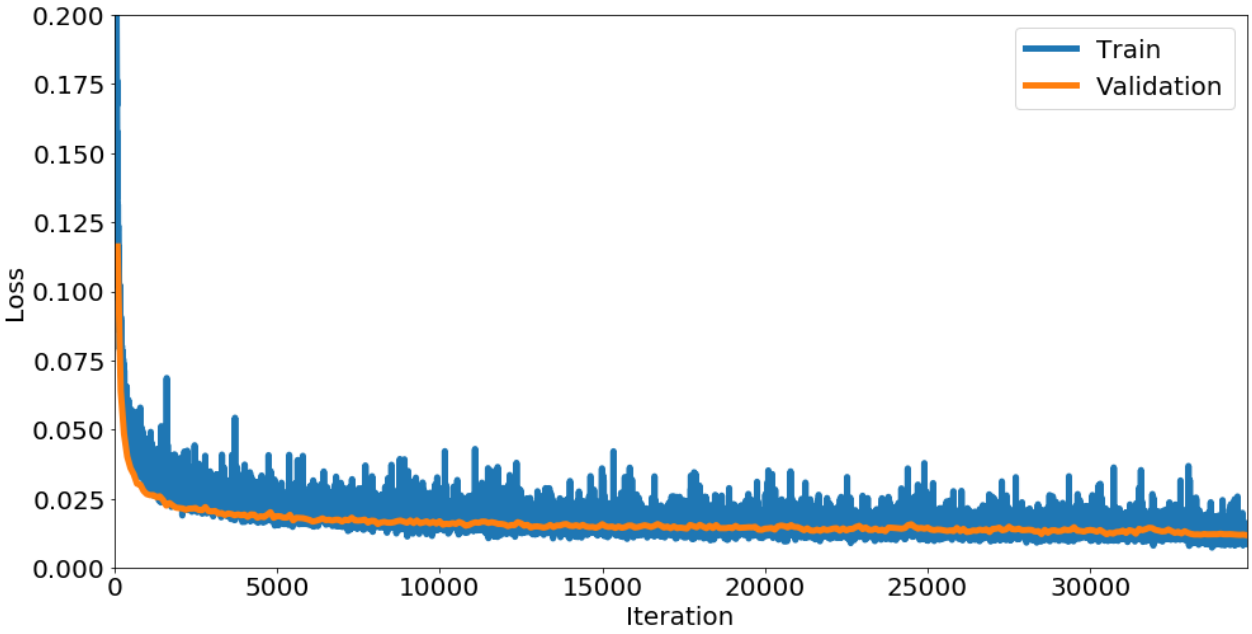

Figure 3: Training and validation losses at each iteration.

Bottom line

We applied word embeddings and bi-directional RNN to Thai word segmentation. The relatively simple structure of the model provided results that are comparable to the state of the art approaches such as Boonkwan, Suphitni (2018). The strength of the bi-directional RNN is that it needs less feature engineering as opposed to convolutional neural networks (CNN) – another successful approach in deep learning. In the context of NLP, *n*-grams are usually constructed for CNNs to capture dependencies between nearby

characters. The downside of a RNN is that it is slower to train than CNN as it does not parallelize well due to the recurrent nature.

From here, there are multitude of ways improving the model, for example by preparing it for the types of texts that are not covered in the training data, and for spelling mistakes. Furthermore, the model can be extended to higher level NLP tasks such as named entity recognition, and part of speech tagging.

References

Boonkwan P., Supnithi T. (2018) Bidirectional Deep Learning of Context Representation for Joint Word Segmentation and POS Tagging. In: Le NT., van Do T., Nguyen N., Thi H. (eds) Advanced Computational Methods for Knowledge Engineering. ICCSAMA 2017. Advances in Intelligent Systems and Computing, vol 629. Springer, Cham