哈爾濱工業大學

HARBIN INSTITUTE OF TECHNOLOGY

Computational Semantics 2017

Project Report

Name: Dongyao Hu (胡东瑶)

Student ID: 1122120218

Instructor: Prof. Wanxiang Che

May, 2017

Project Summary

Semantic slot filling is one of the most important and challenging problems in **natural language understanding (NLU)**, or **spoken language understanding (SLU)**. Slot filling is typically treated as a **sequence classification problem**, which means to convert the user input, S_i, into a task-specific semantic representation of the user's intention, U_i, at each turn.

In many goal-oriented human-machine conversational understanding system, it's necessary to automatically extract the semantic concepts we need. In this project, we try to implement a slot filling model on the well-known **airline travel information system (ATIS)** benchmark. The given training dataset is like below,

```
i       O
want    O
to  O
fly O
from    O
boston  B-fromloc.city_name
at  O
DIGITDIGITDIGIT B-depart_time.time
am  I-depart_time.time
and O
arrive  O
in  O
denver  B-toloc.city_name
at  O
DIGITDIGITDIGITDIGIT    B-arrive_time.time
in  O
the O
morning B-arrive_time.period_of_day
```

Each line of the training data is a pair of the input word (e.g. `boston`) and its corresponding semantic label (e.g. `B-fromloc.city_name`). Notice that all input words have been pre-processed to their lowercase forms, and all numbers have been converted to `DIGIT` flag for convenience. The label in the second column follows the popular in/out/begin (IOB) representation, in which the begin label (`B`) and in label (`I`) consists of many sublabels(`B-x` and `I-x`), according to their semantic concepts (e.g. `departure and arrival cities`).

What we are expected to do is training a model to predict the label when given an input word. There are many approaches to solve this kind of sequence classification problem. Traditional approaches include HMM/CFG, HVS, CRF or SVM. In this project we apply the

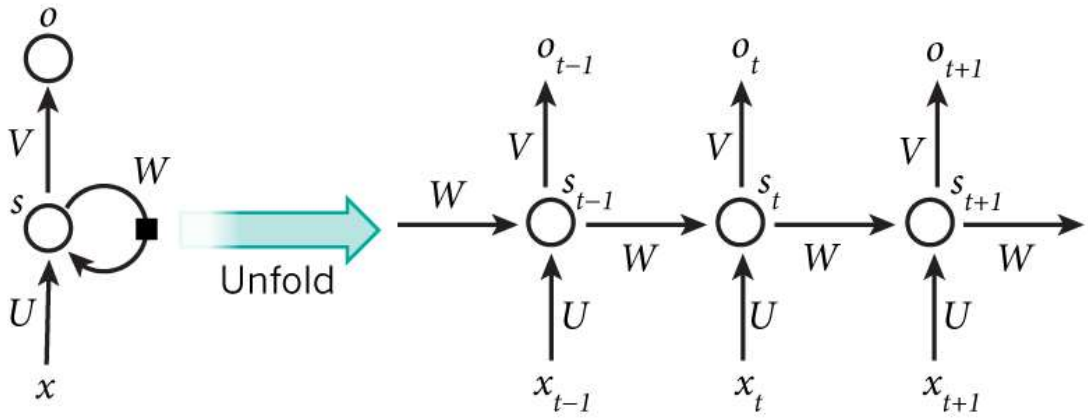
hot **recurrent neural network (RNN)** to solve it.

Principle

Recurrent neural network(RNN) is a class of deep neural network. RNN simply equals to a multi-layer feedforward neural network if we unfold it. The last value of the hidden unit is stored and passed to the current hidden unit.

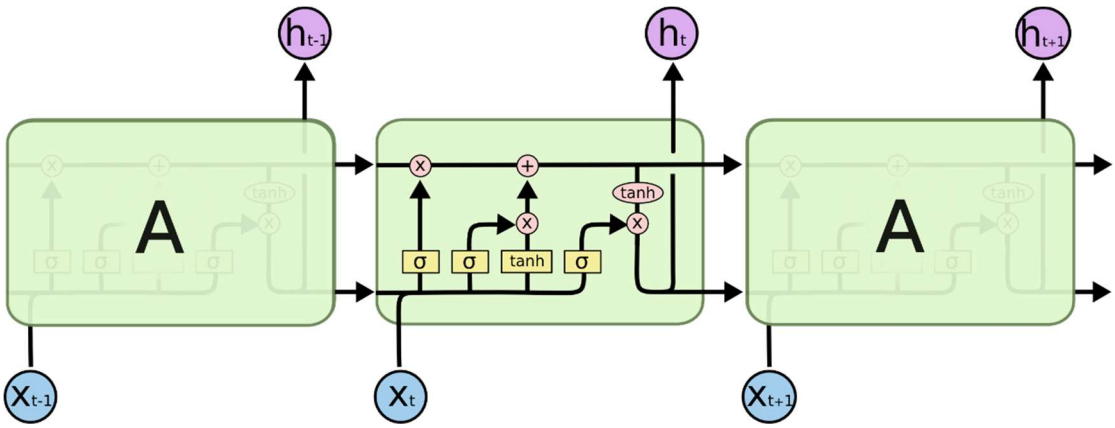
$$s_t = f(Ux_t + Ws_{t-1})$$
$$o_t = Vs_t$$

Where `f` corresponds to the activation function, like `ReLU`. Notice that the weight matrix `U`, `W`, `V` are shared through all time steps.



The `softmax` function will change the output vector to a normalized vector (probability distribution). This predicted output will be compared with the ground truth label. Then we calculate the loss and take the backpropagation algorithm to update three weight matrix `U`, `W`, `V`.

The Simple RNN suffers the problems of gradient vanishing and exploding due to its deep unfolded structure. There are several ways to overcome them, such as using `ReLU` instead of `sigmoid` or `tanh` activation functions. But among them the LSTM, or its related gate structures such as GRU, is the native and effective way to solve the problems.



Besides the simple RNN formula,

$$u_t = \tanh(Wh_{t-1} + Vx_t)$$

We construct three gates to control the flow of the information: the forget gate, the input gate and the output gate:

$$f_t = sigmoid(W_f h_{t-1} + V_f x_t)$$
$$i_t = sigmoid(W_i h_{t-1} + V_i x_t)$$
$$o_t = sigmoid(W_o h_{t-1} + V_o x_t)$$

In LSTM, we will calculate the hidden node state h_t with the usage of three gate above (notice c_t is a temporary variable):

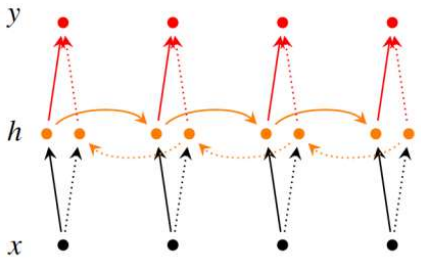
$$c_t = f_t \odot c_{t-1} + i_t \odot u_t$$
$$h_t = o_t \odot \tanh(c_t)$$

Finally, the output is the same as the simple RNN:

$$y_t = Uh_t$$

In this way, the gradient can always be set around 1, thus preventing the vanishing or exploding.

LSTM, however, is similar to the simple RNN and can only memory the historical data but cannot see the future ones. In some situations, this property is reasonable, such as speech recognition, since machine don't know which term people will speak next. But in most cases in NLP, the corpus fed to algorithms is complete, thus enables algorithms to see the future words. Under this condition, we can apply the bi-directional RNN/LSTM to assure a better performance!



The bi-directional RNN/LSTM does nothing more but repeats a process reversely, which makes the RNN/LSTM able to take both past and future information into consideration.

Experiment environment and tools

Linux/macOS
Python 3.6
Keras 2

Detailed Description of the implementation

1) Preprocessing

The original dataset looks like the one shown in the first chapter **Project Summary**. There are 4978 sentences in the training dataset and 893 sentences in the testing dataset.

However, the words and the labels should be converted to their identification numbers if we want to process them. We can count all words/labels and assign each of them a unique index. We store the pairs in a dictionary like this:

Pairs of words and their IDs

```
{'what': 0, 'aircraft': 1, 'is': 2, 'used': 3, 'on': 4, 'delta': 5, 'flight': 6, 'DIGITDIGITDIGITDIGIT': 7, 'from': 8, 'kansas': 9, 'city': 10, 'to': 11, 'salt': 12, 'lake': 13, 'i': 14, 'want': 15, 'go': 16, 'boston': 17, 'atlanta': 18, 'monday': 19, 'need': 20, ...
```

Pairs of labels and their IDs

```
{'O': 0, 'B-airline_name': 1, 'B-flight_number': 2, 'B-fromloc.city_name': 3, 'I-fromloc.city_name': 4, 'B-toloc.city_name': 5, 'I-toloc.city_name': 6, 'B-depart_date.day_name': 7, 'B-cost_relative': 8, 'B-arrive_time.period_mod': 9, 'B-arrive_date.day_name': 10, ...
```

The result of counting shows that there are 572 words and 127 kinds of labels. Here is an example of the first sentence in the training dataset after converting.

Example of converting words/labels to IDs

```
['i' 'want' 'to' 'fly' 'from' 'boston' 'at' 'DIGITDIGITDIGIT' 'am' 'and' 'arrive' 'in' 'denver' 'at' 'DIGITDIGITDIGITDIGIT' 'in' 'the' 'morning']
[ 14  15  11 178   8  17 112  53 202  23  90  69  60 112   7  69  27 111]

['O' 'O' 'O' 'O' 'O' 'B-fromloc.city_name' 'O' 'B-depart_time.time' 'I-depart_time.time' 'O' 'O' 'O' 'B-toloc.city_name' 'O' 'B-arrive_time.time' 'O' 'O' 'B-arrive_time.period_of_day']
[ 0  0  0  0  0  3  0 38 39  0  0  0  5  0 23  0  0 50]
```

2) Constructing the models

Our project uses Python3 and Keras as tools. Keras is a high-level neural networks API capable of running on top of Tensorflow or Theano. Keras' concise APIs encapsulate most of the annoying Tensorflow details thus enables us to implement fast experimentation.

It's very simple to construct a neural network in Keras. We first generate a Sequential() model class, then just simply add various layer classes on the model. After the model is compiled, it will be able to train, to predict and to analyze.

In this project, we construct four kinds of RNN models. The codes and comments are shown below:

Model One: Simple RNN

```
def model_1(input_dim, output_dim):
```

```
    model = Sequential()
    model.add(Embedding(input_dim=input_dim, output_dim=100))
        # Turns positive integers (indexes) into dense vectors of fixed size.
    model.add(Dropout(rate=0.25))
    model.add(SimpleRNN(units=100, return_sequences=True))
        # Fully-connected RNN where the output is to be fed back to input.
    model.add(TimeDistributed(layer=Dense(units=output_dim,
activation='softmax')))
        # This wrapper allows to apply a layer to every temporal slice of an
input.
    model.compile('rmsprop', loss='categorical_crossentropy')
        # RMSProp optimizer is usually a good choice for recurrent neural
networks.
    return model
```

Simple RNN are already powerful enough. It can take historic information into account. However simple RNN cannot see the future information. There are various methods to solve it. We simply insert a convolution layer to let the RNN see the short-range future and past around the present word by blurring the local temporal information.

Model Two: RNN with a convolution layer

```
def model_2(input_dim, output_dim):
    model = Sequential()
    model.add(Embedding(input_dim=input_dim, output_dim=100))
    model.add(Conv1D(filters=64,       kernel_size=5,       padding='same',
activation='relu'))
        # This layer creates a convolution kernel over a single spatial (or
temporal) dimension to produce a tensor of outputs.
    model.add(Dropout(rate=0.25))
    model.add(SimpleRNN(units=100, return_sequences=True))
    model.add(TimeDistributed(layer=Dense(units=output_dim,
activation='softmax')))
    model.compile('rmsprop', loss='categorical_crossentropy')
    return model
```

We also implement the LSTM and Bi-LSTM models since they could be constructed easily in Keras! The principle has been introduced in the second chapter **Principle**, and the codes below are human-readable enough.

Model Three: LSTM model

```
def model_3(input_dim, output_dim):
    model = Sequential()
    model.add(Embedding(input_dim=input_dim, output_dim=100))
    model.add(LSTM(units=100, return_sequences=True))
```

```
    model.add(Dense(units=output_dim))
    model.add(Activation('softmax'))
    model.compile('rmsprop', loss='categorical_crossentropy')
    return model
```

Model Three: Bi-LSTM model

```
def model_4(input_dim, output_dim):
    model = Sequential()
    model.add(Embedding(input_dim=input_dim, output_dim=100))
    model.add(Bidirectional(LSTM(units=100, return_sequences=True)))
    model.add(Dense(units=output_dim))
    model.add(Activation('softmax'))
    model.compile('rmsprop', loss='categorical_crossentropy')
    return model
```

The program will predict the labels for each term in the test dataset after every epoch. The output result will be organized in the format below. The first column represents the words waiting to be predicted, the second column the ground-truth, the third column the predicted labels.

Ground-truth and predicted labels in the output file

```
BOS O O
i O O
would O O
...
charlotte B-fromloc.city_name B-fromloc.city_name
to O O
las B-toloc.city_name B-toloc.city_name
vegas I-toloc.city_name I-toloc.city_name
that O O
...
in O O
st. B-stoploc.city_name B-stoploc.city_name
louis I-stoploc.city_name I-stoploc.city_name
EOS O O
```

The program will use the standard script conlleval.pl to scan this formatted file and calculate the precision rate, recall rate and F1-score. We use a small python script to extract these three scores and print them after every epoch. The total training process looks like this:

Training process

```
Epoch 77
Training =>
```

```
100%
|################################################################################|
Loss = 0.001497703276492683, Precision = 99.96, Recall = 99.96, F1 = 99.96
Validating =>
100%
|################################################################################|
Loss = 0.2513775741931784, Precision = 95.07, Recall = 95.44, F1 = 95.25
Best validation F1 score = 95.25
```

Result and conclusion

We have tested four models. During the 100 epochs over the train dataset, the program will save the weights once the highest F1-score is found in test dataset. The related scores are listed below:

	Model 1 Simple RNN	Model 2 RNN with Conv	Model 3 LSTM	Model 4 Bi-LSTM
Precision	92.70	94.32	93.13	95.07
Recall	93.39	94.63	93.39	95.44
F1	93.05	94.47	93.26	**95.25**

The results indicate that **1)** RNN-based methods are very powerful. Especially the 95.25% F1-score in Model 4 is almost the same as the state-of-the-art reported in 2016. **2)** LSTM (including the bi-direct LSTM), due to its gate mechanism preventing the gradient from vanishing or exploding, obviously outperform the simple RNN.

Reference

Chilamkurthy, S. Keras Tutorial - Spoken Language Understanding.

Mesnil, G., He, X., Deng, L., & Bengio, Y. (2013). Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In Interspeech (pp. 3771–3775).

Mesnil, G., Dauphin, Y., Yao, K., Bengio, Y., Deng, L., Hakkani-Tur, D., ... others. (2015). Using recurrent neural networks for slot filling in spoken language understanding. IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP), 23(3), 530–539.

Che, W., Lec05 – Recurrent Neural Network. Lecture Slide of Deep Learning 2017.